SymbiYosys coursework exercises

John Wickerson

Autumn term 2024

This year's SymbiYosys coursework concerns the verification of an 8-bit multiplier.

Marking principles. If you have completed a task in full, you will get full marks for it and it is not necessary to show your working. If you have not managed to complete a task, partial credit may be given if you can demonstrate your thought process.

Submission process. You are expected to produce a single zip file called `Surname1Surname2.zip`, where `Surname1` and `Surname2` are the surnames of the two students in the pair. This file should contain your solutions to all of the tasks below that you have attempted. You may include `.sv` files and `.sby` files in your zip file. You are welcome to show your working on incomplete tasks by decorating your file with `/*comments*/` or `//comments`.

Plagiarism policy. You **are** allowed to consult internet sources like SymbiYosys and SystemVerilog tutorials. You **are** allowed to work together with the other student in your pair. Please **don't** submit these tasks as questions on Stack Overflow! And please **don't** share your answers to these tasks outside of your own pair.

1 Verifying an 8-bit multiplier

Here is a Verilog design (mostly due to Michalis Pardalos, with some modifications by me) for multiplying two 8-bit numbers, producing a 16-bit output.

```verilog
module multiplier (
                   input          rst,
                   input          clk,
                   input [7:0]    in1,
                   input [7:0]    in2,
                   output [15:0] out
                  );
   reg [3:0]  stage = 0;
   reg [15:0] accumulator = 0;
   reg [7:0]  in1_shifted = 0;
   reg [15:0] in2_shifted = 0;


   // Logic for controlling the stage
   always @(posedge clk)
     if (rst || stage == 9)
       stage <= 0;
     else
       stage <= stage + 1;

   // Logic for in1_shifted and in2_shifted
   always @(posedge clk)
     if (rst) begin
        in1_shifted <= 0;
        in2_shifted <= 0;
     end else if (stage == 0) begin
        in1_shifted <= in1;
        in2_shifted <= in2;
     end else begin
        in1_shifted <= in1_shifted >> 1;
        in2_shifted <= in2_shifted << 1;
     end

   // Logic for the accumulator
   always @(posedge clk)
     if (rst || stage == 9) begin
        accumulator <= 0;
     end else if (in1_shifted[0]) begin
        accumulator <= accumulator + in2_shifted;
     end

   // Output logic
   assign out = accumulator;
endmodule
```

Use SymbiYosys to prove the following properties about the design. Most of the properties are phrased in a slightly vague or slightly incorrect way. Part of your job as a verification engineer is to make these properties *precise* and *correct*.

You can assume that we are only interested in assertions that hold at rising clock edges, such as `always @(posedge clk) assert property (foo);`

If it helps for any of the questions below, you can add extra registers to your design that you use during the proof, as long as these registers only appear inside the `` `ifdef FORMAL ... `endif `` block so that they cannot affect synthesis.

1. Devise a tight upper bound on the value in `out`, and prove that `out` never exceeds this bound.
2. Prove that `stage` increments on each clock cycle.
3. Prove the main property: that if the multiplier is in stage 0 and `in1` holds value x and `in2` holds value y, then 9 cycles later the value of `out` will be equal to $x \times y$. *[Hint: you can write* `$past(e,n)` *to refer to the value of expression* `e` *from* `n` *clock cycles ago, where* `n` *is an integer constant.]*
4. Prove that the value in `out` monotonically increases during the computation.
5. Prove that in the fourth stage of computation, `accumulator` holds the initial value of `in2` multiplied by the lowest 4 bits of the initial value of `in1`.
6. Prove similar properties about the value of the `accumulator` in the other stages.
7. Prove that `in1_shifted` always holds the initial value of `in1`, shifted right by `stage` bits.
8. Prove that `in2_shifted` always holds the initial value of `in2`, shifted left by `stage` bits.
9. Use a `cover` statement to prove that 13 is a prime number.
10. Can you combine all of the properties you wrote for Questions 5 and 6 together into a single, concise property?