From Verilog to Proof:

Information Extraction to Proof Generation
Consider a circuit in Verilog:

```verilog
module controller1 (clk, reset, a1_in, d1_in, we1_in,
                     a2_in, d2_in, we2_in, q1, cf, a1, d1, we1, a2, d2, we2, c);
input clk, reset;
input [4:0] a1_in, a2_in;
input [31:0] d1_in, d2_in, q1;
input we1_in, we2_in, cf;
output [4:0] a1, a2;
output [31:0] d1, d2;
output we1, we2;
output reg c;
reg [4:0] cur_read, cur_write;
reg [31:0] stored_value;
reg cprev;

assign a1 = (c) ? cur_read : a1_in;
assign d1 = d1_in;
assign we1 = (c) ? 1'b0 : we1_in;

assign a2 = (c) ? cur_write : a2_in;
assign d2 = (c) ? stored_value : d2_in;
assign we2 = (cprev & c) ? 1'b1 : (c ? 1'b0 : we2_in);

always @ (posedge clk) begin
  if (~reset) begin
    cur_write <= 5'b00000;
    cur_read <= 5'b00000;
    c <= 1'b0;
    cprev <= 1'b0;
    stored_value <= 32'd0;
  end
  else begin
    if (cur_write == 5'b11111 & ~cf) begin
      cur_write <= 5'b00000;
      cur_read <= 5'b00000;
      c <= 1'b0;
      cprev <= 1'b0;
      stored_value <= 32'd0;
    end
    else begin
      cur_read <= (cf & ~c) ? 5'b00000 : cur_read +
                     5'b00001;
      cur_write <= (cf & ~c) ? 5'b00000 : cur_write;
      stored_value <= q1;
      c <= (cf | c);
      cprev <= c;
    end
  end
endmodule
```
always @ (posedge clk) begin
  if (~reset) begin
    cur_write <= 5'b00000;
    cur_read <= 5'b00000;
    c <= 1'b0;
    cprev <= 1'b0;
    stored_value <= 32'd0;
  end
  else begin
    if (cur_write == 5'b11111 & ~cf) begin
      cur_write <= 5'b00000;
      cur_read <= 5'b00000;
      c <= 1'b0;
      cprev <= 1'b0;
      stored_value <= 32'd0;
    end
    else begin
      cur_read <= (cf & ~c) ? 5'b00000 : cur_read + 5'b00001;
      cur_write <= (cf & ~c) ? 5'b00000 : cur_read;
      stored_value <= q1;
      c <= (cf | c);
      cprev <= c;
    end
  end
end
If Statements Operate on Conditional Expressions:

if (cur_write == 5'b11111 & ~cf)

From these we will extract FSM "states"...
Extraction of Conditional Expressions

always @ (posedge clk) begin
    if (~reset) begin
        cur_write <= 5'b00000;
        cur_read <= 5'b00000;
        c <= 1'b0;
        cprev <= 1'b0;
        stored_value <= 32'd0;
    end
    else begin
        if (cur_write == 5'b11111 & ~cf) begin
            cur_write <= 5'b00000;
            cur_read <= 5'b00000;
            c <= 1'b0;
            cprev <= 1'b0;
            stored_value <= 32'd0;
        end
        else begin
            cur_read <= (cf & ~c) ? 5'b00000 : cur_read + 5'b00001;
            cur_write <= (cf & ~c) ? 5'b00000 : cur_read;
            stored_value <= q1;
            c <= (cf | c);
            cprev <= c;
        end
    end
end
List Extracted Conditional Expressions:

E1: \sim reset
E1': \sim (\sim reset) = reset

E2: cur_write == 5'b11111 & \sim cf
E2': \sim (cur_write == 5'b11111 & \sim cf)
always @ (posedge clk) begin
  if (~reset) begin
    cur_write <= 5'b00000;
    cur_read <= 5'b00000;
    c <= 1'b0;
    cprev <= 1'b0;
    stored_value <= 32'd0;
  end
  else begin
    if (cur_write == 5'b11111 & ~cf) begin
      cur_write <= 5'b00000;
      cur_read <= 5'b00000;
      c <= 1'b0;
      cprev <= 1'b0;
      stored_value <= 32'd0;
    end
    else begin
      cur_read <= (cf & ~c) ? 5'b00000 : cur_read +
                       5'b00001;
      cur_write <= (cf & ~c) ? 5'b00000 : cur_read;
      stored_value <= q1;
      c <= (cf | c);
      cprev <= c;
    end
  end
end

<-- (E1)

<-- (E1', E2)

<-- (E1', E2')
These are states in an FSM

But how do we determine the transitions?
always @(posedge clk) begin
  if (~reset) begin
    cur_write <= 5'b00000;
    cur_read <= 5'b00000;
    c <= 1'b0;
    cprev <= 1'b0;
    stored_value <= 32'd0;
  end
  else begin
    if (cur_write == 5'b11111 & ~cf) begin
      cur_write <= 5'b00000;
      cur_read <= 5'b00000;
      c <= 1'b0;
      cprev <= 1'b0;
      stored_value <= 32'd0;
    end
    else begin
      cur_read <= (cf & ~c) ? 5'b00000 : cur_read +
                                 5'b00001;
      cur_write <= (cf & ~c) ? 5'b00000 : cur_read;
      stored_value <= q1;
      c <= (cf | c);
      cprev <= c;
    end
  end
end

Blue blocks have a payload of non-blocking assign statements
Traverse path from root to leaf

Collect payloads along path, add to combinatorial updates