System-on-Chip Design Supervision Answer (Set 2)

Wei Song 19/04/2016

This answer provides a summary of key points in each question. It is the understanding of the supervisor, which does not necessarily always correct and does not represent the view of lecturers. It is expected to expand the key points into detailed description when similar questions are asked in exams.

SP 4. RTL, simulation, and hazards

Q1: Convert the following behavioural RTL into an equivalent one using only non-blocking assignments.

```
always @(posedge clk) begin
   foo = bar + 22;
   if (foo > 17)
      foo = 17;
   foo_final = foo;
   foo = 0;
end
```

```
always @(posedge clk) begin
   foo <= 0;
   foo_final <= (bar + 22 > 17) ? 17 : bar + 22;
end
```

Q2: Give a fragment of RTL that implements a counter that wraps after 7 clock ticks.

```
reg [3:0] count;
always @(posedge clk) count <= count == 6 ? 0 : count + 1;
```

Q3: Implement a 16-bit multiplier that uses only 8-bit multipliers and adders. The finished multiplier should be fully pipelined, producing a result in every cycle with a 2 cycles delay and using the minimal number of 8-bit multipliers. (You can use * and + to represent an 8-bit multiplier and an adder respectively)

```
module multiplier  (input clk, input [15:0] A, B, output [31:0] C);
   reg [15:0] AhBh, AlBh, AhBl, AlBl;
   always @(posedge clk) begin
      AhBh <= A[15:8] * B[15:8];
      AhBl <= A[15:8] * B[7:0];
      AlBh <= A[7:0] * B[15:8];
      AlBl <= A[7:0] * B[7:0];
   end
   assign C = (AhBh << 16) + (AhBl << 8) + (AlBh << 8) + AlBl;
endmodule
```

Q4: [optional] Implement a FIFO with a depth of 8. The width of data is 16-bit, and the FIFO has full and empty signals.

```
module fifo #(parameter W=16, L=8)
  (
   input clk, rst, write, read
   input [W-1:0] din,
   output full,
   output [W-1:0] dout,
   output reg empty
   );

  localparam PW = $clog2(L);

  reg [PW-1:0] wp, rp;
  reg [W-1:0]  data [L-1:0];

  function [PW-1:0] incr (input [PW-1:0] p);
    return p == L-1 ? 0 : p + 1;
  endfunction

  always @(posedge clk or posedge rst)
    if(rst) wp <= 0;
    else if(write && !full) begin
      wp <= incr(wp);
      data[wp] <= din;
    end

  always @(posedge clk or posedge rst)
    if(rst) rp <= 0;
    else if(read && !empty)
      rp <= incr(rp);

  always @(posedge clk or posedge rst)
    if(rst) empty <= 1'b1;
    else if(write) empty <= 1'b0;
    else if(read && wp == incr(rp))
      empty <= 1'b1;

  assign full = wp == rp && !empty;
  assign dout = data[rp];
endmodule // fifo
```

SP 5. Assertion based design

Q5: What is the difference between a safety and liveness assertion over the behavior of system? How can safety and liveness assertions be used in dynamic validation? Give a short segment of RTL that contains an imperative assertion that holds and give also a pair of valid safety and liveness assertions that holds.

A safety property is one that always holds while a liveness property is something that can be eventually achieved in the future, regardless of the current system state. Dynamic validation is a simulation. In a simulation, safety assertions can be checked whether it is ever violated in the simulation. Liveness assertion cannot be fully verified during simulation. However, a simulation may report at the end whether the liveness condition has been reached.

For the counter in Q2, a liveness property is count == 6, a safety property is count != 7.

Q6: What is the difference between black box testing and white box testing? Can assertions be used for black box testing?

Black box testing is where the implementation details of the design under test are hidden and assertions must be made on the ports. White box testing allows the internal states of the design under test to be monitored and asserted. Assertion can be sued for black box testing but can only be used for the port signals.

Q7: What is "assertion based design"? What is the meaning of coverage? Explain how certain assertions can be reused at different layers of modeling abstraction.

Assertion-based design is a methodology where assertions are written during and in advance of coding and tested at the earliest possible time. Coverage refers to the fraction of the source code that was executed and tested in a test run. Correlation patterns between if statements or system states can also be considered. The state transition of global state machines, bus protocols and packet formats are normally shared from high-level behavioural models down to the detailed RTL implementations. The assertions used to check the correctness of these shared behaviour can be thus reused throughput the design hierarchy.

Q8: What is a combinational equivalence problem and what is a sequential equivalence problem? Why might sequential equivalence be violated? And why might we see false negatives in a sequential equivalence checker?

A combinational equivalence problem is to check whether two implementations of stateless logic are functional equivalent after considering don't care states. A sequential equivalence problem is to check whether two implementations with internal states (sequential logic) are functional equivalent. For sequential logic, besides considering don't care states, the checker may also ignore the difference of delay in the transition of states.

Sequential equivalence check may fail for many reasons. EDA tools can causes errors either by improper use of a tool or some bugs in a tool. The implementation from high-level to low-level models may not be fully automated. Manual implementation also leads to human errors.

There are two types of false errors. Detailed implementations may utilize some optimization, which is not observable in high-level modules, such as removing some impractical states or function simplification. Checkers should be informed of such optimization. The other type of false errors is due to the transitional states introduced in detailed implementation, such as the transitional states needed when implementing a coherent cache using directory based protocols. These transitional states are not observable in behavioral models and might be ruled out by checkers as invalid states; however they are actually valid in a detailed implementation. Similarly, checkers need to be relaxed to cope with such transitional states.