SystemVerilog Assertions Generator

Gary Mejia
University of California, Santa Cruz
Jack Baskin School of Engineering
Santa Cruz, USA
gmejiama@ucsc.edu

***Abstract*—SystemVerilog Assertions (SVA) Generator is a Scala Class created to add formal verification support for the Chisel SDRAM Controller Generator. A user may need additional reassurance of proper controller behavior and an addition check is via formal verification. Using the provided case class created by the JSON file representing to SDRAM memory module's datasheet, a set of SystemVerilog assertions is generated and injected into the generated RTL.**

***Index Terms*—formal verification, SystemVerilog assertions, register-transistor logic, memories, open-source development**

I. Introduction

Off-chip SDRAM memory modules are seen everywhere in any sort of complex digital circuit. Using DRAM memory is not easy, one can not randomly index at will. Several different timings of the DRAM internals must be respected for the data to be correctly written and read. For this reason, circuits require a memory controller to abstract away these timings so that the outside user can see general random access at will. This project is an extension of the SDRAM Controller Generator, a Scala/Chisel program [1]. It takes in JSON files as inputs and outputs SystemVerilog to describe the behavior of DRAM access. The extension is generating the corresponding SystemVerilog assertions required to support formal verification of several important properties of the controller's behavior. Using SymbiYosys as the front-end for SMT solvers, we have formally verified 16 SDRAM controllers. These controllers target ISSI and Winbond SDRAM memory controllers running at different clock frequencies, CAS latencies, and burst lengths. In formally verifying all specifications, we have fixed major bugs with read and writes not being performed.

A. Motivation

All good hardware designs are also verified. A user is less likely to trust a hardware module with no verification for proper behavior. Chisel generates only synthesizable Verilog. Meaning verification is left to using ChiselTest or ChiselSim. However, these libraries test the FIRRTL intermediate representation of the Verilog target. Additionally, since Chisel is meant for generating Verilog for parameterized hardware, a user must write a Verilog testbench with parameters in mind. These testbenches unfortunately only cover specific cases for a large amount of code.

A middle ground is the use of formal verification. Formal verification is the use of mathematical models to analyze the space of possible behaviors of a design. With the use of SystemVerilog assertions, the SDRAM Controller Generator may use a small amount of code to check several possible cases.

II. Background

This section will go over background on how DRAM memories function, an overview of the SDRAM Controller Generator project, and the basics of SystemVerilog assertions.

A. SDRAM Functionality

The goal of a memory controller is to abstract away the internal workings of the SDRAM memory module. This includes the initialization sequence of the memory module, sending appropriate commands to the memory, and waiting the required time between commands. Fig. 1 shows the internal architecture of a standard SDRAM memory module. We see several inputs used for addressing and sending commands. These address lines feed latches because they are briefly saved. Memory cells are organized in banks.

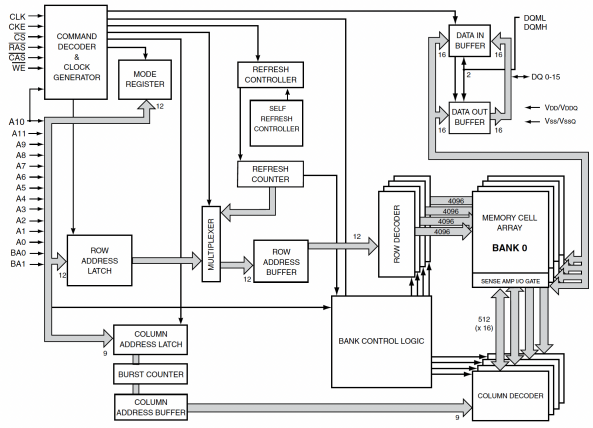

Fig. 1. DRAM Functional Block

For a bank to be accessed, it must be "opened" by an active command. Active commands take in row addresses along with a bank address to pre-charge the corresponding word lines in the provided bank and row. Once this pre-charging completes, a read or write command is sent with the corresponding column to access, and write data in if needed. A read will require an additional latency, known as the column access strobe or CAS latency for read data to be valid. After all this the memory may return to idle to do additional reads or writes.

B. SDRAM Controller Generator

Given the latencies between each steps and the complex interface to send commands to the memory, a finite-state machine helps to abstract access. This finite-state machine is known as the memory controller. For deployment, a user must write RTL in hardware description languages such as SystemVerilog or VHDL. However, these languages have steep learning curves and sparse documentation online. Thus, to remedy this, we have written a Chisel generator aimed at SDRAM memory controllers.

The SDRAM Controller Generator is a Scala/Chisel program. For input it takes in a JSON file describing several parameters found in an SDRAM memory's datasheet. This JSON file is parsed and used to create a Scala case class for use in hardware generation. Its output is a SystemVerilog file with the RTL used to describe the behavior to correctly index the SDRAM memory. However, the user is not provided a SystemVerilog testbench to verify proper behavior. To remedy this, an additional generator is added to generate and inject SystemVerilog assertions into the RTL for use in formal verification.

C. SystemVerilog Assertions

SystemVerilog assertions are statements about a design that a designer is expecting to be true. Compared to software assertions, which are preconditions to program state, SystemVerilog assertions are properties of the design [2]. Instead of preconditions, they are targets for proofs using SMT solvers. This requires special tools. Commercial offerings include Cadence Jasper, Mentor Graphics Questa, or Synopsys VCS. YosysHQ offers SymbiYosys as an open source alternative, which is used in this project.

There are two types of SystemVerilog assertions: immediate and concurrent. [3] Immediate assertions are not dependent on a clock edge or reset; they are evaluated on all time steps. This means they are only able to model combinational logic. Below is an example of an immediate assertion.

```
immediate_assertion_name:
assert (current_state != 0)
else
    $error("%m checker failed");
```

The first line is a block name used by the formal verifier to indicate where an assertion failed. Second is the actual assertion itself, followed by an option else statement with an error message for the checker. Given that immediate assertions are aimed at combinational logic, they are seldom used in favor of concurrent assertions.

Concurrent assertions are tied to clock edges and resets. This allows them to connect strings of events together over time and more expression. Below is an example of a concurrent assertion.

```
concurrent_assertion_name:
assert
    property (@(posedge clk)
        disable iff (rst)
        req |-> ##3 gnt)
else
    $error(
    "%m no grant after request"
    );
```

Like the immediate assertion, the beginning line is a label. The key difference is in the assertion; properties are specifications to be held true and checked by formal tools. It is sampled at the positive edge of the clock and disabled if and only if reset is asserted high. To connect events through time the "req $|->$ ##3 gnt" string means: if req is high then 3 cycles after, gnt must be true. This assertion in short, describes a specification to be checked for.

D. SystemVerilog Assumptions

Since assertions are proven by a formal tool, the cone of influence may include impossible or irrelevant cases. To tighten the cone of influence SystemVerilog has the **assume** keyword. Assumptions have similar syntax to concurrent assertions and work in a similar manner. Instead of exiting the formal verifier, assumptions ignore failure, effectively reducing the cone of influence in the design. As seen in section IV, when assertions fail it is generally due to an impossible case being found.

E. Tooling

As stated before, YosysHQ's Symbiyosys is used as the frontend for SMT solvers. To read System Verilog assertions, SymbiYosys uses a "FORMAL" SystemVerilog prepressor block. An example before showcases the block:

```
`ifdef FORMAL
\\asserts in between
`endif
```

All assertions are placed at the end of the module. This block allows SymbiYosys to separate assertions from the rest of the design. In addition, SymbiYosys interfaces with several SMT solves including yices2 and abc. A .sby file can be used to automate runs.

```
[options]
mode bmc
depth 20
[engines]
smtbmc yices
[script]
read -formal SDRAMController.sv
prep -top SDRAMController
[files]
SDRAMController.sv
```

The script above is the default script used in our experiments. We chose bounded model checking as we do not check for liveness properties, only safety. Depth is kept at 20, which is the default set by SymbiYosys. For the SMT solver we picked yices2 as this is used in courses at UCSC. The rest of

the script is commands to read and evaluate the assertions in the required files.

III. Memory Controller Specifications

The SDRAM memory controller has the following specification:

1) In the initialization state, the state must transition to the idle state after 100 microseconds plus 3 cycles have passed
2) After the state leaves the initialization state, the state must not transition to this state once it leaves it
3) In the idle state, if a read or write request is sent then transition to the active state after 1 cycle
4) In the active state, the state must transition to the read or write state after the appropriate amount of cycles have passed
5) In the read state, the state must transition to the idle state after $CAS + 2^{\text{burst length}}$(burst length is between 0 and 3), cycles have passed
6) In the read state, the controller must set read data valid signal high after CAS cycles have passed
7) In the write state, the state must transition to the idle state after $2^{burstlength}$ cycles have passed

IV. SystemVerilog Assertion Generation

Chisel on its own only generates synthesizable RTL. We created a new class called "SVA_Modifier" that takes in a string representing a file path and SDRAM parameter case class. The class has member functions to start formal blocks, end formal blocks, and write in assertions into the RTL.

To begin a formal block, the RTL file is opened with Scala IO functions, read into an array, and has the last two lines deleted. By default, Chisel ends all RTL with the string "endmodule\n". The begin formal block function replaces these final two lines with "'ifdef FORMAL". Ending a formal block appends "endmodule\n" back into the RTL file.

A. Assertion Construction

Since the case class with all SDRAM parameters is fed into "SVA_Modifier", class all required cycle latencies are accessible. Scala string interpolation is used to place parameterized latencies into assertion properties and assumptions. Once these strings are constructed they are written back into the RTL file.

B. Assertion Injection

As stated before, Chisel generated synthesizable RTL. Verilog is the targetted HDL and Chisel emits Verilog .v files. Since SystemVerilog assertions are required to be in SystemVerilog .sv files, the generated file has its file extension converted to .sv using a file name change Scala function. Once this occurs, the main program creates an instance of the "SVA_Modifier" class with the new .sv file's path as a string argument and the SDRAM parameter case class. The begin formal block function is first called, to remove the last 2 lines of the .sv. Any proceeding function calls are to functions that write in assertions and assumptions. After these calls, the formal end block function is called to complete the file.

V. Assertion Debugging

How are these assertions debugged? When an assertion is inserted by itself, the formal tools check every possible case. If these cases are found to be invalid cases, assumptions are used to filter out these cases. Section V-A will go over an example of creating the active to read or write assertion.

A. Active to Read or Write Assertion Example

As stated in section III, the specification for the active to read or write state transition is as follows: **in the active state, the state must transition to the read or write state after the appropriate amount of cycles have passed**. Figure 2 is the generated SystemVerilog assertion for this property. On its own, this assertion immediately fails.

```
assert property (@(posedge clock) disable iff (reset) (io_state_out == 3) |-> ##4 ((io_state_out == 6) | (io_state_out == 7)));
```

Fig. 2. Generated assertion for active to read or write

When the assertion fails, the SMT solver outputs the counterexample trace. This trace is the sequence of events that caused the assertion to fail. Figure 3 shows the first counterexample with the bare SystemVerilog checked as is.

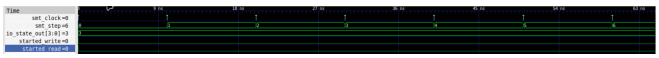

Fig. 3. First counterexample trace - no read or write signal

The counterexample is read as follows:

- The SMT step and clock are signals made by SymbiYosys to provide the circuit a clock signal and a counter to check how deep the check has one through, our case 20 cycles
- We include the "io_state_out", "started_read", and "started_write" signals as these are our signals of interest
- We see that the state is still at state 3, "started_read" and "started_write" are both low. This indicates that after the 4 cycles of expected latency, the active stayed in the active state. However, as shown in the source code of the idle state, the transition to the active state sets one of two registers that output the signals "started_read" and "started_write" high. Meaning, this case is not possible because arriving in the active state requires either register to hold a high value.

This analysis reveals that this assertion at this point in time is reading an impossible case. To filter out this case, we must assume that either signal is high during this state.

B. Building Assumptions

As seen earlier, we need the tools to exclude the case where the read and write registers hold low values. An assumption state is required to do this. We assume that "started_read" or "started_write" is high during the active state. Figure 4 shows the included assumption for the active to read or write assertion.

However, there is still a counterexample found by SymbiYosys. In figure 5 we see the active to read or write counter in the middle of a count. This is expected behavior. Yet this is a

```
assume property (@(posedge clock) disable iff (reset) (started_read | started_write));
assert property (@(posedge clock) disable iff (reset) (io_state_out == 3) |-> ##4 ((io_state_out == 6) | (io_state_out == 7)));
```

Fig. 4. Assumption 1

counterexample. How is this a counterexample? Our assertion states that if the state is in the active state, the state must be in the read or write state 4 cycles after. Here the state end up in the idle state 4 cycles later. This happens since the tools finds the case where the active state is in the middle of a count with a small burst length. A case like this would cause the state to be in the active state, short circuit the write state, and end up in the idle state. We want to assertion to state that once the state enters the active state, it transitions to the read or write state 4 cycles later. Thus we add in another assumption: we assume the active to read or write state counter starts at 0.

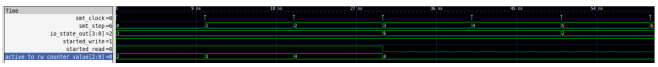

Fig. 5. Second counterexample: active counter is in the middle of counting

In the end for this example we have the following assumptions shown in figure 6. Once this final assumption is included, SymbiYosys finds no counterexamples and the assertion passes.

```
assume property (@(posedge clock) disable iff (reset) (started_read | started_write));
assume property (@(posedge clock) disable iff (reset) (active_to_rw_counter_value == 0));
assert property (@(posedge clock) disable iff (reset) (io_state_out == 3) |-> ##4 ((io_state_out == 6) | (io_state_out == 7)));
```

Fig. 6. Final assumption for active to read or write state specification

VI. Results

Given this program is a hardware generator, there are several designs to explore. We generated 8 memory controllers for ISSI [4] and Winbond [5] memories. Below is a list of the parameters changed:

- Clock Frequencies - ISSI is clocked at 100MHz or 166MHz, Winbond is clocked at 133MHz or 200MHz
- CAS Latency - Both ISSI and Winbond memories support CAS latencies of 2 or 3
- Burst Lengths - All memory controllers have burst lengths from 1,2,4, and 8 items

All 16 designs passed all 7 specifications after the cone of influence had been correctly tightened.

A. Found Bugs

With formal verification we found 2 major bugs: incorrect read and write behavior. Prior to formal verification, we used ChiselTest to test specific behavior of the memory controller. This behavior only checked for data validity after the CAS latency has passed for a read and write. With the assertions included we saw that our specification for read and write behavior did not match. To fix these issues, we added two new counters, read and write state counters. These counters have the sole purpose of checking how long the state is in the read or write state and exit after the appropriate amount of cycles, rather than exiting upon reaching the CAS latency.

B. Final Assertions and Assumptions

Each assertion required at least one assumption to pass. This section will go over each assumption for all other specifications mentioned in section III.

1) Spec 1: Init to Idle: The init to idle specification needed to assume the 100 microseconds counter had finished and 3 cycles passed. This is to remove all the behaviors between that time frame as 100 microseconds in MHz is about 10000 cycles.

2) Spec 2: Never return to init: The never returning to the init state required the assumption that the state is not in the init state. Since the memory controller must reset to the init state and stay there for a set amount of time, we must assume we don't start in the init state for this assertion to be correct. Else we have an obvious counterexample just resetting the controller.

3) Spec 3: Idle to Active: The assumption for the idle to active transition is that a refresh command is not outstanding. This is required due to the finite state machine having experimental support for self-refresh, and delays the transition by an extra cycle. Future work would likely change this assumption.

4) Spec 4: Active to Read or Write: This is the example in section V-A.

5) Spec 5: Read to Idle: The assumption here is the read state counter must start at 0. Else, the SMT solver will find the case of the counter being in the middle and transition prematurely but correctly.

6) Spec 6: Read Data Validity: The assumption for read data being valid after the set CAS latency cycles is the CAS counter starting at 0. Without this assumption the SMT solver will find the case where the CAS counter is in the middle of counting.

7) Spec 7: Write to Idle: The assumption for write to idle transitions is the write state counter starting at 0. This is for the same reason as VI-B6.

C. Challenges

Overall for a simple addition, we faced several challenges. Formal verification for hardware is widely used in industry, but there is a lack of helpful open documentation. Open source tools are also lacking in documentation. While we have access to the source code, any issues with the tools would require lengthy wait times in submitting issues, or us manually fixing them. Additionally, while SymbiYosys is free, the actual formal verification backend is not. Reading counterexamples is not easy, one must analyze whether or not the counterexample is a true bug or an impossible case.

VII. Future Work

Formal verification support is at a good start. Of the 7 specifications, only 1 is a constant property while the rest are transitions. More properties may be related to addition interfaces of the memory controller. As of now, only a basic read-valid interface is supported. If a user wanted to use AXI Xilinx IPs, the generator would need to include an AXI

interface. This new interface would allow more properties to check.

This generator still does not support bi-directional ports. Bi-directional ports are required to get data from the memory module itself, this is due to the physical nature of memories. Chisel supports bi-directional ports with the ”Analog” data type. This data type, unfortunately, is experimental. An additional setback is that bi-directional ports are also an experimental of SymbiYosys. To support the actual dataflow and formal verification, we may look into moving Chisel versions or sending pull requests to YosysHQ.

VIII. Conclusion

While including formal verification support for the SDRAM controller generator sounds simple, there are several challenges. Challenges included lack of documentation, paywalls, and lack of support for several SystemVerilog features. However, the coverage formal verification provides for the memory controller is worth the effort. With a small amount of code we found 2 major bugs with read and write behavior. This small amount of code allows us to effectively generate a testbench for users to fall back on if they don't want to write their own. We showcased these checks on 2 SDRAM memory modules, showing that we are able to apply this verification in the field.

Acknowledgment

I would like to thank Assistant Professor Dustin Richmond for providing me with the evaluation license required to run the formal verification tools. Without this I would not have a project for this course. In addition, I would like to thank Assistant Professor Scott Beamer for guiding me in the initial creation of the SDRAM Controller Generator and the Scala programming language.

References

[1] Gary Mejia, “GitHub - gmejiamtz/sdram_controller_generator: A Chisel hardware generator for SDRAM controllers from their datasheets,” GitHub, 2024. https://github.com/gmejiamtz/sdram_controller_generator (accessed Dec. 14, 2024).
[2] Wikipedia Contributors, “Assertion (software development),” Wikipedia, Nov. 24, 2024.
[3] E. Seligman, T. Schubert, and A. Kiran, Formal Verification. Elsevier, 2023.
[4] ISSI, ”IS42/45S16800F”, Sept 11, 2019
[5] Winbond, ”W9825G6KH”, Dec 17, 2021