SymbiYosys coursework exercises

John Wickerson

Autumn term 2023

There are two tasks, both of which should be completed, and both of which are worth 50% of your total mark.

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`. Some of the tasks contain questions that require short written answers; these answers can be provided as 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 First task: Verifying a binary-to-BCD converter

Here is a Verilog design (due to Peter Cheung) for performing conversion from 10-bit binary numbers into binary-coded decimal (BCD).

```
//------------------------------
// Module name: bin2bcd_10
// Function: Converts a 10-bit binary number to 4 digits BCD
//           .... using the shift-and-add3 algorithm
// Creator:  Peter Y.K. Cheung
// Version:  1.0
// Date:     19 Sept 2016
//------------------------------


module add3_ge5(iW,oA);

 input [3:0] iW;
 output reg [3:0] oA;

 always @ (iW)
   case (iW)
     4'b0000: oA <= 4'b0000;
     4'b0001: oA <= 4'b0001;
     4'b0010: oA <= 4'b0010;
     4'b0011: oA <= 4'b0011;
     4'b0100: oA <= 4'b0100;
     4'b0101: oA <= 4'b1000;
     4'b0110: oA <= 4'b1001;
     4'b0111: oA <= 4'b1010;
     4'b1000: oA <= 4'b1011;
     4'b1001: oA <= 4'b1100;
     4'b1010: oA <= 4'b1101;
     4'b1011: oA <= 4'b1110;
     4'b1100: oA <= 4'b1111;
     default: oA <= 4'b0000;
   endcase
endmodule

module bin2bcd_10 (B, BCD_0, BCD_1, BCD_2, BCD_3);

  input [9:0] B;
  output [3:0]  BCD_0, BCD_1, BCD_2, BCD_3;

  wire [3:0] w1,w2,w3,w4,w5,w6,w7,w8,w9,w10,w11,w12;
  wire [3:0] a1,a2,a3,a4,a5,a6,a7,a8,a9,a10,a11,a12;

  add3_ge5 A1 (w1,a1);
  add3_ge5 A2 (w2,a2);
  add3_ge5 A3 (w3,a3);
  add3_ge5 A4 (w4,a4);
  add3_ge5 A5 (w5,a5);
  add3_ge5 A6 (w6,a6);
  add3_ge5 A7 (w7,a7);
```

```
  add3_ge5 A8 (w8,a8);
  add3_ge5 A9 (w9,a9);
  add3_ge5 A10 (w10,a10);
  add3_ge5 A11 (w11,a11);
  add3_ge5 A12 (w12,a12);

  assign w1 = {1'b0, B[9:7]};
  assign w2 = {a1[2:0], B[6]};
  assign w3 = {a2[2:0], B[5]};
  assign w4 = {1'b0, a1[3], a2[3], a3[3]};
  assign w5 = {a3[2:0], B[4]};
  assign w6 = {a4[2:0], a5[3]};
  assign w7 = {a5[2:0], B[3]};
  assign w8 = {a6[2:0], a7[3]};
  assign w9 = {a7[2:0], B[2]};
  assign w10 = {1'b0, a4[3], a6[3], a8[3]};
  assign w11 = {a8[2:0], a9[3]};
  assign w12 = {a9[2:0], B[1]};

  assign BCD_0 = {a12[2:0], B[0]};
  assign BCD_1 = {a11[2:0], a12[3]};
  assign BCD_2 = {a10[2:0], a11[3]};
  assign BCD_3 = {3'b0, a10[3]};
endmodule
```

Use SymbiYosys to prove that the value in each 4-bit BCD digit is always in the correct range to encode a decimal digit. Then use SymbiYosys to prove that the numerical value of the BCD output is always equal to the numerical value of the binary input. What value(s) of k are required for those k-inductions, and why?

Now using SymbiYosys in `cover` mode, establish whether every valid 4-digit BCD number can be produced by this design.

2 Second task: Verifying a circular queue

Here is a Verilog design for a circular queue.

```
module queue
  #( parameter ADDR=5, // width of address bus; can be anything
     parameter DATA=42, // width of data bus; can be anything
     parameter Q_SIZE=32) // must be equal to 2 ^ ADDR
   (
    input                clk, // clock
    input                rst, // reset
    input                wen, // write-enable
    input                ren, // read-enable
    input [DATA-1:0]     wdata, // data in
    output [DATA-1:0]    rdata, // data out
```

```
12      output reg [ADDR:0] count, // how many elements in queue
13      output              full, // queue is full
14      output              empty // queue is empty
15      );
16
17    reg [ADDR-1:0]       waddr; // next cell to write to
18    reg [ADDR-1:0]       raddr; // next cell to read from
19    reg [DATA-1:0]       data [Q_SIZE-1:0]; // contents of queue
20
21    // incrementing the write-address when write-enable is set
22    initial waddr = 0;
23    always @(posedge clk or posedge rst) begin
24       if (rst)
25         waddr <= 0;
26       else if (wen || (ren && empty))
27         waddr <= waddr + 1;
28    end
29
30    // incrementing the read-address when read-enable is set
31    initial raddr = 0;
32    always @(posedge clk or posedge rst) begin
33       if (rst)
34         raddr <= 0;
35       else if (ren || (wen && full))
36         raddr <= raddr + 1;
37    end
38
39    // updating the count
40    initial count = 0;
41    always @(posedge clk or posedge rst) begin
42       if (rst)
43         count <= 0;
44       else if (wen && !ren && count < Q_SIZE)
45         count <= count + 1;
46       else if (ren && !wen && count > 0)
47         count <= count - 1;
48    end
49
50    // synchronously writing to the queue
51    always @(posedge clk)
52      if (wen) data[waddr] <= wdata;
53
54    // asynchronously reading from the queue
55    assign rdata = data[raddr];
56
57    // setting the full/empty signals
58    assign full = count == Q_SIZE;
59    assign empty = (count == 0) && ~rst;
60
```

```
61 `ifdef FORMAL
62
63    // TODO: assertions go here
64
65 `endif
66
67 endmodule
```

Your task is to state and prove sixteen properties of this queue. Most of them are phrased in a slightly vague way, so you will have to make a judgement about how best to translate them into precise assertions. Several of them are not strictly correct, either. For instance, the property might say '`foo` goes up by one', but you might need to refine this to 'as long as the reset signal has not been set, then `foo` goes up by one, unless it is equal to 127, in which case it returns to zero'.

Please add a comment next to each of the assertions you write to give the number of the property it is intending to capture, and also explain whether you had to make any refinements to each property.

1. The write-address is always less than `Q_SIZE`.
2. The read-address is always less than `Q_SIZE`.
3. `count` is always less than `Q_SIZE`.
4. Between successive clock cycles, `count` never changes by more than one.
5. The reset signal makes all the outputs go low.
6. The `full` signal is asserted if and only if the queue is full.
7. The `empty` signal is asserted if and only if the queue is empty.
8. If the write-address and the read-address differ, then `count` is calculated from their difference.
9. If the write-address and the read-address are the same, then `count` is either `Q_SIZE` or 0.
10. `rdata` always holds the element of `data` at the read-address.
11. When write-enable is set, `wdata` is latched into the `data` array at the write-address.
12. If the queue is empty and write-enable is set, then in the next clock cycle, `rdata` will hold the previous value of `wdata`.
13. Setting read-enable causes the read-address to increment.

14. Setting write-enable causes the write-address to increment.

15. If read-enable is low, the read-address doesn't change.

16. If write-enable is low, the write-address doesn't change.

A final note: all of the assertions you write should hold at every rising edge of the clock. That is, they should all be of the form:

```
always @(posedge clk) assert (foo);
```

or:

```
assert property (@(posedge clk) foo);
```