GoogleBard assisted Formal Verification of **Pattern Generator** design

Group 15 - Fourmal Methods four System Verification

Harshit Seksaria - 210101047
Siddhant Jethwani - 210101053
Vatsal Jain - 210101110
Badekai Vijesh Ramachandra Bhat - 210101121

1) Project Report:

This project aims to evaluate the efficiency of using a large language model (LLM) to generate and improve SystemVerilog Assertions (SVA) for formal verification of the Pattgen design.

Specifically, the project will investigate the following:

- How efficiently can an LLM model generate SVA assertions that accurately and comprehensively specify the correctness properties of the Pattgen design?
- How efficiently can an LLM model improve the quality of SVA assertions, given a prompt?

The results of this project will help to determine the feasibility of using LLMs to improve the efficiency and effectiveness of formal verification in practice.

2) Design Description:

Description

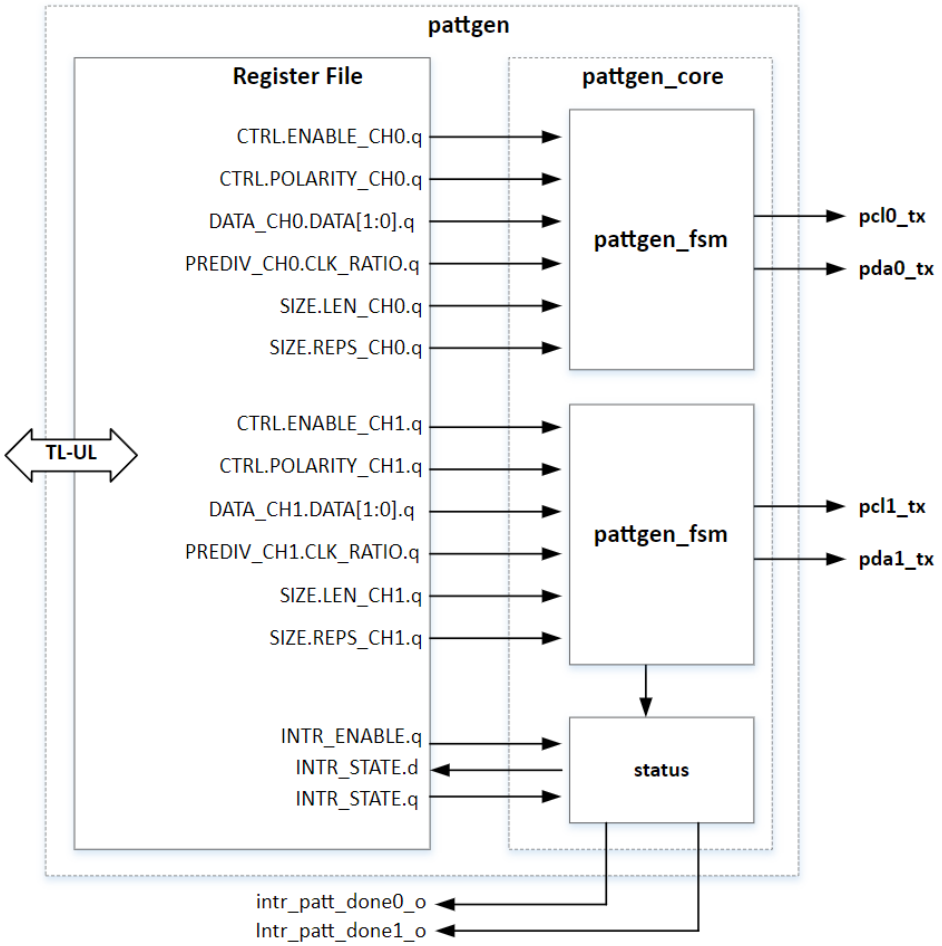
The pattern generator HWIP transmits short (maximum 64 bits) time-dependent data patterns on two clock-parallel channels. Each channel consists of one clock (pcl) and one data (pda) line. The output channels may be activated and operated independently, or they can be started at the same time to effectively create a 4-output pattern.

Features

- Generates time-dependent patterns on two (2) channels, each with its own clock.
    - In each channel, data is transmitted serially on a one-bit data (pda) output, which is synchronous to a corresponding parallel clock signal (pcl).
    - The channels can operate independently or synchronously with each other.
- Each output channel supports the following configuration settings:
    - Pattern data per output (up to 64 bits of data).

    - 32-bit pre-divider to derive pattern clock from I/O clock (minimum ratio: 2).
    - Each pattern can be repeated up to 1024 times.
    - The polarity of the clock signal is programmable.
- The block sends an interrupt on pattern completion.

Block Diagram



Inputs:

– clk_io: System clock input.
– reset: Reset signal for initialization.
– enable: Control signal to activate the pattern generation.
– clk_pre_divide: Input for configuring the clock pre-divider. – pattern: Data pattern to be transmitted.
– pattern size: Size of the data pattern.
– polarity: Signal polarity configuration.
– n repeats: Number of repetitions for the pattern.

Outputs:

– pda: Data output signal. – pcl: Clock output signal.

3) Properties:

The following are the properties generated by the group members by carefully studying the design. The completeness of the properties have been manually checked by ensuring that all of the **INPUTS-OUTPUTS** are covered.

1. G((enable -> state = ACTIVE | state = END) & (!enable -> state = IDLE))

2. G((!polarity -> pcl = clock_int) & (polarity -> pcl = !clock_int))

3. G((bit_ctr <patt.size -> X(bit_ctr) = bit_ctr+1) & (bit_ctr = patt.size -> X(bit_ctr=0)))

4. G(bit_ctr = patt.size -> X(rep_ctr) = rep_ctr+1)

5. G(rep_ctr = n_repeats ->X (state = END))

6. G(reset -> X(rep_ctr = 0 & state = IDLE & bit_ctr = 0 & clk_div = 0))

7. G(enable=1 -> ((clk_div<clk_prediv -> X(clk_div)=clk_div+1) & ((clk_div=clk_prediv -> (X(clk_div=0) & X(clk_int) = !clk_int))))

8. G((FSM state=ACTIVE) → pda=pattern[bit_ctr])

9. G(state = ACTIVE → F(interrupt_done))

4) Assertions obtained from the design specification:

a. Give details of the prompt used

Initial prompt -

H You are an expert verification engineer. I need you to write a set of complete and consistent LTL properties for a model of a "pattern generator" with descriptions as follows :

Theory of Operation: The pattern can be started (or halted) on either channel by setting the corresponding CTRL.ENABLE bit to 1 (or 0) for the desired channel. Once disabled, either channel can be configured independently. The channel parameters (i.e. clock divider ratio, clock polarity, pattern length, pattern data, and repetition count) can all be programmed on a per-channel basis. Enabling the pattern generator channel starts the pattern from the beginning.

Response -

General Properties:

- G (enable = 0 -> next state = IDLE)
  - This property states that if the enable bit is low, then the next state of the FSM must be IDLE.
- G (reset = 1 -> next state = IDLE)
  - This property states that if the reset signal is asserted, then the next state of the FSM must be IDLE.

State Transitions:

- G (state = IDLE -> enable = 1 -> next state = ACTIVE)
  - This property states that if the state of the FSM is IDLE and the enable bit is set to 1, then the next state of the FSM must be ACTIVE.
- G (state = ACTIVE -> bit_ctr = pattern_size -> next state = END)
  - This property states that if the state of the FSM is ACTIVE and the bit counter reaches the pattern size, then the next state of the FSM must be END.
- G (state = END -> repeat_ctr = n_repeats -> next state = IDLE)
  - This property states that if the state of the FSM is END and the repeat counter reaches the number of repeats, then the next state of the FSM must be IDLE.

Output Generation:

- G (state = ACTIVE -> pda = pattern[bit_ctr])
  - This property states that if the state of the FSM is ACTIVE, then the PDA output must be equal to the bit value at the position specified by the bit counter in the pattern.

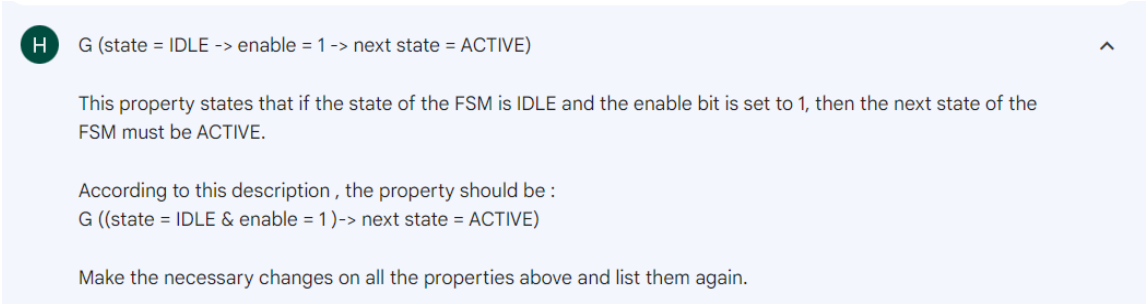
Interrupt Generation:

- G (state = END -> done = 1)
  - This property states that if the state of the FSM is END, then the done interrupt flag must be set to 1.

7 initial properties were obtained. The LLM very well described all the properties correctly in English except a few (which we will try to correct in later prompts) but almost all of the LTL properties were logically inconsistent with the English properties. The logical mistake was of a similar nature in every LTL property (implication was used in place of conjunction), So we tried to give a prompt to correct one of the LTL, and asked it to apply the same changes to the rest.

b. Converging on the good assertions.

Prompt -



Response -

General Properties:

- G ((state = IDLE) & (enable = 1) -> next state = ACTIVE)
  - This property states that if the state of the FSM is IDLE and the enable bit is set to 1, then the next state of the FSM must be ACTIVE.
- G (reset = 1 -> next state = IDLE)
  - This property states that if the reset signal is asserted, then the next state of the FSM must be IDLE.

State Transitions:

- G ((state = ACTIVE) & (bit_ctr = pattern_size) -> next state = END)
  - This property states that if the state of the FSM is ACTIVE and the bit counter reaches the pattern size, then the next state of the FSM must be END.
- G ((state = END) & (repeat_ctr = n_repeats) -> next state = IDLE)
  - This property states that if the state of the FSM is END and the repeat counter reaches the number of repeats, then the next state of the FSM must be IDLE.
- G ((state = IDLE) & (enable = 0) -> next state = IDLE)
  - This property states that if the state of the FSM is IDLE and the enable bit is set to 0, then the next state of the FSM must remain IDLE.

Output Generation:

- G (state = ACTIVE -> pda = pattern[bit_ctr])
  - This property states that if the state of the FSM is ACTIVE, then the PDA output must be equal to the bit value at the position specified by the bit counter in the pattern.

Interrupt Generation:

- G (state = END -> done = 1)
  - This property states that if the state of the FSM is END, then the done interrupt flag must be set to 1.

LLM understood the prompt correctly and made the necessary changes.

The first two state transition properties were incorrect, as the LLM had a wrong idea about the END state. So we tried to correct it with a prompt, clarifying the significance of the END state.
Prompt -

> H The state END means that pattern generation is complete including all the repetitions, not just one.
> Correct the properties listed above and list them again

Response -

General Properties:

- G ((state = IDLE) & (enable = 1) -> next state = ACTIVE)
  - This property states that if the state of the FSM is IDLE and the enable bit is set to 1, then the next state of the FSM must be ACTIVE.
- G (reset = 1 -> next state = IDLE)
  - This property states that if the reset signal is asserted, then the next state of the FSM must be IDLE.

State Transitions:

- G ((state = IDLE) & (enable = 0) -> next state = IDLE)
  - This property states that if the state of the FSM is IDLE and the enable bit is set to 0, then the next state of the FSM must remain IDLE.
- G (state = ACTIVE -> bit_ctr = pattern_size -> next state = END)
  - This property states that if the state of the FSM is ACTIVE and the bit counter reaches the pattern size, then the next state of the FSM must be END.
- G (state = END -> repeat_ctr = n_repeats -> next state = IDLE)
  - This property states that if the state of the FSM is END and the repeat counter reaches the number of repetitions, then the next state of the FSM must be IDLE.

Output Generation:

- G (state = ACTIVE -> pda = pattern[bit_ctr])
  - This property states that if the state of the FSM is ACTIVE, then the PDA output must be equal to the bit value at the position specified by the bit counter in the pattern.

Interrupt Generation:

- G (state = END -> done = 1)
  - This property states that if the state of the FSM is END, indicating the completion of all repetitions, then the done interrupt flag must be set to 1.

LLM could not understand the significance of the prompt, and didn't make the necessary changes. So instead, we tried to give a property-specific prompt, and told it exactly what it should be.

Prompt -

H

G (state = ACTIVE -> bit_ctr = pattern_size -> next state = END)
state=ACTIVE and bit_ctr=pattern_size should hold simultaneously so put an & between them.

Also this property is incorrect as this condition marks the end of one repetition but not the end of pattern generation so state should not be changed to END here

Response -

General Properties:

- G ((state = IDLE) & (enable = 1) -> next state = ACTIVE)
  - This property states that if the state of the FSM is IDLE and the enable bit is set to 1, then the next state of the FSM must be ACTIVE.
- G (reset = 1 -> next state = IDLE)
  - This property states that if the reset signal is asserted, then the next state of the FSM must be IDLE.

State Transitions:

- G ((state = IDLE) & (enable = 0) -> next state = IDLE)
  - This property states that if the state of the FSM is IDLE and the enable bit is set to 0, then the next state of the FSM must remain IDLE.
- G ((state = ACTIVE) & (bit_ctr < pattern_size) -> next state = ACTIVE)
  - This property states that if the state of the FSM is ACTIVE and the bit counter is less than the pattern size, then the next state of the FSM must remain ACTIVE.
- G ((state = ACTIVE) & (bit_ctr = pattern_size) & (repeat_ctr < n_repeats) -> next state = ACTIVE)
  - This property states that if the state of the FSM is ACTIVE, the bit counter has reached the pattern size, and the repeat counter is less than the number of repetitions, then the next state should remain ACTIVE.
- G ((state = ACTIVE) & (bit_ctr = pattern_size) & (repeat_ctr = n_repeats) -> next state = END)
  - This property states that if the state of the FSM is ACTIVE, the bit counter has reached the pattern size, and the repeat counter has reached the number of repetitions, then the next state must be END.
- G (state = END -> next state = IDLE)
  - This property states that if the state of the FSM is END, then the next state must be IDLE.

Output Generation:

- G (state = ACTIVE -> pda = pattern[bit_ctr])
  - This property states that if the state of the FSM is ACTIVE, then the PDA output must be equal to the bit value at the position specified by the bit counter in the pattern.

Interrupt Generation:

- G (state = END -> done = 1)
  - This property states that if the state of the FSM is END, indicating the completion of all repetitions, then the done interrupt flag must be set to 1.

This time, LLM correctly understood the prompt. Even made necessary changes in the rest of the properties.

c. Assertions that LLM could not generate.

d. Does LLM provide new important assertions that was not there in Point 3?

Yes, as shown in the above image. There is one new assertion that was not there in Point 3. This behavior, however, is not mentioned in the design specification explicitly, but, is indeed true as checked in EDA playground simulation. This shows that LLM is capable of speculating obvious properties not explicitly mentioned in the design specification.

e. Are the assertions generated by LLM are consistent?

Assertions generated by LLM could be logically incorrect, but there is no inconsistency. We would conclude that this may be because our model is so designed that there is very little chance of inconsistency in the properties as most of the properties are independent of each other (and the outputs of any clock cycle are not the inputs in the future clock cycles).

f. Completeness of the assertions generated by LLM?

Assertions generated by LLM are not complete, as shown in the image in 4c, there were quite a few properties that were not generated by LLM just using the design specifications.

g. Verification by EDA Playground of the LLM generated assertions.

Now, we give prompt to get SVA from the generated LTL properties.

Prompt -

H Generate the system verilog assertions(sva) for the above properties

Response (Among many others) -

Code snippet

```
always @(posedge clk) begin
  if (reset) begin
    state <= IDLE;
  end else begin
    case (state)
      IDLE: begin
        if (enable) begin
          state <= ACTIVE;
        end
      end
      ACTIVE: begin
        if (repeat_ctr == n_repeats) begin
          state <= END;
        end
      end
      END: begin
        state <= IDLE;
      end
    endcase
  end
end
```

Use code with caution. Learn more

This is a design rather than SVA.

Prompt -

H The general properties that you have mentioned are not assertions rather it is the design , I need you to generate assertions to check the properties that you mentioned above

Response-

Code snippet

```
always @(posedge clk) begin
  assert property (@(posedge clk) (state == IDLE) & (enable == 1) |=> next state == ACTIVE);
end
```

Use code with caution. Learn more

This assertion ensures that the FSM transitions from IDLE to ACTIVE only when the enable bit is asserted.

Code snippet

```
always @(posedge clk) begin
  assert property (@(posedge clk) (state == ACTIVE) & (bit_ctr < pattern_size) |=> next state == AC
end
```

Use code with caution. Learn more

This assertion verifies that the FSM remains in the ACTIVE state as long as the bit counter is less than the pattern size.

Code snippet

```
always @(posedge clk) begin
  assert property (@(posedge clk) (state == ACTIVE) & (bit_ctr == pattern_size) & (repeat_ctr < n_r
end
```

Use code with caution. Learn more

This assertion ensures that the FSM remains in the ACTIVE state after completing one pattern repetition but before completing all repetitions.

Code snippet

```
always @(posedge clk) begin
  assert property (@(posedge clk) (state == ACTIVE) & (bit_ctr == pattern_size) & (repeat_ctr == n_
end
```

Use code with caution. Learn more

This assertion verifies that the FSM transitions from ACTIVE to END upon completing all specified repetitions of the pattern.

Code snippet

```
always @(posedge clk) begin
  assert property (@(posedge clk) (state == END) |=> next state == IDLE);
end
```

Use code with caution. Learn more

This assertion ensures that the FSM transitions back to IDLE after completing the pattern generation process.

Code snippet

```
always @(posedge clk) begin
  assert property (@(posedge clk) (state == ACTIVE) |=> pda == pattern[bit_ctr]);
end
```

Use code with caution. Learn more

This assertion verifies that the PDA output matches the corresponding bit in the pattern while the FSM is in the ACTIVE state.

Code snippet

```
always @(posedge clk) begin
  assert property (@(posedge clk) (state == END) |=> done == 1);
end
```

Use code with caution. Learn more

After replacing the dummy variable names generated by LLM with actual variable names, we ran the simulation on EDA playground by writing a testbench and importing necessary design Verilog files from open titan github into EDA playground.

Please find the EDA playground link here: https://edaplayground.com/x/mqna

Bug found in the design

- G ((state = ACTIVE) & (bit_ctr = pattern_size) & (repeat_ctr = n_repeats) -> next state = END)
    - This property states that if the state of the FSM is ACTIVE, the bit counter has reached the pattern size, and the repeat counter has reached the number of repetitions, then the next state must be END.

The above assertion fails in the simulation of the design. Upon inspection, it has come to our knowledge that the design is producing a redundant zero after every repetition of the pattern. This behavior is not mentioned in the design specifications at all.

5) Systematic Module-wise generations of SVAs by Google Bard

a. Initial System Prompt - (pattgen_chan.sv module)

V Here is the system verilog code for the pattgen_chan.sv module. Generate five LTL properties for this verilog code -
// Copyright lowRISC contributors.
// Licensed under the Apache License, Version 2.0, see LICENSE for details.
// SPDX-License-Identifier: Apache-2.0...

Response :-

Property	Description	LTL Formula
Safety 1	pcl_o and pda_o must always have different values	G (clk_i -> X (pcl_o != pda_o))
Safety 2	clk_cnt_q must always be equal to prediv_q whenever pcl_o is asserted	G (pcl_o -> X (clk_cnt_q == prediv_q))
Liveness 1	complete_q must eventually become asserted	F (complete_q)
Liveness 2	event_done_o must eventually become asserted	F (event_done_o)
Fairness	If enable is asserted, then bit_cnt_q must eventually become equal to len_q	F (enable -> X (bit_cnt_q == len_q))

Export to Sheets

Now, we try to generate System Verilog Assertions for each of the mentioned properties :-

1. **Prompt 1:-**

V Generate the system verilog assertion for this property :- "pcl_o and pda_o must always have different values"

Response :-

```
always @(posedge clk_i or negedge rst_ni) begin
  if (!rst_ni) begin
    assert (pcl_o != pda_o);
  end
end
```

2. **Prompt 2 :-**

V Generate the system verilog assertion for this property :- "clk_cnt_q must always be equal to prediv_q whenever pcl_o is asserted"

Response :-

```
always @(posedge clk_i or negedge rst_ni) begin
  if (!rst_ni) begin
    assert (pcl_o implies (clk_cnt_q == prediv_q));
  end
end
```

3. **Prompt 3 :-**

V Generate the system verilog assertion for this property :- "complete_q must eventually become asserted"

Response :-

Code snippet

```
property complete_q_eventually_asserted;
  @(posedge clk_i)
  disable iff(rst_ni)
  assert property(##1 complete_q);
endproperty

always_ff @(posedge clk_i or negedge rst_ni) begin
  if (!rst_ni) begin
    assert property(complete_q_eventually_asserted);
  end
end
```

Use code with caution. Learn more

4. **Prompt 4 :-**

V generate the system verilog assertion for this property - "event_done_o must eventually become asserted"

Response :-

Code snippet

```
property event_done_o_eventually_asserted;
  @(posedge clk_i)
  disable iff(rst_ni)
  assert property(##1 event_done_o);
endproperty

always_ff @(posedge clk_i or negedge rst_ni) begin
  if (!rst_ni) begin
    assert property(event_done_o_eventually_asserted);
  end
end
```

Pattgen_core Module :

Initial System Prompt

You are an expert in system design verification. I need you to write some LTL properties for a "pattgen" or "Pattern generator".

Description : The pattern generator HWIP transmits short (maximum 64 bits) time-dependent data patterns on two clock-parallel channels. Each channel consists of one clock (pcl) and one data (pda) line. The output channels may be activated and operated independently, or they can be started at the same time to effectively create a 4-output pattern.

Design Details : The design consists of two identical and independent finite state machines, each an instance of module pattgen_fsm. Each FSM is essentially three nested counters, with one counter to control the clock division, another to count out the sequence bits, and a third to keep count of the number of repetitions.
Each FSM consists of Inputs:
clk_io, reset, enable, clk_pre_divide, pattern, pattern_size, polarity, and n_repeats
Outputs: pda and pcl, a single state variable with three states(IDLE, ACTIVE, and END), a clock-divide counter, clk_div, a single clock-divide flop, clk_int, and two additional internal counters bit_ctr and repeat_ctr.
Each FSM is disabled when enable is low. Disabling the FSM is equivalent to an FSM reset, and both operations place the FSM in the IDLE state. While in IDLE, the other state machine registers assume their default states: The internal counters, the clock-

Prompt 1

```
// Copyright lowRISC contributors.
// Licensed under the Apache License, Version 2.0, see LICENSE for details.
// SPDX-License-Identifier: Apache-2.0
//
// Description: Pattgen Core Module

module pattgen_core
 import pattgen_reg_pkg::*;
 import pattgen_ctrl_pkg::*;
(
 input        clk_i,
 input        rst_ni,
 input pattgen_reg2hw_t reg2hw,
 output pattgen_hw2reg_t hw2reg,
```

Response

Property Description	LTL Formula
The pda0_tx_o output should be zero when the FSM for channel 0 is in the IDLE state.	G(fsm0_state == IDLE -> pda0_tx_o == 0)
The pda0_tx_o output should be driven by the pattern input for channel 0 when the FSM for channel 0 is in the ACTIVE state.	G(fsm0_state == ACTIVE -> pda0_tx_o == pattern0[bit_ctr0])
The FSM for channel 0 should transition to the END state when the repeat counter for channel 0 overflows.	F(repeat_ctr0 == 0 -> fsm0_state == END)
The interrupt signal for channel 0 should be asserted when the FSM for channel 0 transitions to the END state.	F(fsm0_state == END -> intr_done_ch0_o == 1)
The pda1_tx_o output should be zero when the FSM for channel 1 is in the IDLE state.	G(fsm1_state == IDLE -> pda1_tx_o == 0)

Prompt 2

Generate System verilog assertion for the following property - "The pda0_tx_o output should be zero when the FSM for channel 0 is in the IDLE state."

Response

Code snippet

```
assert property (@(posedge clk_i) !((fsm0_state == IDLE) && (pda0_tx_o != 0)));
```


Prompt 3

Generate System verilog assertion for the following property - "The pda0_tx_o output should be driven by the pattern input for channel 0 when the FSM for channel 0 is in the ACTIVE state."

Response

Code snippet

```
assert property (@(posedge clk_i) !((fsm0_state == ACTIVE) && (pda0_tx_o != pattern0[bit_ctr0])));
```


Prompt 4

Generate System verilog assertion for the following property - "The FSM for channel 0 should transition to the END state when the repeat counter for channel 0 overflows."

Response

Code snippet

```
assert property (@(posedge clk_i) ((repeat_ctr0 == 0) -> (fsm0_state == END)));
```

Use code with caution. Learn more

Prompt 5

Generate System verilog assertion for the following property - "The interrupt signal for channel 0 should be asserted when the FSM for channel 0 transitions to the END state."

Response

Code snippet

```
assert property (@(posedge clk_i) ((fsm0_state == END) -> (intr_done_ch0_o == 1)));
```

Use code with caution. Learn more

Prompt 6

Generate System verilog assertion for the following property - "The pda1_tx_o output should be zero when the FSM for channel 1 is in the IDLE state."

Response

Code snippet

```
assert property (@(posedge clk_i) !((fsm1_state == IDLE) && (pda1_tx_o != 0)));
```

Use code with caution. Learn more

b. How to converge on the good assertions?

Similar to the case as in the design specifications, the SVA's generated had plenty of mistakes but the mistakes could be corrected to great lengths by individually pointing out the incorrect ones and

prompting the LLM in the right direction which have been shown in the previous point. We observed that multiple SVA's had to be combined to generate the form which we had concluded manually.

c. Your overall recommendation generating verification assertion from Verilog? Please write the strategy used.

When we give the Verilog code directly to the LLM, most of the assertions generated are not consistent with their generated English and LTL properties . This inconsistency generally arises when we try to generate all English, LTL and SVAs together.

Strategy that can be followed to generate better assertions is as follows: Generate the properties in English and their respective LTL formulae, and then for each property mentioned in English, ask the LLM to generate the SVA. Deal with the further inconsistencies by giving more correctness oriented prompts for each incorrect SVA generated.

6) **Discussion:**

a. Conclusions from the design specifications:

In our endeavor of Formal property verification of Pattern Generator with **GoogleBard** using design specifications , we conducted a comparative analysis of the effectiveness across three languages: English, Linear Temporal Logic (LTL), and SystemVerilog Assertions (SVA). Our focus was on understanding the quality and coherence of the generated assertions in each language.

1) English Properties: We observed that the generated assertions in **English surpassed those in LTL and SVA** in terms of clarity and precision. The English responses demonstrated an adept ability to articulate crucial points from the design specification, providing logically correct assertions. Hence, the linguistic capabilities of GoogleBard was made evident.
2) SVA: The initial SystemVerilog Assertions (SVA) generated were pretty wayward and weak in covering critical aspects of the design. Upon multiple prompts, the SVAs could be improved to great lengths in covering all aspects of the design.
3) LTL properties: The LTL properties too were not up to the mark as the LLM usually confused the operators in the LTL language (For example, confused the Global(G) and Future(F) operators) but could correct them later upon multiple prompts. They clearly did not outshine the English and SVA counterparts.

b. Conclusions from the Verilog code:
The conclusions from using the Verilog code follow the same trend as the design specifications.

c/d. Overall recommendation of LLM for model checking:
The LLM's proficiency in generating correct assertions in the English language highlights its ability to grasp syntactic and semantic structures.

The SVA generation, while maintaining correctness in its assertions, encounters a critical stumbling block. The discrepancy between the specified input/output signals and the actual signals used in the system introduces a significant risk. This can be definitely corrected by manually comprehending the design and prompting the LLM accordingly.

The challenges become more apparent when transitioning to formal languages like LTL. The generation of LTL properties appears to face substantial issues, with a prevalence of inaccuracies when using the operators. This suggests a difficulty in translating natural language understanding into precise formal logic specifications, where subtle nuances play a critical role.