Assertion-Based Formal Debugging During RTL Development

N. Engelhardt

Assertion-Based Formal Verification

Quiz time!

- Who here has formally verified one of their designs before?
- Who has had formal verification done on their design by someone else?
- Who has used a formal tool before even though it wasn't their job?
- Who has a vague idea of what formal verification is?

Background

- Traditionally, Assertion-Based Formal Verification is done on the finished RTL by a separate team of verification engineers

This talk is about a different use case for formal tools!

- Target audience: FPGA developers with no prior exposure to formal
  - Introduce some use cases where formal might add something to your toolkit
  - Try to show the process of using formal (you can look up the syntax later)
- Goal: develop faster!
  - examples for during development
  - don't need to be 100% sure of everything, this is not the verification stage
  - just some things to try out when stuck

Tools used in this talk

- SBY (formerly SymbiYosys): Formal Property Checker, uses Yosys under the hood
  - Works on netlist representation, i.e. uses **synthesis** semantics
  - Cannot parse simulation-only constructs (e.g. testbenches, UVM), only the DUT
  - Command line oriented tool
  - Open source frontends:
    - Verilog-2005 frontend with a few SV(A) extensions
    - GHDL plugin for VHDL & PSL
  - Commercial frontend: Verific (SystemVerilog & VHDL)
- Language: SystemVerilog
  - This talk will mostly use examples with full SVA syntax (not compatible with open source frontend)
  - It's possible to express the same things with only the immediate assertions from open source frontend, by manually constructing the checker state machine (too much text for slides)
  - Can also do the same in VHDL+PSL with GHDL plugin: stay for the next talk!
  - Code for the examples used in this talk is available here: https://github.com/nakengelhardt/fdf24-examples

What is Assertion-Based Formal Verification?

The basic keywords/statements:

- assume
  - for preconditions *external* to the DUT (assuming behavior of input signals)
  - this is also evaluated in simulation to make sure it's not violated (same as assert, just different message)
- restrict
  - same as assume in FV, but makes it clear to any reader that this is only limiting the states you explore
  - use this to exclude behaviors on internal signals/state
  - this is not checked in simulation
- assert
  - express the guarantees that bad things shouldn't happen
  - produces a counterexample trace if the bad thing can in fact happen
  - also evaluated in simulation
- cover
  - under the hood, same as asserting the inverse
  - produces an example trace of a desired behavior happening

What is Assertion-Based Formal Verification?

Simple example:

```
module dut(input a, input b, output o);
    assign o = a ^ b;
endmodule
```

If `b` is low, `o` will be the same as `a`.

- `assume(b == 1'b0);`
- `assert(o == a);`

Where to insert the properties?

Directly in DUT:

```
module dut(input a,
input b, output o);

assign o = a ^ b;

always_comb begin

      assume(b == 1'b0);

      assert(o == a);

end

endmodule
```

In a TB:

```
module tb();

wire a, b, o;

dut dut_i(.*);

always_comb begin

      assume(b == 1'b0);

      assert(o == a);

end

endmodule
```

Using bind:

```
module dut_check(input
a, input b, input o);

always_comb begin

        assume(b ==
1'b0);

        assert(o == a);

end

endmodule

bind dut dut_check
dut_check_i(.*);
```

Configuring SBY

```
module dut(
input a, b,
output o
);

assign o = a ^ b;

always_comb begin

      assume(b == 1'b0);

      assert(o == a);

end

endmodule
```

```
[options]
mode bmc
depth 1

[engines]
smtbmc yices

[script]
read -sv dut.sv
prep -top dut

[files]
dut.sv
```

```
sby -f dut.sby
SBY 15:53:42 [dut] engine_0: smtbmc
SBY 15:53:42 [dut] base: starting process "cd dut/src; yosys -ql
../model/design.log ../model/design.ys"
SBY 15:53:42 [dut] base: finished (returncode=0)
SBY 15:53:42 [dut] prep: starting process "cd dut/model; yosys -ql design_prep.log
design_prep.ys"
SBY 15:53:42 [dut] prep: finished (returncode=0)
SBY 15:53:42 [dut] smt2: starting process "cd dut/model; yosys -ql design_smt2.log
design_smt2.ys"
SBY 15:53:42 [dut] smt2: finished (returncode=0)
SBY 15:53:42 [dut] engine_0: starting process "cd dut; yosys-smtbmc --presat
--unroll --noprogress -t 1  --append 0 --dump-vcd engine_0/trace.vcd --dump-yw
engine_0/trace.yw --dump-vlogtb engine_0/trace_tb.v --dump-smtc engine_0/trace.smtc
model/design_smt2.smt2"
SBY 15:53:42 [dut] engine_0: ##   0:00:00  Solver: yices
SBY 15:53:42 [dut] engine_0: ##   0:00:00  Checking assumptions in step 0..
SBY 15:53:42 [dut] engine_0: ##   0:00:00  Checking assertions in step 0..
SBY 15:53:42 [dut] engine_0: ##   0:00:00  Status: passed
SBY 15:53:42 [dut] engine_0: finished (returncode=0)
SBY 15:53:42 [dut] engine_0: Status returned by engine: pass
SBY 15:53:42 [dut] summary: Elapsed clock time [H:MM:SS (secs)]: 0:00:00 (0)
SBY 15:53:42 [dut] summary: Elapsed process time [H:MM:SS (secs)]: 0:00:00 (0)
SBY 15:53:42 [dut] summary: engine_0 (smtbmc) returned pass
SBY 15:53:42 [dut] summary: engine_0 did not produce any traces
SBY 15:53:42 [dut] DONE (PASS, rc=0)
```

Under the hood

The circuit, the assumptions/restrictions, and the assertions/covers are all transformed into equations expressing relations between the signal values.

Variables: a_{t0} = value of signal `a` in timestep `t0` etc

Base model:

$$o_{t0} = a_{t0} \text{ xor } b_{t0}$$

$$b_{t0} = 0$$

Goal ("bad") state model:

$$o_{t0} \neq a_{t0}$$

This system of equations is handed to a *solver*. The solver either returns a set of values fulfilling all the equations (a counterexample => VCD) or "unsatisfiable" (no such values exist, i.e. the assertion holds => PASS)

```
module dut(
  input a, b,
  output o
);

  assign o = a ^ b;

  always_comb begin

    assume(b == 1'b0);

    assert(o == a);

  end

endmodule
```

Implications

- The solver is very good at finding input combinations you'd never have thought of, because signals have meanings in your mind
- But without good assumptions, the results are mostly useless
    - if any initial value is unconstrained, it will just start in bad state
    - if an invalid input is allowed, GIGO principle applies
- This is where you do have to put in some work
    - but doing it progressively as you develop the RTL is less burdensome than someone who didn't write the code doing it after the fact

What can we do with this functionality?

- using cover statements to create testbenches
- using properties to confirm invariants that the design relies on
- validating subsystem interactions
- bug hunting with assertions

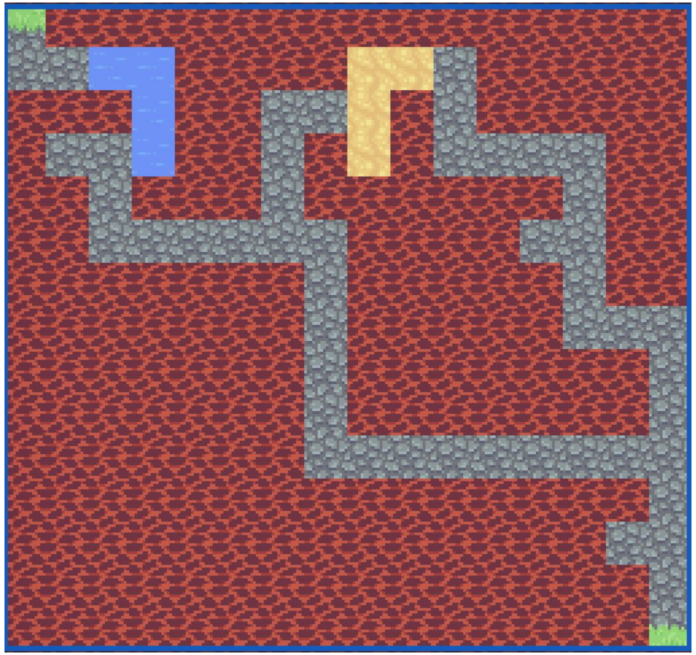
Create testbench stimulus with cover

Create testbench stimulus with cover

- If a design has deep state space
- or some difficult-to-reach edge cases
- Tell the tool the end goal
- It can figure out how to get there

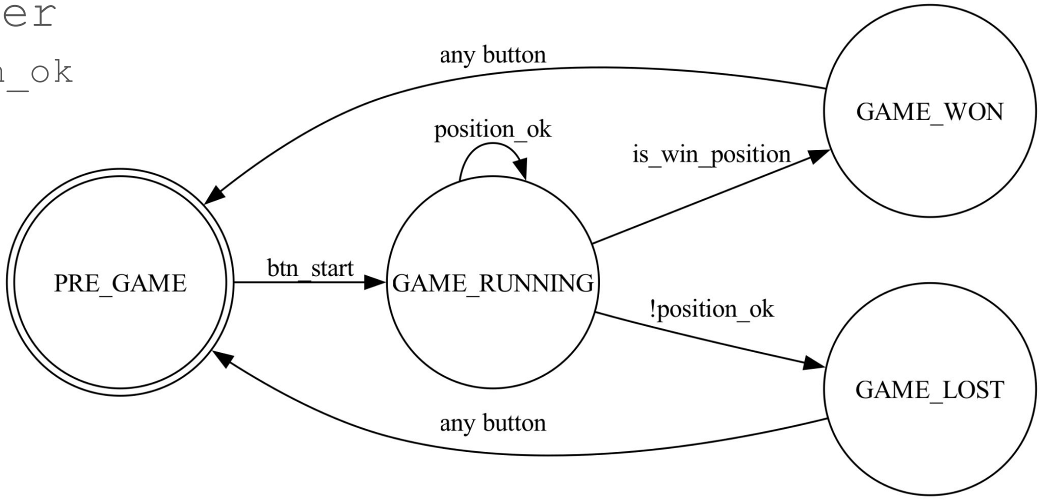
Example design: game logic

- Navigate a map without falling into lava, drowning in water, or choking in gas
- inputs `btn_left`, `btn_right`, `btn_up`, `btn_down` to move positions
- inputs `btn_A` (swim) and `btn_B` (hold breath) to survive water/gas tiles
- input `btn_start` to start the game



Example design: game logic

- DUT: FSM for game state
  - keeps track of menu/in-game/won/lost screen
  - also updates player position
  - has two submodules
- Submodule `position_checker`
  - combinatorially derives `position_ok` from next position + buttons A/B
- Submodule `debug_module`
  - allows changing position for debugging



Generating testbench stimuli

- Generate the inputs to navigate the whole map to win the game

```
won: cover property (@(posedge clk) game_state == GAME_WON);
```

- There are some known constraints

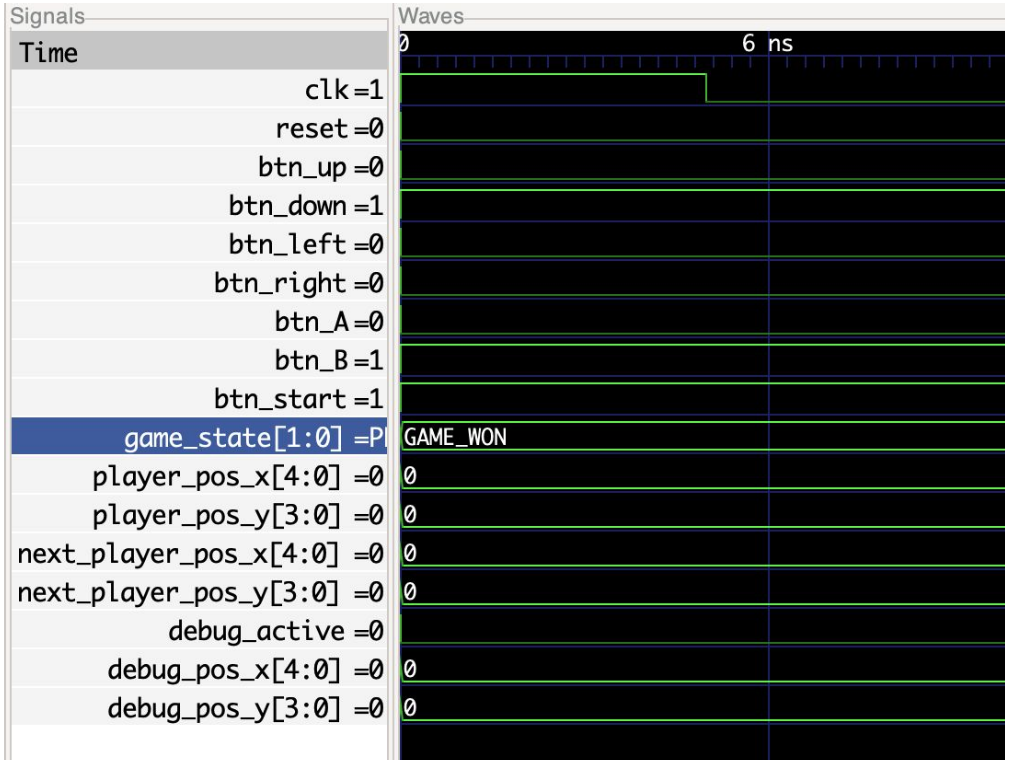
```
assume property (@(posedge clk)
(btn_up + btn_down + btn_right + btn_left < 2));
assume property (@(posedge clk) (btn_A + btn_B < 2));
```

- Let's run this:

```
[fsm] summary: engine_0 (smtbmc bitwuzla) returned pass
[fsm] summary: cover trace: fsm/engine_0/trace0.vcd
[fsm] summary:   reached cover statement game_fsm.won at
fsm.sv:123.10-123.65 in step 1
```

Generating testbench stimuli

- Most registers in the design are uninitialized
- The solver simply decides to start up the design in the `GAME_WON` state



```
clk =1
reset =0
btn_up =0
btn_down =1
btn_left =0
btn_right =0
btn_A =0
btn_B =1
btn_start =1
game_state[1:0] =P   GAME_WON
player_pos_x[4:0] =0   0
player_pos_y[3:0] =0   0
next_player_pos_x[4:0] =0   0
next_player_pos_y[3:0] =0   0
debug_active =0
debug_pos_x[4:0] =0   0
debug_pos_y[3:0] =0   0
```

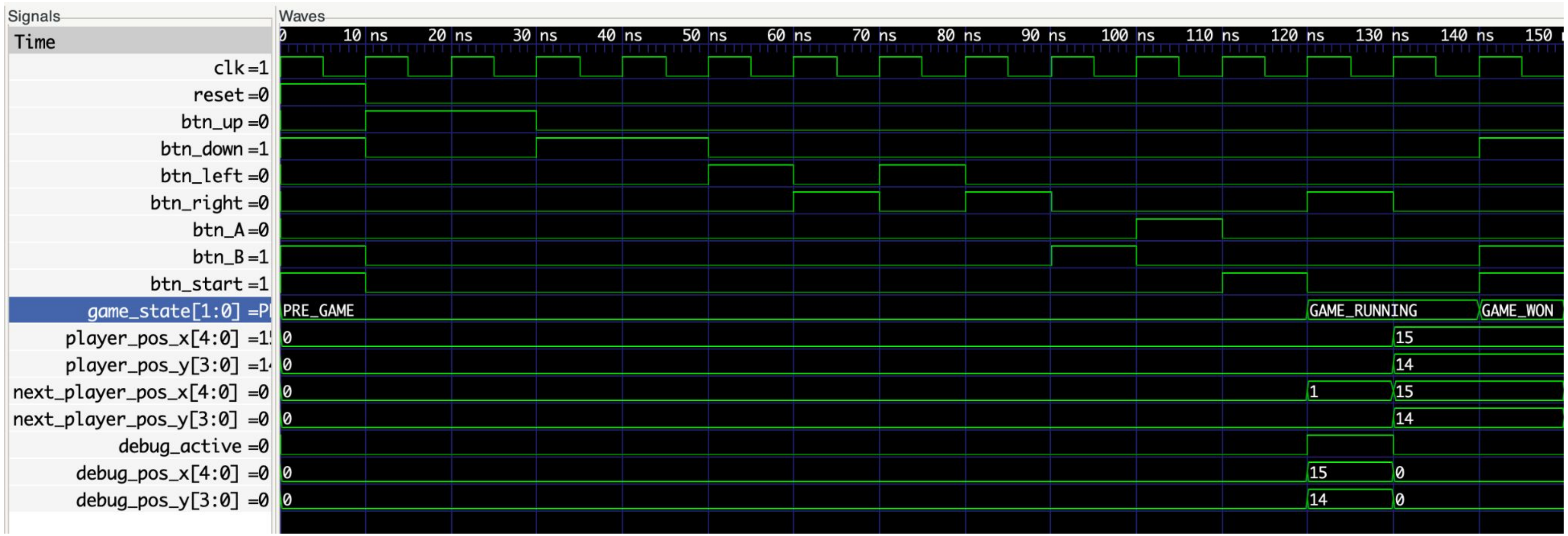
Generating testbench stimuli

- Let's add some assumptions

```
initial assume (reset);
```

- Note that we don't constrain reset to stay low past the first cycle, the solver can make use of repeat reset if it finds it useful
  - in this example it doesn't happen
- Let's run this again:

```
[fsm] summary: engine_0 (smtbmc bitwuzla) returned pass
[fsm] summary: cover trace: fsm/engine_0/trace0.vcd
[fsm] summary:   reached cover statement game_fsm.won at
fsm.sv:123.10-123.65 in step 15
```



- The solver finds the key combination to activate the debug unit
- The solver finds that it can be activated during pre-game screen, with the final `btn_start` both starting the game and activating the position update
- But this is still not actually what we were looking for…

Generating testbench stimuli

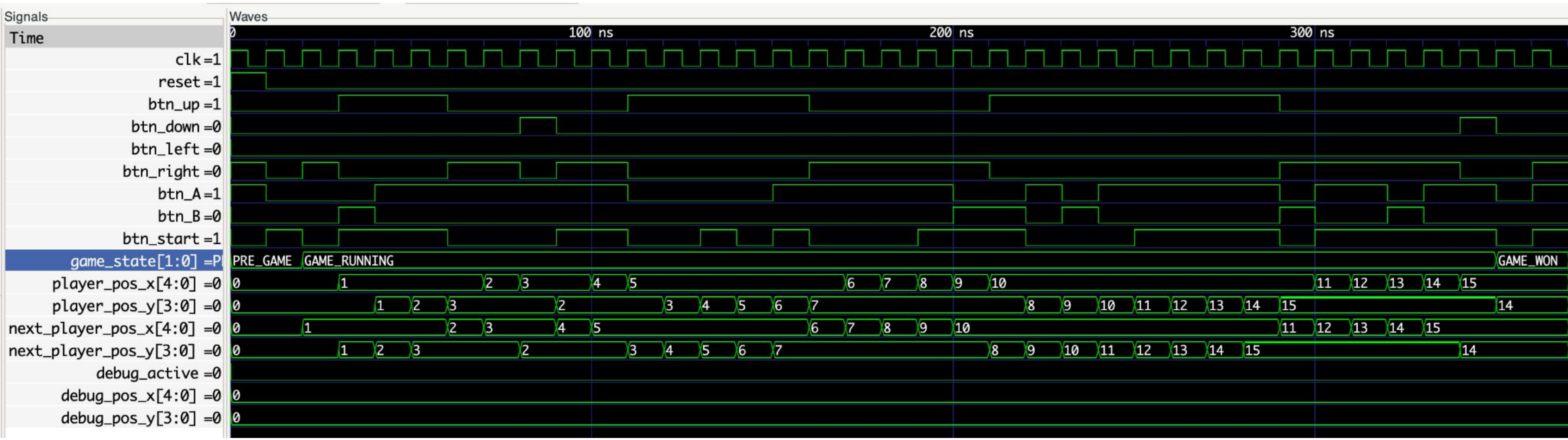
- This time, we want to exclude an intended functional state of the design

```
restrict property (@(posedge clk) !debug_active);
```

- Run again:

```
[fsm] summary: engine_0 (smtbmc bitwuzla) returned pass
[fsm] summary: cover trace: fsm/engine_0/trace0.vcd
[fsm] summary:   reached cover statement game_fsm.won at
fsm.sv:123.10-123.65 in step 36
```

Generating testbench stimuli



- This time we get the actual input sequence for reaching the winning position

Generating testbench stimuli

- SBY also generates a simulation testbench file
    - but the initialization section often needs adjustment

```
always @(posedge clock) begin
    // state 1
    if (cycle == 0) begin
    PI_btn_start <= 1'b1;
    PI_reset <= 1'b0;
    PI_btn_left <= 1'b0;
    PI_btn_B <= 1'b0;
    PI_btn_A <= 1'b0;
    PI_btn_right <= 1'b0;
    PI_btn_down <= 1'b0;
    PI_btn_up <= 1'b0;
    end
```

```
// state 2
if (cycle == 1) begin
PI_btn_start <= 1'b0;
PI_reset <= 1'b0;
PI_btn_left <= 1'b0;
PI_btn_B <= 1'b0;
PI_btn_A <= 1'b0;
PI_btn_right <= 1'b1;
PI_btn_down <= 1'b0;
PI_btn_up <= 1'b0;
end
```

...

Confirm Invariants

Confirm Invariants

- You think you know something that should be true about the design
- You want to rely on this to make your implementation simpler
- Double-check that it's actually true!
  - (This is where it's probably necessary to be a bit more rigorous)

Confirm Invariants

- Example: ML accelerator compute unit with 10-stage pipeline
- Three stages need to access memory, but the module only has one port
- Could write an arbiter to be safe, but it would be more efficient to just introduce a stall sometimes to ensure they never conflict
- Did we get it right?

```
memlock: assert property(@(posedge clock) disable iff(reset)
 (mem_rd0_en + mem_rd1_en + |mem_wr_en) < 2);
```

- no assumptions beyond initial reset

```
 [compute_memlock] summary: engine_0 (abc pdr) returned PASS
 [compute_memlock] summary: engine_0 did not produce any traces
```

Confirm your formal setup

- PASS! Hooray! But… are we sure?
- Common mistake in formal: accidentally overconstrain the design, it's not possible to for the solver to even reach any interesting state without violating assumptions
- Try to confirm the check is not vacuous

Confirm your formal setup

- One option: cover the "good" state of the thing the assertion is about

```
cover_memlock: cover property(@(posedge clock) disable
iff(reset) (mem_rd0_en + mem_rd1_en + |mem_wr_en) == 1);
```

```
[compute_memlock] summary: engine_0 (smtbmc yices) returned pass
[compute_memlock] summary: cover trace:
compute_memlock/engine_0/trace0.vcd
[compute_memlock] summary:   reached cover statement
marlann_compute.cover_memlock at compute.v:188.17-188.113 in
step 4
```

Confirm your formal setup

- Another option: let's try breaking the design…

```
assign s1_stall = ~~|(memlock_res & memlock_mask) ||~~
(maxlock_b && maxlock_a_q);
```

```
[compute_memlock] summary: engine_0 (abc pdr) returned FAIL
[compute_memlock] summary: counterexample trace:
compute_memlock/engine_0/trace.vcd
[compute_memlock] summary:   failed assertion
marlann_compute.memlock at compute.v:186.11-186.107 in step 8
```

- Can be fairly certain the formal property does what we want it to do

Validating subsystem interactions

Validating subsystem interactions

- If you've been adding assumes and asserts during the last two steps
- And so has your colleague on the other end of a shared interface
- Use each other's properties!
- Convert your colleagues assumptions into assertions and check that your design fulfils them
- Convert all assumptions on internally-generated signals into assertions and check them on the integrated design
- For an advanced example, we have a set of properties to verify AXI4 protocol compliance: https://yosyshq.readthedocs.io/projects/ap320/en/latest/

Bug hunting with assertions

Bug hunting with assertions

- Have observed some undesired behavior in testing on FPGA
- Can't reproduce in simulation
  - Don't know correct inputs?
  - Known set of inputs too long for simulation?
- Try directly to assert or cover the observed behavior
  - (essentially back to generating a tb)
  - This works if the problem is with an input edge case that isn't deep but just requires a very precisely timed sequence of inputs that almost never happens naturally
- Can't reach within a few cycles of initial state? Use sim trace as starting point
  - Especially if system has a bringup procedure
  - Can re-use saved end-of-sim state for many FV runs as long as RTL is unchanged
- Have partial information from ILA? Use SCY to trace in multiple hops
  - May fail if non-recorded state needs to have a specific value for end state to be reachable

Try it out!

Get the Tools

- Download nightly builds of the OSS CAD Suite
    - https://github.com/YosysHQ/oss-cad-suite-build/releases/latest
    - Includes Yosys, SBY, MCY, all dependencies, supported solvers, GHDL plugin (linux only)
    - Also nextpnr, Amaranth, cocotb, …
- Documentation: https://yosyshq.readthedocs.io/en/latest/
- Ask for an evaluation license to the commercial Tabby CAD Suite
    - Email contact@yosyshq.com or fill the contact form https://www.yosyshq.com/contact

Q&A

SBY – formal property checking with Yosys

- Frontend for formal flows
  - Allows easy use of SystemVerilog `assume(), assert(), cover()` statements
    - Complex SVA properties/sequences are supported with the commercial version
  - SBY has modes for bounded and unbounded proofs
    - Support for different unbounded proof methods (k-induction, pdr/ic3)
- Automates the steps for running formal proofs with Yosys
  - Yosys translation of design to formal problem formats (SMT2, BTOR2, Aiger…)
  - Running solvers to find a set of signal values responding to the problem (or not)
    - Allows using many solvers being developed by researchers
  - Using Yosys to translate the set of variable assignments back into a VCD trace
- Myriad of different input/output formats "under the hood"
  - SBY provides a uniform interface for a wide range of solvers, hiding those differences.
- Example projects:
  - riscv-formal: formally verify ISA compliance (rv32imc/rv64imc) https://github.com/YosysHQ/riscv-formal/
  - AXI4 formal verification IP (requires SVA support) https://github.com/YosysHQ-GmbH/SVA-AXI4-FVIP

What is Assertion-Based Formal Verification?

Available modes in SBY:

- Bounded Model Check (`mode bmc`):
  - Checks whether a state violating any assertion can be reached in N cycles from initial state
  - bound N = `depth` config option
- Cover (`mode cover`)
  - For each cover property, tries to find a trace of length N cycles or less from initial state to a state fulfilling the cover condition (and then checks that the found trace doesn't violate any assertions)
  - bound N = `depth` config option
- K-induction (`mode prove` + engine `smtbmc`):
  - Checks whether the set of assumptions + assertions is inductive
  - annoying for anything non-trivial
- IC3/PDR (`mode prove` + engine `abc pdr`)
  - Checks whether all assertions hold indefinitely
  - If you use this, set a timeout because it may be impossible to tell if it's making progress

Directly in DUT

```
module dut(input a, input b, output o);
    assign o = a ^ b;
    always_comb begin
            assume(b == 1'b0);
            assert(o == a);
    end
endmodule
```

Advantages:

- Least effort to set up
- Access to all internal signals
- Works even for modules deep in hierarchy with multiple instances
- Properties also get checked in sim

Disadvantages:

- Not easy to enable/disable
  - can use `` `ifdef FORMAL ``
- Could get lost within the design logic

In a testbench module:

```
module tb();
    wire a, b, o;
    dut dut_i(.*);

    always_comb begin
        assume(b == 1'b0);
        assert(o == a);
    end
endmodule
```

Advantages:

- Familiar format
- Neatly separated in its own file and module
- Works for mixed-language design

Disadvantages:

- Can't access internal signals

In a separate module injected with `bind`

```
module dut_check(input a, input b, input o);
    always_comb begin
            assume(b == 1'b0);
            assert(o == a);
    end
endmodule
bind dut dut_check dut_check_i(.*);
```

Advantages:

- Neatly separated in its own file
- Can access internal signals
- Works even for modules deep in hierarchy with multiple instances
- Works for mixed-language design

Disadvantages:

- Not supported in OSS version
  - Can still instantiate checker in DUT under `` `ifdef FORMAL `` guards
- Syntax can be a bit complex

Validating subsystem interactions

- On the game FSM there were 3 assumptions:

```
initial assume (reset);
```

- This one is presumably external (reset comes from outside the chip)

```
assume property (@(posedge clk)
(btn_up + btn_down + btn_right + btn_left < 2));

assume property (@(posedge clk) (btn_A + btn_B < 2));
```

- These two are about signals that come from another module (button handler)
- On button handler module, assert these properties!

Validating subsystem interactions

- Can of course just send those properties to your colleague via email
- But to track any changes in the interface, would be best to have single source
- Keep the properties in a separate file
- Switch between assume and assert as needed
- Taking up the game FSM example again, there was the interface with the module in charge of displaying graphics:

```
input logic show_pre_game_screen,
input logic show_won_game_screen,
input logic show_lost_game_screen,
input logic [MAP_IDX_SIZE_X-1:0] player_pos_x,
input logic [MAP_IDX_SIZE_Y-1:0] player_pos_y
```

Validating subsystem interactions

- Create a separate checker module for each interface:

```
property one_screen;
    @(posedge clk) (show_pre_game_screen +
show_won_game_screen + show_lost_game_screen < 2);
endproperty

generate if (ASSUME_MODE) begin
        assume property (one_screen);
end else begin
        p_one_screen: assert property (one_screen);
end
endgenerate
```

Validating subsystem interactions - unit test

- Graphics module would use it in assume mode
- In FSM check we use it in assert mode:

```
[file bind_fsm_graphics_interface_properties.sv]

bind game_fsm fsm_graphics_interface_properties
#(.ASSUME_MODE(0)) fsm_graphics_interface_properties_i(.*);
```

- Prove the above under condition of assuming the input behavior:

```
[file bind_fsm_btn_interface_properties.sv]

bind game_fsm fsm_btn_interface_properties
#(.ASSUME_MODE(1)) fsm_btn_interface_properties_i(.*);
```

Validating subsystem interactions - integration test

- Top module instantiating FSM and other modules it communicates with
- Use all properties in assert mode:

```
[file bind_fsm_graphics_interface_properties.sv]

bind integration_test_debounce_fsm
fsm_graphics_interface_properties #(.ASSUME_MODE(0))
fsm_graphics_interface_properties_i(.*);

[file bind_fsm_btn_interface_properties.sv]

bind integration_test_debounce_fsm
fsm_btn_interface_properties #(.ASSUME_MODE(0))
fsm_btn_interface_properties_i(.*);
```

Validating subsystem interactions

- If you keep the properties compatible with simulation they can be useful there too (for your colleagues that don't use formal yet)
- This is a simple example of assume-guarantee technique
-

Co-simulation for initial state

- Not-very-clever example: Load map data via write port added to game_fsm module
- Run `yosys sim_script.ys` for Yosys co-simulation from VCD first
  - Yosys `sim` command is not a fast simulator, this will take longer than it took to generate the VCD

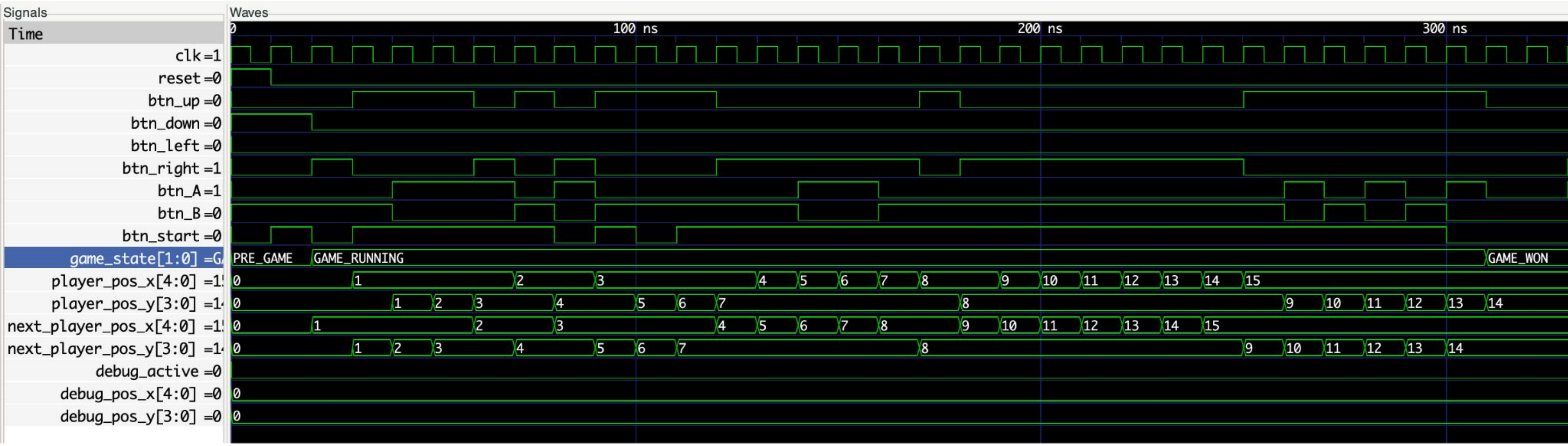
```
read -formal game_state.vh fsm.sv position_checker.sv
fsm_btn_interface_properties.sv bind_fsm_btn_interface_properties.sv
bind_global_assumes.sv bind_cover_win_check.sv
prep -top game_fsm
sim -r fsm_tb.vcd -w -scope fsm_tb.fsm_inst
write_rtlil fsm_post_sim.il
```

- Then instead of reading the design, just load the saved checkpoint in SBY `[script]` section:

```
read_rtlil fsm_post_sim.il
```

Co-simulation for initial state

- In the end, the trace looks just like before



SCY: Generate long traces with intermediate cover states

No time for an example…

Idea of SCY:

- Multiple sets of cover properties to be reached in sequence
- Use VCD from first cover task as starting point for second task
- Stitched together into a longer trace than would be tractable for a single cover task