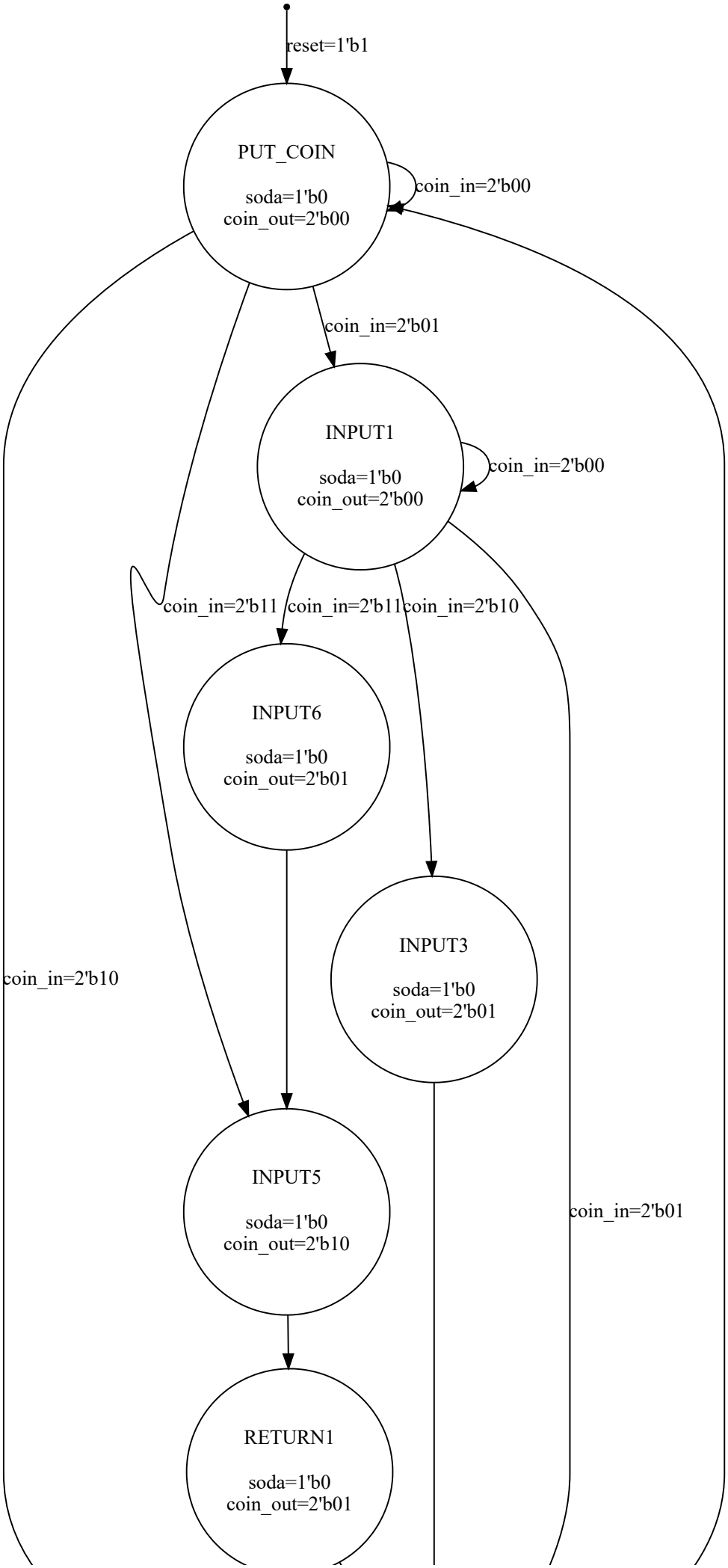
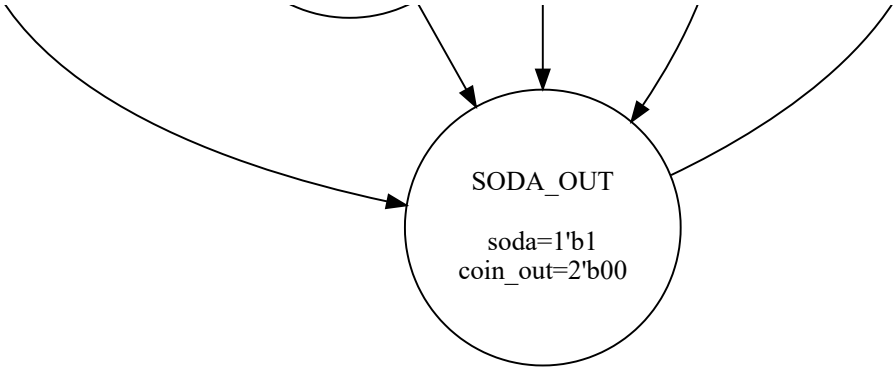
Computation II: embedded system design (5EIB0)

Available from: Friday, 5 April 2024, 2:00 PM
Requested files: vending_machine.v (Download)
Type of work: Individual work

Assignment

The figure below shows the state transition diagram of a Finite State Machine(FSM) which describes the functionality of a soda vending machine. The machine sells soda cans that cost **€2** each. Also, the machine accepts only three types of currency denominations: **€1, €2 and €5**. The machine determines when to dispense a can and also, returns the required change. In this assignment you will debug a Verilog implementation of its Finite State Machine (FSM) with the following specifications. Including the clock (clk) and reset signals, the implemented FSM will have 3 input signals and 2 output signals.


reset=1'b1
PUT_COIN
soda=1'b0
coin_out=2'b00
coin_in=2'b00
coin_in=2'b01
INPUT1
soda=1'b0
coin_out=2'b00
coin_in=2'b00
coin_in=2'b11
coin_in=2'b11
coin_in=2'b10
INPUT6
soda=1'b0
coin_out=2'b01
INPUT3
soda=1'b0
coin_out=2'b01
coin_in=2'b10
INPUT5
soda=1'b0
coin_out=2'b10
coin_in=2'b01
RETURN1
soda=1'b0
coin_out=2'b01



The clk input signal is 1-bit wide, the reset input signal is 1-bit wide and the coin_in input signal is 2-bits wide. The state elements of the FSM are to capture their inputs on the positive clock (clk) edge and reset their state when the synchronous reset signal is low. A reset can be asserted (synchronously) at any time causing all state elements to reset and hence a transtion to the initial state.

The soda output signal is 1-bit wide and the coin_out output signal is 2-bits wide. Initial values of the output signals upon reset are shown in the FSM diagram.

Your solution will be tested using bounded model checking against a golden reference implementation. The model checking software will stop at the earliest moment that your implementation diverges from the behaviour of the golden reference implementation. If a divergence takes place, a waveform will be created showing a trace of the module signals resulting in the divergent state. This waveform also shows the behaviour of the golden reference (REF) so that you can see what your implementation (DUT) should have done differently.

Please note that the golden reference implementation supersedes all other implementation descriptions. If the output from your solution implementation does not diverge from the golden reference implementation then your solution is marked as correct. Otherwise, your solution is incorrect and you should use the waveform that is produced to assist you to correct your implementation.

The currency denominations are encoded as follows: No input = 00, €1 = 01, €2 = 10 & €5 = 11.

Debug

Click on the symbol marked below to see the waveforms produced by your design. Please note that if your code has an error that prevents it being simulated it will not produce any waveforms.

Requested files

vending_machine.v

```
module vending_machine (
    input clk,
    input reset,
    output [1 : 0] coin_in,
    output soda,
    output [1 : 0] coin_out
);

// define unique constants to identify each state
localparam PUT_COIN = 3'd0;
localparam INPUT1 = 3'd1;
localparam INPUT5 = 3'd2;
localparam INPUT6 = 3'd3;
localparam INPUT3 = 3'd4;
localparam RETURN1 = 3'd5;
localparam SODA_OUT = 3'd6;

// declare verilog variables of type reg for use in always blocks
// <name>_r is used to infer a register in the clocked always block
// <name>_nxt is used assign the next state of the register in the combinational always block
reg [2 : 0] state_r, state_nxt;
reg soda_r, soda_next;
reg [1 : 0] coin_out_r, coin_out_nxt;

// clock synchronous always block will only be evaluated on the positive edge of the clock
// only use non-blocking assignments (<=) in this block
always @(posedge clk) begin
    // check for the reset signal on the clk edge, infering a synchronous reset
    // if the module is not being reset, assign all of the derived combinational <name>_nxt values to <name>_r
    // if the module is being reset, assign initial constant values to the <name>_r variables
    // this will infer a register for <name>_r if it is assigned defined values at all times
    // otherwise an unintentional latch will be inferred
    if (reset == 1'b0) begin
        // module is not being reset
        // assign all of the derived combinational <name>_nxt values to the respective <name>_r
        state_r = state_nxt;
        soda_r = soda_nxt;
        coin_out_r = coin_out_nxt;
    end else begin
        // module is being reset
        // assign constant values to each <name>_r variable
        state_r = PUT_COIN;
        soda_r = 1'b0;
        coin_out_r = 2'b00;
    end
end

// combinational always block will evaluate whenever any signal in its sensitivity list changes
// here we use the wildcard * sensitivity list, which means that the list will be inferred from the assignments in the block
// only use blocking assignments (=) in this block
always @(*) begin
    // make sure that <name>_nxt signals are always defined to avoid latches
    // to ensure register elements minimally retain their last value, assign each <name>_nxt its respective <name>_r value
    coin_out_nxt = coin_out_r;

    // case statement is used to perform different logical derivations depending on current state
    // state_r stores the current state of this FSM
    // the unique state identifiers that were defined near the top of this file are used to identify the current state
    case (state_r)
        // state_r will remain constant between positive clock edges
        // only one unique state can match at any time
        PUT_COIN: // evaluate logic for state PUT_COIN
        begin
            soda_nxt = 1'b0;
            coin_out_nxt = 2'b00;
            if (coin_in == 2'b00)
            begin
                state_nxt = PUT_COIN;
            end
            else if (coin_in == 2'b01)
            begin
                state_nxt = INPUT1;
            end
            else if (coin_in == 2'b10)
            begin
                state_nxt = INPUT5;
            end
            else if (coin_in == 2'b11)
            begin
                state_nxt = SODA_OUT;
            end
        end

        INPUT1: // evaluate logic for state INPUT1
        begin
            soda_nxt = 1'b0;
            coin_out_nxt = 2'b00;
            if (coin_in == 2'b00)
            begin
                state_nxt = INPUT1;
            end
            else if (coin_in == 2'b01)
            begin
                state_nxt = INPUT6;
            end
            else if (coin_in == 2'b10)
            begin
                state_nxt = INPUT3;
            end
            else if (coin_in == 2'b11)
            begin
                state_nxt = SODA_OUT;
            end
```

```
        end

        INPUT5: // evaluate logic for state INPUT5
        begin
            soda_nxt = 1'b0;
            coin_out_nxt = 2'b10;
            state_nxt = RETURN1;
        end

        INPUT6: // evaluate logic for state INPUT6
        begin
            soda_nxt = 1'b0;
            coin_out_nxt = 2'b01;
            state_nxt = INPUT6;
        end

        INPUT3: // evaluate logic for state INPUT3
        begin
            soda_nxt = 1'b0;
            coin_out_nxt = 2'b01;
            state_nxt = SODA_OUT;
        end

        RETURN1: // evaluate logic for state RETURN1
        begin
            soda_nxt = 1'b0;
            coin_out_nxt = 2'b01;
            state_nxt = SODA_OUT;
        end

        SODA_OUT: // evaluate logic for state SODA_OUT
        begin
            soda_nxt = 1'b1;
            coin_out_nxt = 2'b00;
            state_nxt = PUT_COIN;
        end

        default: // should not be reachable if the state register is initialised and updated correctly
        begin
            soda_nxt = 1'b0;
            coin_out_nxt = 2'b00;
            state_nxt = PUT_COIN;
        end
    endcase
end

// assign values to output ports
assign soda = soda_r;
assign coin_out = coin_out_r;

endmodule

```

VPL

You are logged in as Thomas Stirling Valdez (Log out)
5EIB0

Data retention summary