INITIAL CODE LISTING:

```
`timescale 1ns / 1ps
module my_state_machine2(
  output reg startA,
  output ready,
  input goA,
  input readyA,
  input clk,
  input rst);

reg [7:0] CS;

parameter S_init       = 8'b00000000;
parameter S_startA0    = 8'b00000001;
parameter S_startA1    = 8'b00000010;

assign ready = readyA;

always @ (posedge clk) begin
  if (rst == 1)
    CS<=S_init;
  else
    case (CS)
      S_init: begin
        if (goA == 1) begin
          startA <= 1;
          CS<=S_startA0;
        end else begin
          startA <= 0;
          CS<=S_init;
        end
      end
      S_startA0: begin
        startA <= 1; //***
        CS<=S_startA1;
      end
      S_startA1: begin
        startA <= 0;
        CS<=S_init;
      end
    endcase
end
endmodule
```
1. Enter the code.
   - Write a testbench and demonstrate its behavior.
   - Synthesize the design and show the RTL view results. Briefly explain what you see.
2. remove the line marked //***.
   - Show and explain what if any the change in the RTL view was. Does it make sense?
   - Run the same testbench from problem 1 and explain the results.
3. Try to code the exact same behavior using separate blocks for registers and combinatorial logic while maintaining registered outputs. Your state machine case statement should have the same number of case items. The output of your combinatorial block should be a signal called `startA_prereg` which feeds into a register to output `startA`.
   - Again, show the testbench results
   - Show the RTL view and explain any differences from the original implementation
4. Similar to problem 2, remove the corresponding assignment to `startA_prereg` in the case item equivalent to state `S_startA0`
   - What warning do you get?
   - Identify and show the source of the warning in the RTL view.
   - Explain.
5. Repeat 3 without explicitly coding for registered outputs, (use a blocking statement to assign `startA` and no `startA_prereg` should exist).
6. Modify all three implementations to output `startA` for 256 clk cycles before setting it low again for at least one clk cycle before any restart. Write a testbench and demonstrate operation when `go` is set and left high.
7. Modify all three implementations in problem 6 to immediately run again without dropping `startA` if “go” is left high. Write a testbench and demonstrate operation.
8. Modify both implementations to start only on when `go` changes to 1, allowing immediate restart without any drop of `startA` if `go` goes high at the right time. Show testbench results.
   - Show that systems start on a first high of `go`.
   - Show that systems don't rerun if `go` is left high.
   - Show that two cycles will run without `startA` going low if `go` is set high at the right time.