Introduction to ASIC Formal Verification

Model Checking with Symbiyosys

Riley Peters

Cal Poly: San Luis Obispo

January 29, 2026

Background

What is Formal Verification?

- The two common approaches to hardware verification are functional and formal verification:

Functional Verification

Defn: Test an implementation behaves the same as its specification by sending many random inputs and comparing the results to a reference. The more inputs tested, the higher the probability that the two are equal.

Formal Verification

Defn: Prove an implementation always behaves in accordance with its specification using some form of mathematical reasoning, e.g. theorem proving, model checking, etc.

- Generally speaking, functional verification is simpler to implement and run, but provides a much weaker guarantee that the system behaves correctly.

What is Formal Verification?

- Consider a simple example: Show that $(x+2)^2 \equiv x^2 + 4x + 4$

Functional Testing

x	$(x+2)^2$	x^2+4x+4	Match?
1	9	9	✓
2	16	16	✓
5	49	49	✓
10	144	144	✓
...	...	...	?

Formal Proof

Proof.

$$(x+2)^2 \equiv x^2 + 4x + 4$$

$$(x+2) * (x+2) \equiv x^2 + 4x + 4$$

$$x^2 + 2x + 2x + 4 \equiv x^2 + 4x + 4$$

$$x^2 + 4x + 4 \equiv x^2 + 4x + 4$$

□

- On the left, each input test increases the likelihood that the two are the same, but we can not be certain unless we try every possible input.
- The proof on the right requires applying mathematical reasoning to show the two are the same, but after doing the work, we can be certain they are the same.

What is Formal Verification?

- Formal verification encompasses a broad range of topics.

Varities of Formal

- Theorem Proving: an engineer proves some property holds for a device manually
- Model Checking: given a device and a set of assertions, have a tool demonstrate that the assertions always hold.
- Equivalence Checking: Prove that two implementations have the same logical behavior
- Symbolic Trajectory Evaluation: Use symbolic simulation to determine how a system behaves over time.
- ... Among other things

- Each area has a variety of useful applications, but today we will focus on the most popular branch for ASIC verification **model checking**.

Model Checking

The Model Checking Problem

Model Checking

Input: An implementation of a device M, set of all states S, and set of properties F.
Goal: Show M satisfies all formulas in F for all reachable states. $\forall s \in S, \forall f \in F : M, s \vDash f$

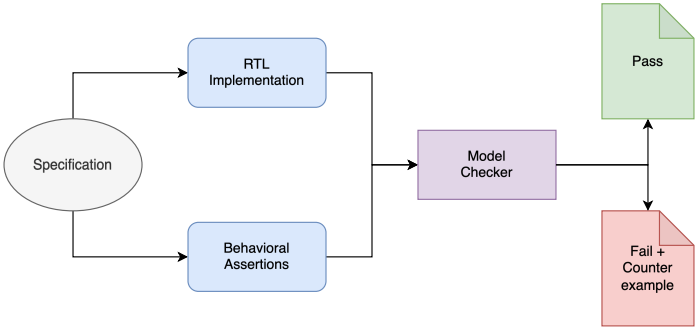


Figure: Model Checking Flow

What Do We Want To Prove?

What should the properties in F show?

- We want to prove that the system never breaks, while also taking action.

This can be described in terms of two fundamental properties [1]:

Safety Properties

Defn: Some bad behavior P never happens: $\square\neg P$

Example: Show that when a CPU interrupts, it will always block any updates to registers and memory from the interrupted instruction.

Liveness Properties

Defn: A good behavior P eventually happens [$\Diamond P$], or always eventually happens [$\square\Diamond P$]

Example: Show that a CPU will always eventually execute an instruction; it will never halt entirely.

What Do We Want To Prove?

Notice that we must show both hold for our system to be correct:

- A device that does nothing will always pass safety checks.
- A device that always does something, even if it is wrong, will pass liveness checks.

Figure: An always red traffic light is safe, but deadlocked.

Figure: An always green traffic light is live, but unsafe.

How Can We Prove It?

We now have a structured way to check if an implementation is correct.

- Having a way to express this is nice, but how do we solve it?
- Ideally, we want some way to determine a solution automatically.

Introducing Satisfiability

Conjunctive Normal Form (CNF)

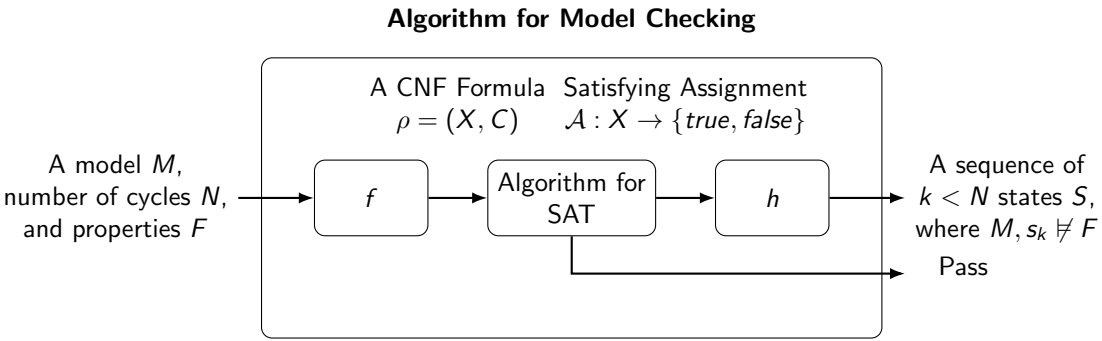
A Boolean formula where variables are in groups (clauses) joined by OR. The clauses are then joined by AND. e.g., $(x_1 \vee \neg x_2) \wedge (x_2 \vee x_3)$

Satisfiability (SAT)

Input: A CNF formula.
Goal: Determine if there is a true/false assignment for the variables that makes the entire formula evaluate to true.

- If such an assignment exists, the formula is **Satisfiable** (SAT).
- If no such assignment exists, it is **Unsatisfiable** (UNSAT).
- We can rephrase an instance of model checking in terms of Satisfiability, use the existing tools to solve the SAT problem, and construct a solution to our original input.

The Reduction: Model Checking $\leq_p$ SAT [4]



- f transforms the model checking problem into a SAT equation where a satisfying assignment means there is a sequence of states where M breaks a property.
- h takes the output and transforms it back into the sequence of states that we can debug.

Bounded Model Checking

- This approach to solving the model checking problem is called **Bounded Model Checking**.

Bounded Model Checking (BMC)

Input: A model M, a set of properties F, and a number of steps N
Goal: Demonstrate that the model never violates the properties for N time steps.

- With this approach, we can prove a system always behaves properly for the first N cycles.

Why use SAT? (Isn't it NP-Complete?)

SAT is provably **NP-Complete**. In the worst case, the time to solve it grows exponentially (2^N). A brute-force check of 100 variables is impossible.

Why Does it Work For Hardware?

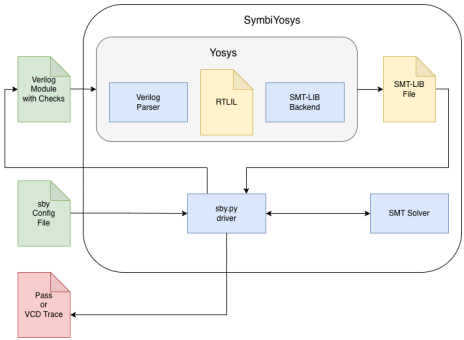
Hardware problems are generally contain certain logical patterns; they are not entirely random.

- **Modern Solvers (CDCL):** Tools like MiniSAT, Z3, and Boolector use intelligent heuristics (Conflict-Driven Clause Learning) to "learn" from mistakes and prune vast sections of the search space.
- **Satisfiability Modulo Theories (SMT):** Most modern tools go beyond SAT and support SMT solving, allowing the tool to efficiently represent formulas with real numbers, data structures, etc.
- While the worst case is bad, verifying hardware logic rarely leads to this case. We can routinely solve problems with millions of variables in seconds.

SymbiYosys

SymbiYosys

- SymbiYosys [3] is an open source tool for running bounded model checks on verilog modules.
- It uses Yosys to transform Verilog modules and assertions into SMT-LIB files used by SAT solvers.
- For a BMC, it repeatedly sends larger and larger instances of the problem until a counterexample is found or the cycle limit is reached.



Getting Started with SBY

- Running SymbiYosys checks requires a .sby file that includes the path to all Verilog files and the types of checks to run.
- When we want to run a check, we can invoke the sby tool on this file.

```
# Run a certain task on the module
sby -f example.sby <task-name>
```

```
[tasks]
bmc      # Run a bounded model check
prove    # Prove module inductively
cover    # Check cover properties

[options]
bmc: mode bmc
prove: mode prove
cover: mode cover
depth 20
expect pass

[engines]
smtbmc boolector

[script]
read -formal example.v
prep -top eaxmple

[files]
rtl/example.v
```

BMC with SymbiYosys

BMC with SymbiYosys

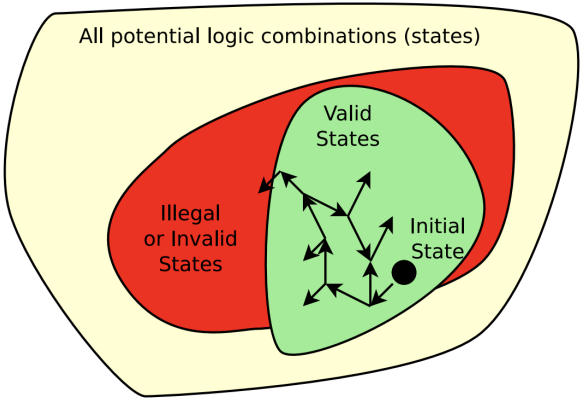
- SymbiYosys adds **assume** and **assert** functions to allow your module to interact with the SAT solver
- **assert**: Defines illegal states, if the solver encounters an assert that is not true the BMC fails and returns a counterexample
- **assume**: Defines unreachable states, these are states that may not necessarily be illegal, but would never happen when running the system.
- Generally we assume the first cycle of a check will be a reset, since we would never run a device without resetting.

```
// Assume we start in the reset state
reg f_past_valid = 0;
always @(posedge i_clk)
    f_past_valid <= 1;
always @(posedge i_clk)
    if (!f_past_valid) assume (!i_rstn);

// Assert that invalid states are
    impossible
always @(*) if (f_past_valid && i_rstn)
begin
    assert(o_cyc || !o_stb);
end
```

Visualizing BMC

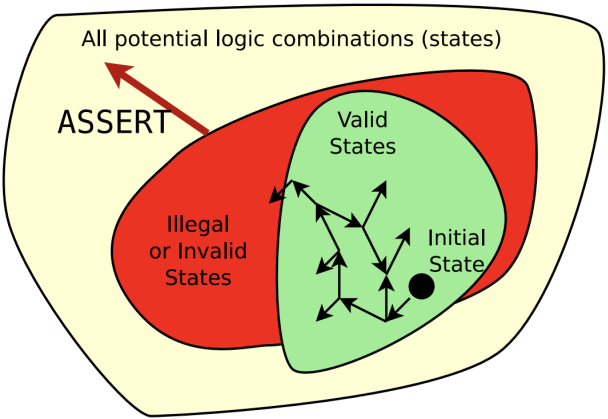
- Consider all possible states your design can be in.



Figures adapted from ZipCPU Verification Tutorial [2].

Visualizing BMC

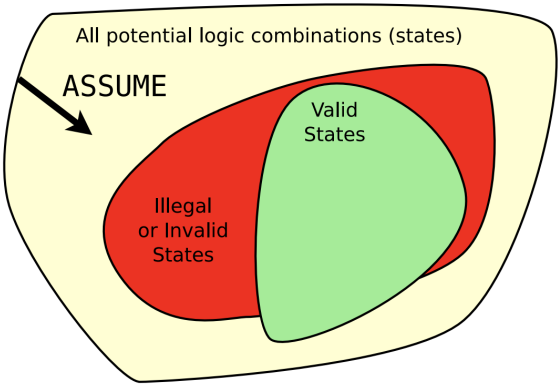
- Assertions expand the set of illegal states.



Figures adapted from ZipCPU Verification Tutorial [2].

Visualizing BMC

- Assumptions restrict the number of states.



Figures adapted from ZipCPU Verification Tutorial [2].

BMC Tips and Tricks

Clocked Assertions

- As is, we have only discussed assertions that rely on the current state
- To define assertions that rely on previous states, SymbiYosys adds the $past function
- **$past(x, N)**: return the value of this expression from N cycles ago. N is 1 by default. Past can only exist inside clocked code blocks.
- Always add a precondition to these assertions that the simulation has run for at least N cycles, otherwise $past returns an undefined value. The solver will use this to break your assertions.

```
// AXI data stability
always @(posedge i_clk) if (
        f_past_valid
    && $past(!i_rst)
    && $past(o_imem_resp_vld)
    && $past(!i_imem_resp_rdy)
) begin
    assert(o_imem_resp_vld);
    assert($stable(o_imem_resp_data));
end

// Note: $stable is the same as:
// o_imem_resp_data == $past(
    o_imem_resp_data)
```

Arbitrary Values and Shadow Logic

- A common method for verifying data integrity is showing that some arbitrary transaction enters and exits the DUT without being corrupted
- SymbiYosys offers the (* anyconst *) and (* anyseq *) modifiers to drive these checks.
- Adding one of these modifiers to a variable tells the solver it can select an arbitrary constant or sequence of values to assign to it.
- With (* anyconst *), the solver can tag an arbitrary input transaction and verify the corresponding output matches.

```
// tells the solver to select an arbitrary
     constant value
(* anyconst *) int f_watch_id;

// track arbitrary data as it passes
    through the fifo
reg                 f_shadow_valid;
reg [XLEN-1:0]      f_shadow_data;

always @(posedge i_clk) begin
    if (!i_rstn) begin
        f_shadow_valid <= 0;
        f_shadow_data  <= 0;
    end else if (
        f_writing
        && (write_count == f_watch_id)
    ) begin
        f_shadow_valid <= 1;
        f_shadow_data  <= s_axis_tdata;
    end
end
```

BMC Limitations

- An important caveat is that BMC only proves the system behaves properly for a finite number of cycles. Ideally, we want to claim that these properties are **never** violated.
- Luckily, we can solve this problem with **induction**.

Unbounded Proofs with K-Induction

K-Induction

- For traditional induction, suppose we want to prove $\forall n : P[n], n \in \mathbb{N}$
- **Base Case**: Show $P[0]$
- **Inductive Step**: Assuming $P[n]$ is true, show $P[n+1]$
- K-Induction applies this same reasoning to model checking for unbounded proofs.

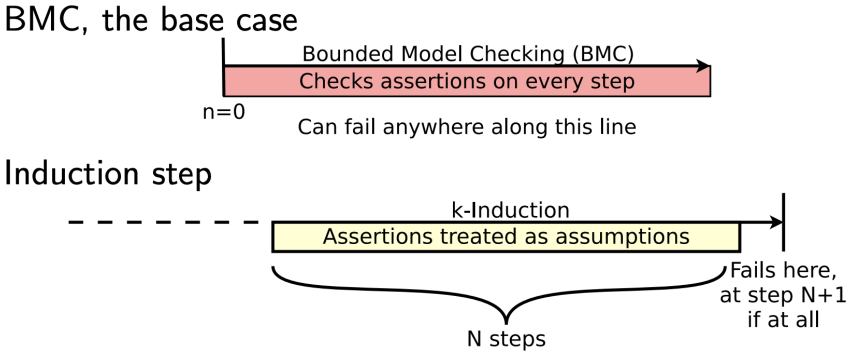
K-Induction

Base Case: Using a BMC, prove the model holds for N cycles.
Inductive Step: Assume you have a valid sequence of N states, show it holds for $N+1$

- For the inductive step, the solver treats the assertions as assumptions and constructs an arbitrary sequence of states, before checking the next step is valid.
- This can become complicated when you run into invalid states.

Visualizing K-Induction

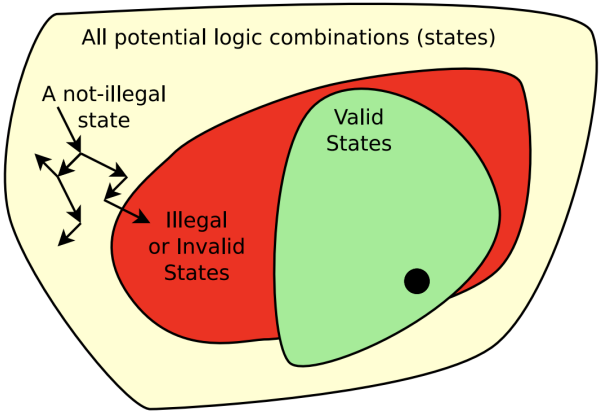
- BMC starts at reset and checks some number of states. The inductive step finds an arbitrary series of states that meet our assertions, and checks to see if the next step is also valid.



Figures adapted from ZipCPU Verification Tutorial [2].

Visualizing K-Induction

- The solver could choose to start in an unreachable state. This would break induction.



Figures adapted from ZipCPU Verification Tutorial [2].

K-Induction Best Practices

Even after you pass BMC, you may fail induction.

- Add assumptions for inputs that should never happen.
- Add stricter assertions to make unreachable states invalid. This may require exposing some of the module's internal signals to force inner registers to never start in an invalid state.
- Different solvers are better at different parts of induction, switching the solver may lead to a pass.
- If all else fails, the base case may be insufficient. Increase the number of cycles the BMC runs.

Reachability with Cover Properties

Cover Properties

Another common question when verifying a design is determining whether a certain condition is reachable.

- Certain types of bugs or too many assumptions may cause the SAT solver to never visit some states.

For instance, consider a simple FIFO:

- How can we ensure that the FIFO can be completely emptied or filled?

SymbiYosys' **cover** properties allow us to quickly verify behavior

Adding Cover Properties

- The **cover** function tells the solver to search for a sequence of states that makes the expression inside its parentheses true.
- During a cover run, the solver returns multiple VCDs that show the sequences of states that cause each cover property to be true. It will mark any covers that can not be satisfied as failing.
- These are effectively the reverse of assertions. Rather than proving that bad behavior never happens, these prove that good behavior can happen at least once.

```
// Cover properties
always @(*) if (!i_rst) begin
    cover(w_queue_is_empty);
    cover(w_queue_is_full);
end
```

Recap Quiz

What do safety properties prove?

What do safety properties prove?

Something bad never happens

What do liveness properties prove?

What do liveness properties prove?

Something good eventually happens

BMC vs. Induction

Which statement accurately describes the difference between Bounded Model Checking (BMC) and Induction?

- BMC is used for Liveness properties, while Induction is used exclusively for Safety properties.
- BMC proves the design is correct for all time by checking all reachable states, while Induction checks only the first k cycles.
- BMC checks that the property holds for k steps, while Induction attempts to prove that if the property holds for k steps, it must hold for step k+1.
- BMC is slower but more thorough, while Induction is a quick approximation.

BMC vs. Induction

Which statement accurately describes the difference between Bounded Model Checking (BMC) and Induction?

- BMC is used for Liveness properties, while Induction is used exclusively for Safety properties.
- BMC proves the design is correct for all time by checking all reachable states, while Induction checks only the first k cycles.
- **BMC checks that the property holds for k steps, while Induction attempts to prove that if the property holds for k steps, it must hold for step k+1.**
- BMC is slower but more thorough, while Induction is a quick approximation.

Assert, Assume, and Cover

Function that defines the set of illegal states. If the expression in parentheses is false, the check fails.

Function that defines the set of illegal states. If the expression in parentheses is false, the check fails.

Assert

Assert, Assume, and Cover

Function that restricts what states the solver can choose. The solver may never make the expression inside its parentheses false.

Function that restricts what states the solver can choose. The solver may never make the expression inside its parentheses false.

Assume

Assert, Assume, and Cover

Function that defines states that should be reachable by the system. If the solver can find a sequence of states that makes the expression inside its parentheses true, the check passes.

Function that defines states that should be reachable by the system. If the solver can find a sequence of states that makes the expression inside its parentheses true, the check passes.

Cover

Clocked Assertions

Why is it necessary to use if (f_past_valid) (or a similar valid signal) when using $past() in an assertion?

- Because $past() is computationally expensive and should be used sparingly.
- To prevent the solver from checking the assertion in the very first cycle, where "past" is undefined.
- Because $past() can only be used on valid AXI streams.
- To ensure the clock is stable before checking data.

Clocked Assertions

Why is it necessary to use if (f_past_valid) (or a similar valid signal) when using $past() in an assertion?

- Because $past() is computationally expensive and should be used sparingly.
- **To prevent the solver from checking the assertion in the very first cycle, where "past" is undefined.**
- Because $past() can only be used on valid AXI streams.
- To ensure the clock is stable before checking data.

Free Variables

In your formal testbench, you declare a configuration signal cfg_thresh as (* anyconst *). How will the SAT solver determine the value of this signal during a proof?

- It will assign a random valid value to cfg_thresh at the start of the run.
- It will iterate through every possible value of cfg_thresh sequentially (0, 1, 2...) until the proof finishes.
- It will treat cfg_thresh as a constant, but it will mathematically search for and select the specific value (if one exists) that causes an assertion to fail.
- It will default the value to 0 to keep the initial state clean.

Free Variables

In your formal testbench, you declare a configuration signal cfg_thresh as (* anyconst *). How will the SAT solver determine the value of this signal during a proof?

- It will assign a random valid value to cfg_thresh at the start of the run.
- It will iterate through every possible value of cfg_thresh sequentially (0, 1, 2...) until the proof finishes.
- **It will treat cfg_thresh as a constant, but it will mathematically search for and select the specific value (if one exists) that causes an assertion to fail.**
- It will default the value to 0 to keep the initial state clean.

Let's Practice!

Formally Verifying an AXI FIFO

- For some hands-on practice, let's try to verify an AXI Stream FIFO formally.
- `https://github.com/rpeters54/AXI-FIFO-Formal` Includes a set of bugged FIFOs and a formal verification scaffold.
- The CARP Docker container should come with all the tools you need to run the lab.
- I included a link to install the tools natively if the docker does not work.

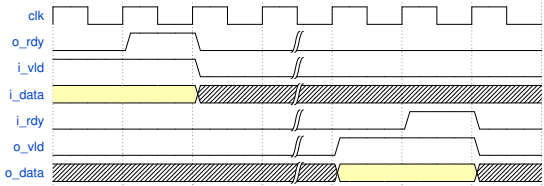


Figure: AXI Write and Read Transaction

Bug 1 - Overfill and Underfill

- The first bugged FIFO acts as if it always has data, but is never full.
- How can we prove that this behavior is incorrect?

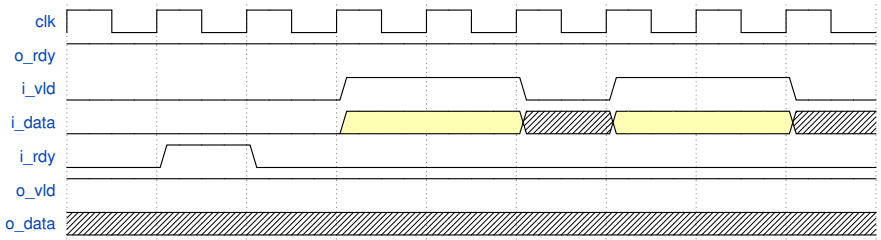


Figure: First Bugged FIFO

Bug 2 - Corrupted Data

- The second bugged seems to begin corrupting data after a few transactions.
- How can we prove that this behavior is incorrect?

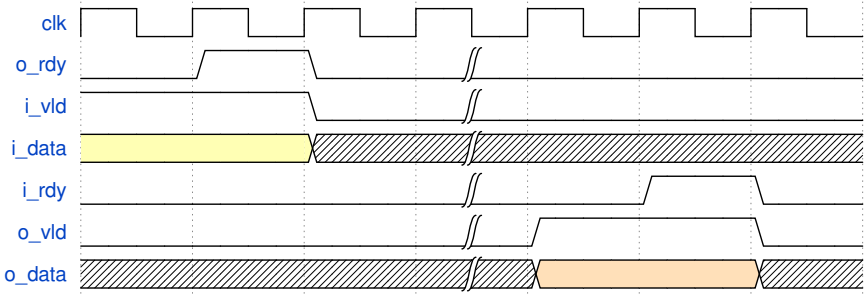


Figure: Second Bugged FIFO

Bug 3 - Dead FIFO

- This FIFO does absolutely nothing.
- How can we prove that this behavior is incorrect?

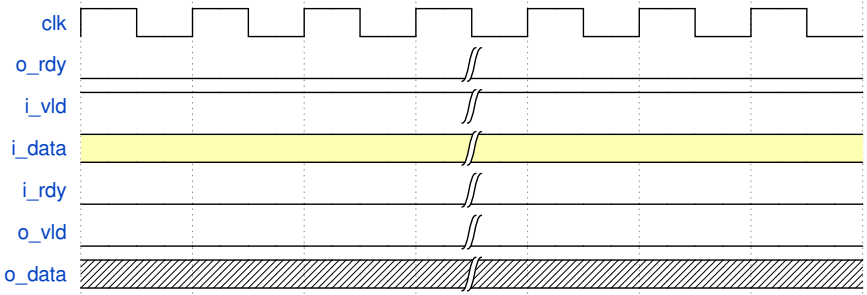


Figure: Third Bugged FIFO

Try Out Induction

- After filling in the missing assertions, check your work against the key to make sure you have everything.
- Below the blanks are some strengthening assertions that make it possible for induction to pass.
- Try out the unbounded proof and verify it passes for the reference model.

References I

B. Alpern and F. B. Schneider.
Defining liveness.
Information Processing Letters, 21(4):181–185, 1985.

D. Gisselquist.
Formal Verification Tutorials.
ZipCPU Blog, `https://zipcpu.com/tutorial/formal.html`.

C. Wolf.
Yosys Open SYnthesis Suite.
`https://yosyshq.net/yosys/`, 2013.

A. Biere, A. Cimatti, E. Clarke, and Y. Zhu.
Symbolic Model Checking without BDDs.
Tools and Algorithms for the Construction and Analysis of Systems, 1999.

Thank You

Questions?