CSEE 6863 Formal Verification
Final Project
Verification of a N-Body Simulator Hardware Accelerator

Kristian Nikolov (kdn2117)
Linxiao Wu (lw3227)

Contributions

Kristian Nikolov (kdn2117): Initial work on TCL scripts, work on assertions and assumptions, fixing bugs & fixing black-boxing of shift registers using parameters. Work on report & presentation.

Linxiao Wu (lw3227): Writing the assertions, assumptions and cover properties for shift_register and n_body, detecting design bugs in FSM. Work on report & presentation.

Project Overview

The core of the project was a n-body simulation hardware accelerator made for a final project for Embedded Systems. The primary module is an FSM with three main states: Read/Write, Calculate Acceleration, and Update position. The overall flow is as follows: starting in the Read/Write state the user provides information about the number of bodies, their initial positions, masses and velocities, as well as the number of timesteps to be computed. Once they raise the go signal, the state transitions to Calculate Acceleration, where the bodies' position information is fed into the module getAccl. This pipelined module is made up of intel floating point IP blocks and shift registers, and after a delay it outputs the acceleration of one body due to another every clock cycle. Finally, once all the accelerations have been calculated and added to the respective body's velocity, the state transitions to update position. This state is much simpler and simply loops over every body, adding their velocity to their position. From here, we either loop back to Calculate Acceleration again if we have not yet run through the number of timesteps specified, or go to Read/Write and raise the done signal if we have.

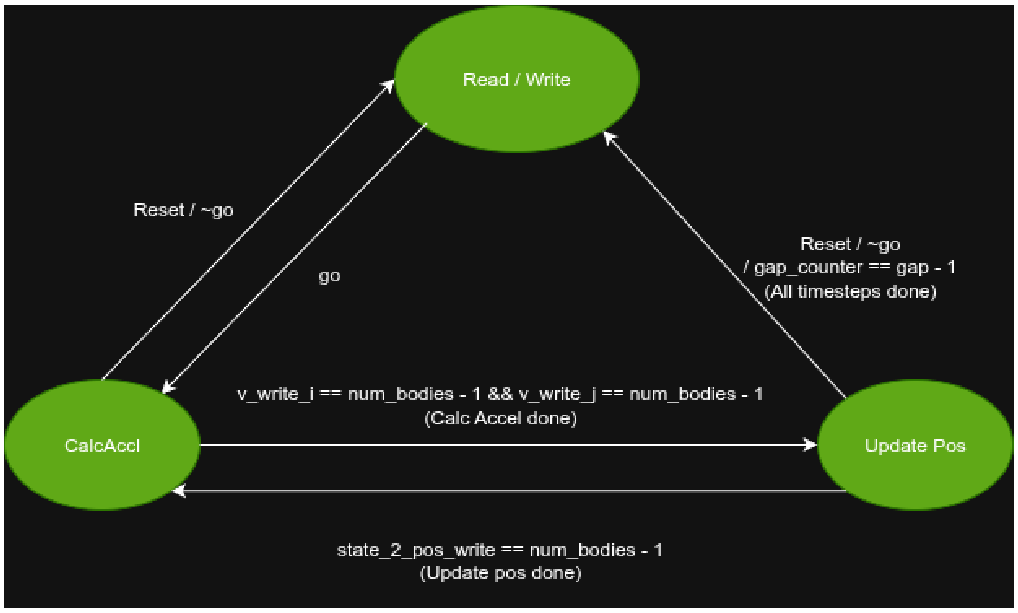


Figure 1: High Level overview of the FSM

For this project, the previous approach to verify functional correctness was based on simulation. Specifically, the initial values of *N* bodies were written into RAM, followed by running simulations and verifying the output. This was done through several testbenches. Although the design behaves correctly under these test conditions, this does not provide sufficient evidence of overall functional correctness, and leaves the potential for unfound errors. Therefore, our project focused on completely verifying the functionality of the N-body simulation hardware accelerator's FSM.

Notes on how to run:

The code is attached to the submission, and can be found in the github repository here:
https://github.com/mthrndr/n-body-sim
The files with the SystemVerilog code, and various assertions, assumptions and cover cases are found in the hardware subdirectory and are nbody.sv, getAccl.sv and shift_register.sv
Running the Jasper Gold scripts is quite simple. In hardware/verification you can run either nbody.sh or shift_register.sh.

Verification

Verification of the Shift Register

Shift register is a module that can be directly synthesized, and is written in SystemVerilog. The getAccl module instantiates this module four times to support the function of pipeline computation. Therefore, it is also important to verify this module. The structure of this module is shown below in Figure 2, with the shift register columns in green.

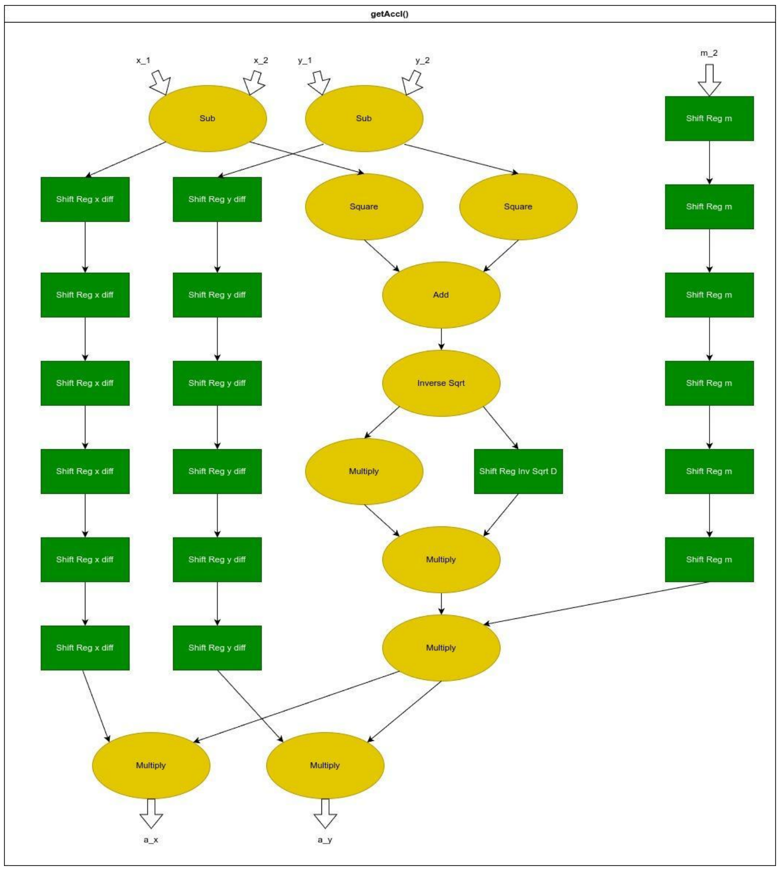


Figure 2: getAccl module

Checking Properties

The control signal of the shift register should be reasonable for the whole system. We wrote some assumption properties in the top module to constrain the control signal of the shift register.
We implemented the following assertion properties to check whether the shift register's functional correctness:

	briefly description	comments
a_reset_clears_out:	When the rst signal is high, output of the shift register should be set to zero.	
a_stage0_loads_in:	Stage0 loads previous input each cycle (when not in reset)	Shown below
a_stage_shifts	Each stage shifts from previous stage (when not in reset)	This assertion is implemented to verify whether the data shift

		from one the [n-1] registers to [n] correctly.
a_delay_correct	End-to-end delay contract: out == in delayed by SHIFTS cycles	This assertion is set to check the key property of the shift register: the delay between the input and output should equal to the cycles of shifting:
c_in_changes_then_out_changes	(Cover property) If the input changes, the outputs should also change after a certain amount of time.	

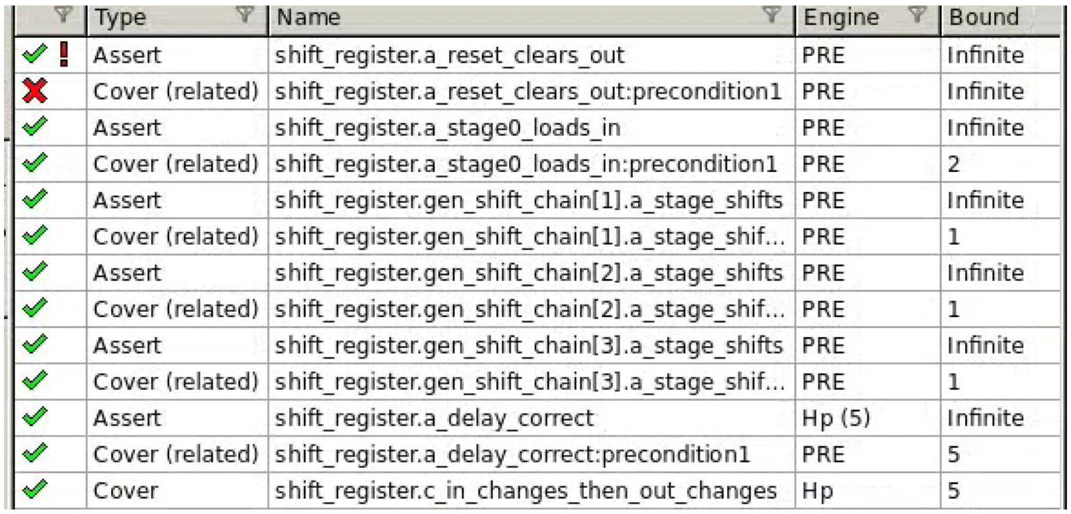
For the second property a_stage0_loads_in:

```
a_stage0_loads_in:
  assert property (@(posedge clk) disable iff (rst)
      (!rst && !$past(rst)) |-> (shift_reg[0] == $past(in))
  );
```

The shift register uses an asynchronous reset. Original assertions using $past(...) did not account for reset boundaries. When reset was asserted, registers were cleared to zero while $past(in) still referred to pre-reset values, causing valid assertion failures. Therefore, assertions were rewritten to be reset-aware, checking behavior only when both the current and previous cycles were not in reset.

Jasper Gold results

All the assertions are proven correct. The reset_clears_out assertion hasn't been checked because during the verification process, the reset signal is never set to high.



	Type	Name	Engine	Bound
✓ !	Assert	shift_register.a_reset_clears_out	PRE	Infinite
✗	Cover (related)	shift_register.a_reset_clears_out:precondition1	PRE	Infinite
✓	Assert	shift_register.a_stage0_loads_in	PRE	Infinite
✓	Cover (related)	shift_register.a_stage0_loads_in:precondition1	PRE	2
✓	Assert	shift_register.gen_shift_chain[1].a_stage_shifts	PRE	Infinite
✓	Cover (related)	shift_register.gen_shift_chain[1].a_stage_shif...	PRE	1
✓	Assert	shift_register.gen_shift_chain[2].a_stage_shifts	PRE	Infinite
✓	Cover (related)	shift_register.gen_shift_chain[2].a_stage_shif...	PRE	1
✓	Assert	shift_register.gen_shift_chain[3].a_stage_shifts	PRE	Infinite
✓	Cover (related)	shift_register.gen_shift_chain[3].a_stage_shif...	PRE	1
✓	Assert	shift_register.a_delay_correct	Hp (5)	Infinite
✓	Cover (related)	shift_register.a_delay_correct:precondition1	PRE	5
✓	Cover	shift_register.c_in_changes_then_out_changes	Hp	5

Figure 3: Output of the shift register assertions

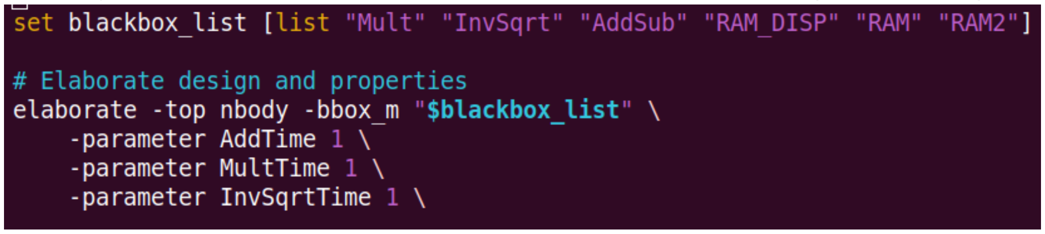
As we learned during the presentation, when the reset signal is set in the tcl script, it is treated as a unique input, and cannot be used in a cover. Thus for the final code this assertion is commented out.

Verification of the FSM

N-Body TCL Setup

The N-body simulation hardware accelerator integrates several external Intel IP blocks. As these are provided externally by Intel, and our analysis is not focused on the mathematical correctness of the results, it is unnecessary to verify them. Therefore, we decided to blackbox those modules. Instead, the verification effort will focus entirely on the correctness of FSM state transitions, and the timings of the loops therein.

Additionally, as we will discuss more in depth later, several of the pipeline delays are parameterized in order to facilitate switching between various IP Blocks. In the implementation of the pipelined computation, the delay of these blocks also determines the depth of the shift registers. By setting the delay time of each computation module to a minimum value, we not only significantly reduce runtime, but even prevent JG from blackboxing the shift registers.



```
set blackbox_list [list "Mult" "InvSqrt" "AddSub" "RAM_DISP" "RAM" "RAM2"]

# Elaborate design and properties
elaborate -top nbody -bbox_m "$blackbox_list" \
    -parameter AddTime 1 \
    -parameter MultTime 1 \
    -parameter InvSqrtTime 1 \
```

Figure 4: Relevant portions of the n-body tcl script

Assumptions

Verifying the functionality of the FSM requires that the input control signals be set reasonably. To do so we use the following assumptions:

1. Flag for configuration done:

This logic tracks whether the software has written the required configuration registers N_BODIES and GAP. Two flags, seen_nb and seen_gap, are used to record if each register has ever been written since reset.

2. Write should only happens under the stage SW_READ_WRITE

```
  as_write_only_inRW_except_GO:
assume property (@(posedge clk) disable iff (rst)
  (chipselect && write && (addr[15:9] != GO)) |-> (state == SW_READ_WRITE)
);
```

Initially, we wrote a strong assumption:

```
  assume property (@(posedge clk) disable iff (rst)
  chipselect && write |-> state == SW_READ_WRITE
);
```

However, the assumption over-constrains bus writes, accidentally blocking the write that would deassert GO, so GO stays high.

3. Go cannot be set to high before the configuration is done
4. After configuration, the num_bodies should be limited.

This assumption should only happen after the configuration is done, otherwise there will be a conflict during the configuration, as the num_bodies will be reset. Additionally, the minimum number of bodies must exceed the getAccl pipeline depth in order for the timing logic to behave correctly.

5. Configuration parameters num_bodies and gap remain unchanged while the accelerator is running
6. Whenever the hardware asserts done, the software will eventually issue a read request (read_sw) within 1 to 500 clock cycles.
7. Read is set high only in the stage SW_READ_WRITE

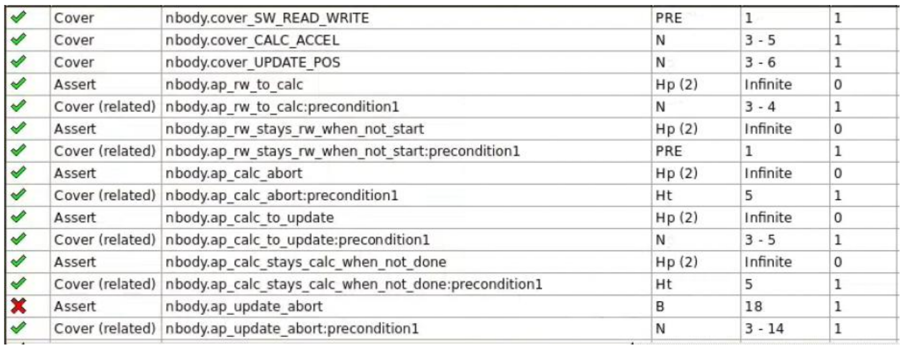
Assertions

After setting reasonable assumption properties, we check all transitions of the three states to verify that all the transitions that happen in the system are legal.

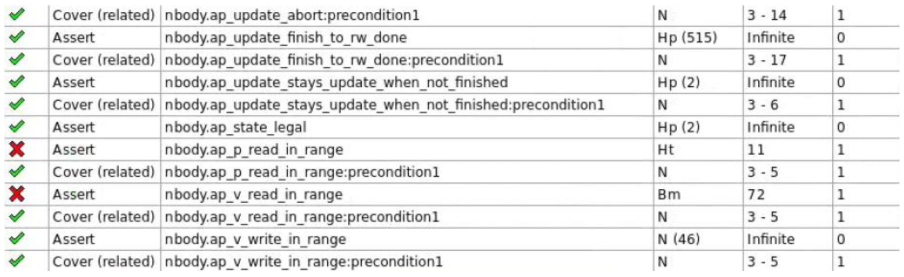
Check if pointers are in legal range

In addition to checking the FSM, we can also check the pointers p_read_i, p_read_j, v_read_i, v_read_j, v_write_i, v_write_k. These are used to loop over various values for the bodies and either read them out of or write them into memory. Since the states use multiple of these loops, and they are staggered by pipelined stages, a significant portion of debugging during the original development was spent fixing off-by-1 errors with them. Verifying their correctness is crucial.

Jasper Gold results



✓	Cover	nbody.cover_SW_READ_WRITE	PRE	1	1
✓	Cover	nbody.cover_CALC_ACCEL	N	3 - 5	1
✓	Cover	nbody.cover_UPDATE_POS	N	3 - 6	1
✓	Assert	nbody.ap_rw_to_calc	Hp (2)	Infinite	0
✓	Cover (related)	nbody.ap_rw_to_calc:precondition1	N	3 - 4	1
✓	Assert	nbody.ap_rw_stays_rw_when_not_start	Hp (2)	Infinite	0
✓	Cover (related)	nbody.ap_rw_stays_rw_when_not_start:precondition1	PRE	1	1
✓	Assert	nbody.ap_calc_abort	Hp (2)	Infinite	0
✓	Cover (related)	nbody.ap_calc_abort:precondition1	Ht	5	1
✓	Assert	nbody.ap_calc_to_update	Hp (2)	Infinite	0
✓	Cover (related)	nbody.ap_calc_to_update:precondition1	N	3 - 5	1
✓	Assert	nbody.ap_calc_stays_calc_when_not_done	Hp (2)	Infinite	0
✓	Cover (related)	nbody.ap_calc_stays_calc_when_not_done:precondition1	Ht	5	1
✗	Assert	nbody.ap_update_abort	B	18	1
✓	Cover (related)	nbody.ap_update_abort:precondition1	N	3 - 14	1



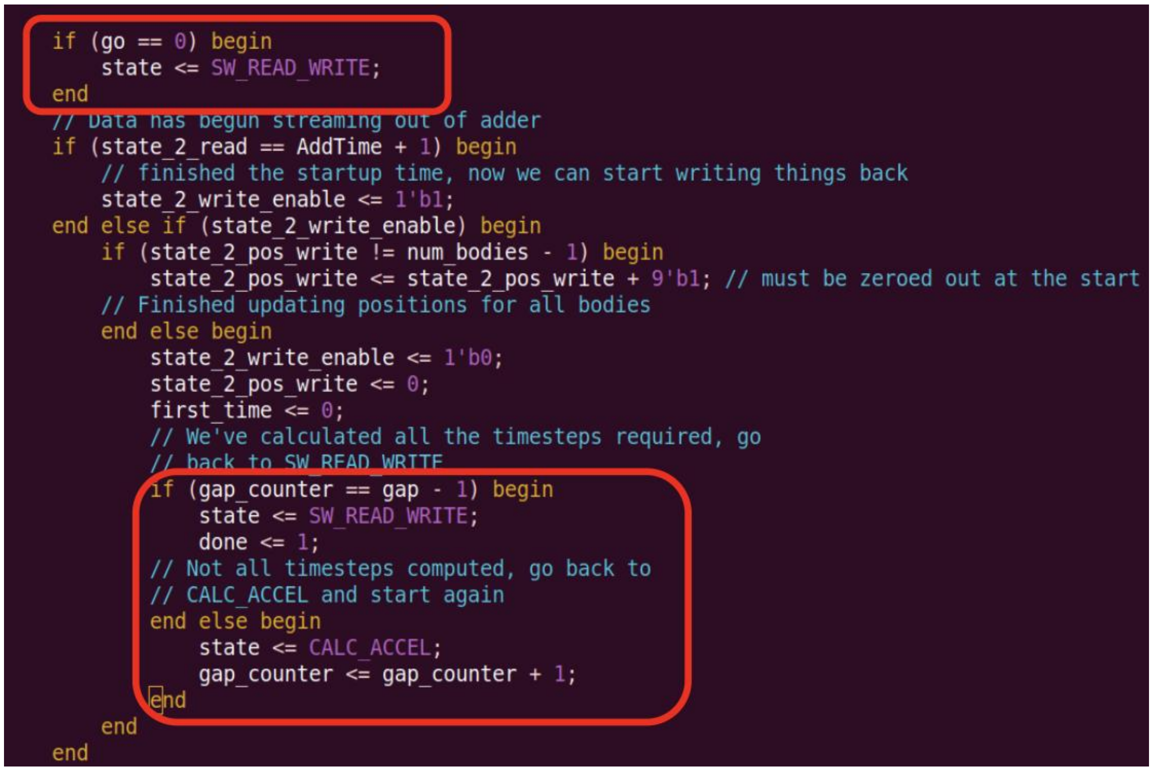
✓	Cover (related)	nbody.ap_update_abort:precondition1	N	3 - 14	1
✓	Assert	nbody.ap_update_finish_to_rw_done	Hp (515)	Infinite	0
✓	Cover (related)	nbody.ap_update_finish_to_rw_done:precondition1	N	3 - 17	1
✓	Assert	nbody.ap_update_stays_update_when_not_finished	Hp (2)	Infinite	0
✓	Cover (related)	nbody.ap_update_stays_update_when_not_finished:precondition1	N	3 - 6	1
✓	Assert	nbody.ap_state_legal	Hp (2)	Infinite	0
✗	Assert	nbody.ap_p_read_in_range	Ht	11	1
✓	Cover (related)	nbody.ap_p_read_in_range:precondition1	N	3 - 5	1
✗	Assert	nbody.ap_v_read_in_range	Bm	72	1
✓	Cover (related)	nbody.ap_v_read_in_range:precondition1	N	3 - 5	1
✓	Assert	nbody.ap_v_write_in_range	N (46)	Infinite	0
✓	Cover (related)	nbody.ap_v_write_in_range:precondition1	N	3 - 5	1

Figure 5: Initial n-body Jasper Gold Results

The formal verification results reveal several illegal behaviors in the system, even though the design passed multiple simulation-based testbenches. **These issues highlight limitations of simulation and demonstrate the advantages of systematic formal verification.**

1. ap_update_abort:

Missing a single else statement here leads to a scenario where go == 0, state is set to SW_READ_WRITE, and in the same clock cycle, we have finished updating positions and go back to calc_accel, which is an invalid transition.



```
if (go == 0) begin
    state <= SW_READ_WRITE;
end
// Data has begun streaming out of adder
if (state_2_read == AddTime + 1) begin
    // finished the startup time, now we can start writing things back
    state_2_write_enable <= 1'b1;
end else if (state_2_write_enable) begin
    if (state_2_pos_write != num_bodies - 1) begin
        state_2_pos_write <= state_2_pos_write + 9'b1; // must be zeroed out at the start
    // Finished updating positions for all bodies
    end else begin
        state_2_write_enable <= 1'b0;
        state_2_pos_write <= 0;
        first_time <= 0;
        // We've calculated all the timesteps required, go
        // back to SW_READ_WRITE
        if (gap_counter == gap - 1) begin
            state <= SW_READ_WRITE;
            done <= 1;
        // Not all timesteps computed, go back to
        // CALC_ACCEL and start again
        end else begin
            state <= CALC_ACCEL;
            gap_counter <= gap_counter + 1;
        end
    end
end
```

Figure 6: Faulty code highlighted

However, this illegal behavior is difficult to detect through simulation unless a very specific input sequence is applied. In contrast, formal verification systematically explores all possible signal orderings and correctly detects the missing FSM transition.

2. Pointer range:

The assertions ap_p_read_in_range and ap_v_read_in_range fail due to an off-by-one counter overflow in the CALC_ACCEL state.

When both index counters reach num_bodies − 1, the inner counter resets to zero and the outer counter is incremented without an upper-bound check. As a result, the outer counter temporarily becomes equal to num_bodies, which is outside the valid range [0, num_bodies − 1] for one cycle.

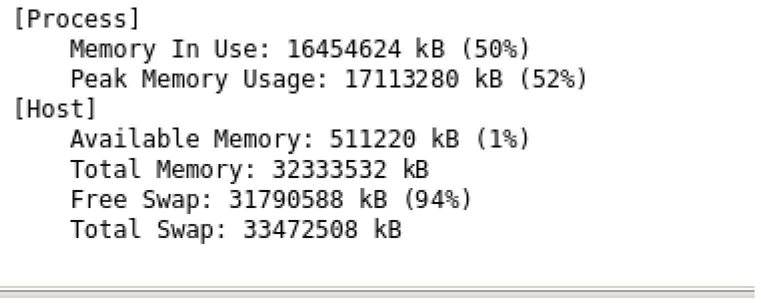
Analysis

By implementing the Formal Verification, we detected several invalid behaviors in the system, which hadn't been discovered through simulation testing. For the transition of states in the FSM, the errors can only be detected under specific input conditions, so the prior testbenches weren't able to detect them. However, with Formal Verification, the Jasper Gold tool helps to apply every possible input set to explore every possible state of the system, thus potential errors can be more easily detected compared to the simulation.

Issues and Discovery

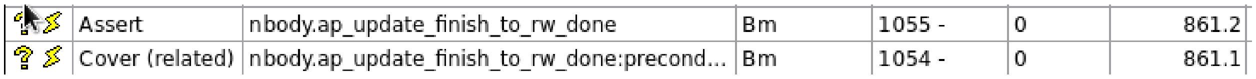
Automatic Black-boxing and Resource Utilization

Quite early on we noticed that Jasper was black-boxing the shift-register module when nbody.sv was being tested. We left it alone for a while, as the shift-register was being verified separately. However we eventually noticed that one of our assertions was failing due to a faulty assumption. This was the fact that we had set the minimum number of bodies to 1, when in reality it had to be set to the minimum latency of the getAccl module + 1. Once this assumption was updated, several of the assertions began taking significantly longer to process, as well as using up all the memory (16GB+) available to them on the remote servers.



```
[Process]
    Memory In Use: 16454624 kB (50%)
    Peak Memory Usage: 17113280 kB (52%)
[Host]
    Available Memory: 511220 kB (1%)
    Total Memory: 32333532 kB
    Free Swap: 31790588 kB (94%)
    Total Swap: 33472508 kB
```

Figure 7: Memory used



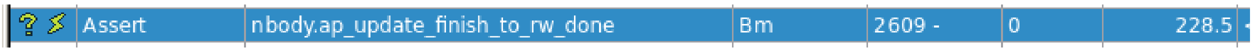
	Assert	nbody.ap_update_finish_to_rw_done	Bm	1055 -	0	861.2
	Cover (related)	nbody.ap_update_finish_to_rw_done:precond...	Bm	1054 -	0	861.1

Figure 8: High runtimes of some assertions (over 14 minutes)

In order to remedy this, we took advantage of the parameterization written into the top module, and set the various floating point block delay times to 1 cycle. Although this wouldn't work in a version that actually used the blocks, since we had blackboxed them it wasn't an issue. This not only reduced the runtime and memory usage, but also stopped Jasper Gold from automatically black-boxing the shift register instances.

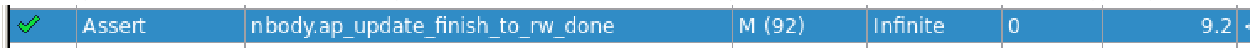
Proof Orchestration

Another interesting aspect of Jasper Gold that we observed was the proof orchestration. Jasper Gold runs several engines concurrently, and occasionally settles on one while others are running. The exact mechanisms behind this are proprietary and we can only guess at exactly how they work, but it is interesting nonetheless to note the effectiveness of certain engines. Some of the same assertions we discussed in the previous section (namely *ap_update_finish_to_rw_done*) ran for several minutes, trying to be solved using bounded-model checking. This was taking a significant amount of time, as expected, as even the reduced sized shift-registers still lead to an expansive potential state space. However after a while it immediately switched to its standard model checking engine results and showed a processing time of only 9.2 seconds.



? ⚡	Assert	nbody.ap_update_finish_to_rw_done	Bm	2609 -	0	228.5

Figure 9: Assertion running for several minutes using a Bounded Model checking engine



✓	Assert	nbody.ap_update_finish_to_rw_done	M (92)	Infinite	0	9.2

Figure 10: The same assertion as in Fig 9 after switching to a Model Checking engine and finishing within 9.2 seconds

This exact process is obscured by the software, and re-running the same tcl script several times can lead to slightly different models being chosen as the final accepted version, and at different times.

Conclusions and Further work

The project proved not only capable of detecting previously unfound bugs in a project that seemed stable during testing and presentation, but led to several interesting interactions with Jasper Gold. If we were to continue to work on the project, there are several interesting avenues that could be explored. First, trying to check mathematical correctness could be accomplished through several techniques. One of these would be Direct Program Insertion (DPI) which would allow us to directly inject C code to mimic the IP blocks functionality, as well as their delays. Another would be to replace the IP blocks with non-optimized HDL code. Beyond just verifying mathematical correctness, it would also be interesting to try and experiment with various engines and their settings, both to compare their run-times and resource utilization.

Appendix

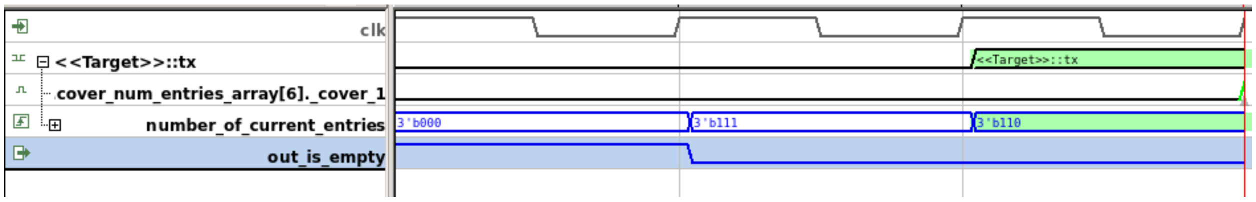
Extra work: FIFO tasks

Task 1)

A) Refer to the attached code.

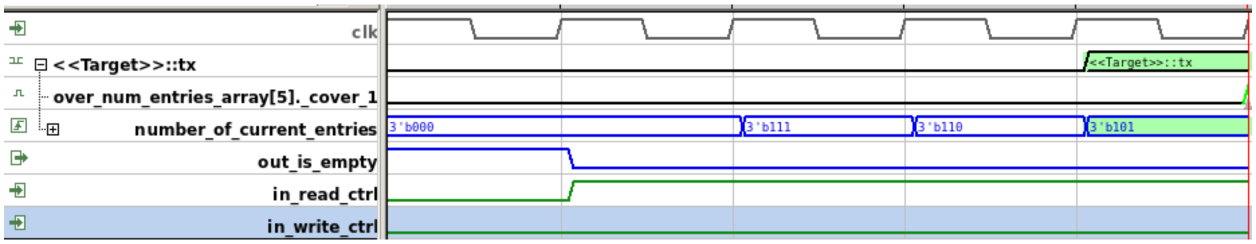
B) Since we have a 4 deep FIFO, only cover properties that check if *number_of_current_entries* is between 0 and 4 should be coverable.

C) It appears that we are able to cover *number_of_current_entries* equal to greater than 4. Analyzing the waveforms we can see that even when the *out_is_full* is True, if we receive a read signal, the number of entries can increment. Additionally, when we have 0 entries, and we receive a write signal, we are able to overflow to having 7 entries in the FIFO.

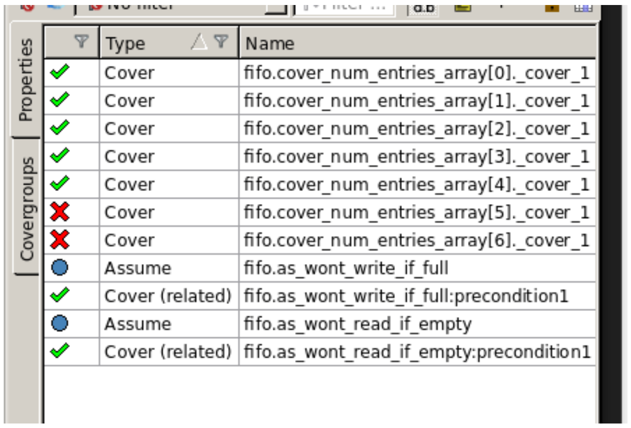


D) Refer to the attached code.

E, F & G) Although we have now prevented overflow, the new bug is that if on a given clock posedge we receive neither read nor write command, both out_is_empty and out_is_full are set to 0. Thus the trace that allows us to cover number_of_current_entries == 5 or 6 first waits a clock cycle with both in_read_ctrl and in_write_ctrl set to 0, then overflows.



H) The bug is fixed by simply removing the final else if statement in the primary always_ff block. Refer to the attached code.
I) Once the code has been fixed, the covers for number_of_current_entries == 0, 1, 2, 3, 4 are coverable, and those for 5 and 6 are not.

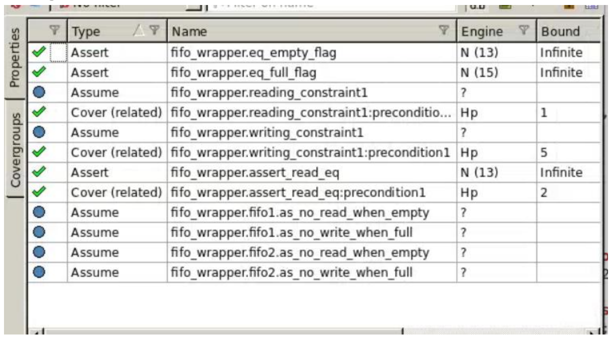


	Type	Name
✓	Cover	fifo.cover_num_entries_array[0]._cover_1
✓	Cover	fifo.cover_num_entries_array[1]._cover_1
✓	Cover	fifo.cover_num_entries_array[2]._cover_1
✓	Cover	fifo.cover_num_entries_array[3]._cover_1
✓	Cover	fifo.cover_num_entries_array[4]._cover_1
✗	Cover	fifo.cover_num_entries_array[5]._cover_1
✗	Cover	fifo.cover_num_entries_array[6]._cover_1
●	Assume	fifo.as_wont_write_if_full
✓	Cover (related)	fifo.as_wont_write_if_full:precondition1
●	Assume	fifo.as_wont_read_if_empty
✓	Cover (related)	fifo.as_wont_read_if_empty:precondition1

Task 2)

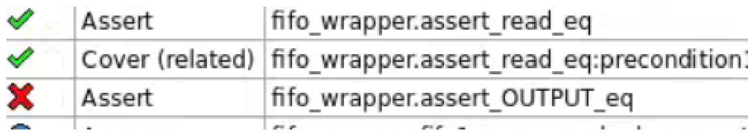
A)
To compare the output of two FIFOs, the following properties are set:

The assertion properties check whether the three outputs of the two fifos are equivalent or not. For *out_read_data*, the assertion property should be set such that the comparison can only happen after the *in_read_ctrl* signal is high. “##1” means that the comparison happens at the next rising edge of the clk signal.



	Type	Name	Engine	Bound
✓	Assert	fifo_wrapper.eq_empty_flag	N (13)	Infinite
✓	Assert	fifo_wrapper.eq_full_flag	N (15)	Infinite
●	Assume	fifo_wrapper.reading_constraint1	?	
✓	Cover (related)	fifo_wrapper.reading_constraint1:preconditio...	Hp	1
●	Assume	fifo_wrapper.writing_constraint1	?	
✓	Cover (related)	fifo_wrapper.writing_constraint1:precondition1	Hp	5
✓	Assert	fifo_wrapper.assert_read_eq	N (13)	Infinite
✓	Cover (related)	fifo_wrapper.assert_read_eq:precondition1	Hp	2
●	Assume	fifo_wrapper.fifo1.as_no_read_when_empty	?	
●	Assume	fifo_wrapper.fifo1.as_no_write_when_full	?	
●	Assume	fifo_wrapper.fifo2.as_no_read_when_empty	?	
●	Assume	fifo_wrapper.fifo2.as_no_write_when_full	?	

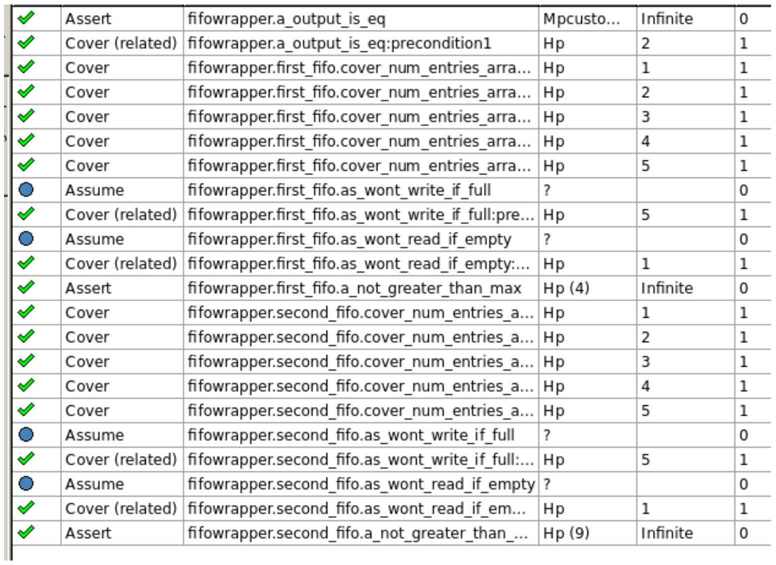
Otherwise,the assert property will fail, because the output in its unstable state was being checked.



✓	Assert	fifo_wrapper.assert_read_eq
✓	Cover (related)	fifo_wrapper.assert_read_eq:precondition1
✗	Assert	fifo_wrapper.assert_OUTPUT_eq

B) The error in Task 1 is re-introduced to fifo2. Additionally, an assertion is added in both fifos to assert that the number of current entries is NOT greater than the maximum.

We would expect this assertion to fail, as it does fail when just testing a single fifo. However, when running fifowrapper this does not fail!



✓	Assert	fifowrapper.a_output_is_eq	Mpcusto...	Infinite	0
✓	Cover (related)	fifowrapper.a_output_is_eq:precondition1	Hp	2	1
✓	Cover	fifowrapper.first_fifo.cover_num_entries_arra...	Hp	1	1
✓	Cover	fifowrapper.first_fifo.cover_num_entries_arra...	Hp	2	1
✓	Cover	fifowrapper.first_fifo.cover_num_entries_arra...	Hp	3	1
✓	Cover	fifowrapper.first_fifo.cover_num_entries_arra...	Hp	4	1
✓	Cover	fifowrapper.first_fifo.cover_num_entries_arra...	Hp	5	1
●	Assume	fifowrapper.first_fifo.as_wont_write_if_full	?		0
✓	Cover (related)	fifowrapper.first_fifo.as_wont_write_if_full:pre...	Hp	5	1
●	Assume	fifowrapper.first_fifo.as_wont_read_if_empty	?		0
✓	Cover (related)	fifowrapper.first_fifo.as_wont_read_if_empty:...	Hp	1	1
✓	Assert	fifowrapper.first_fifo.a_not_greater_than_max	Hp (4)	Infinite	0
✓	Cover	fifowrapper.second_fifo.cover_num_entries_a...	Hp	1	1
✓	Cover	fifowrapper.second_fifo.cover_num_entries_a...	Hp	2	1
✓	Cover	fifowrapper.second_fifo.cover_num_entries_a...	Hp	3	1
✓	Cover	fifowrapper.second_fifo.cover_num_entries_a...	Hp	4	1
✓	Cover	fifowrapper.second_fifo.cover_num_entries_a...	Hp	5	1
●	Assume	fifowrapper.second_fifo.as_wont_write_if_full	?		0
✓	Cover (related)	fifowrapper.second_fifo.as_wont_write_if_full:...	Hp	5	1
●	Assume	fifowrapper.second_fifo.as_wont_read_if_empty	?		0
✓	Cover (related)	fifowrapper.second_fifo.as_wont_read_if_em...	Hp	1	1
✓	Assert	fifowrapper.second_fifo.a_not_greater_than_...	Hp (9)	Infinite	0

This leads us into part C

C) Despite the fact that fifo2 has the error discussed in Task1, we do not seem to find it. This happens due to the fact that both fifo1 and fifo2 share the same inputs! Despite the assertion being very much false when testing it in fifo2 alone, when it is tested together with fifo1, it is unable to reach the overflow. When both *in_read_ctrl* and *in_write_ctrl* are set to 0 during a clock cycle, fifo2’s *out_is_empty_2* is set to 0, however fifo1’s is still 1. Thus fifo1 overconstrains fifo2 and prevents us from finding the error. Adding the error into fifo1 results in both *a_not_greater_than_max* assertions failing!