EECS 151/251A Homework 4

Due Monday, Feb 22th, 2021

For this HW Assignment

You will be asked to write several Verilog modules as part of this HW assignment. You are encouraged to test them to verify functionality by running them through a testbench. As in Homework 2, a highly suggested simulator is `https://www.edaplayground.com` which is a free, online, Verilog simulator.

Warning: As per our EECS151 "no register inference policy", you will need to use the register library in `EECS151.v` when using registers in your Verilog. `EECS151.v` is located at `https://inst.eecs.berkeley.edu/~eecs151/sp21/files/lib/EECS151.v`. We will only accept solutions using this library.

You can import this library by adding `` `include "EECS151.v" `` to your Verilog code.

As with previous homework assignments, **please include a short (1-2 sentence) explanation of your work and the design/analysis process unless otherwise directed in the problem.**

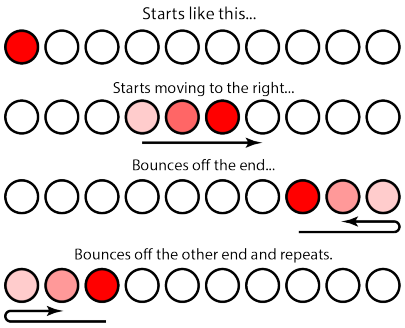
Problem 1. Light Display

As a side project, you have been consulted for the design of a simple light display for use as a decorative element in your friend's livestreaming setup. Your friend asks for the following features for their light fixture:

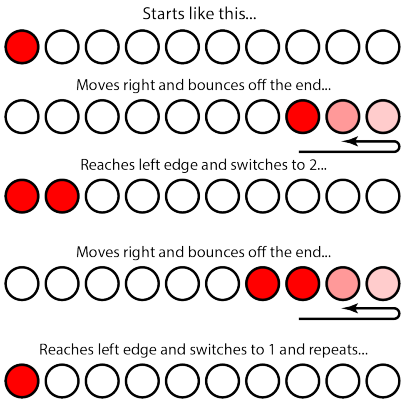
- The display is a bar of LEDs, which have a common cathode (negative connection), but independent anodes (postitive connections).
- The bar of LEDs starts up with one LED on at the left edge.
- When running, the lights will switch on and off such that it gives the appearance of the light traveling back and forth along the bar. That is it will look like one light is traveling to the right, bouncing off the right end of the bar, then travelling left until it bounces off the left end, and so on.
- There will be an input `reset` to reset the bar to the initial state of having one light on at the left end.
- There will be another input `switch` that, when on, causes the number of lights to change between just 1 light on and 2 lights on every time the light reaches the left end of the bar.
- If `switch` is off, the display will keep whichever number of lights it currently has on and maintain the bouncing behavior.

- If `reset` is on, revert to the initial state of one light at the left. This takes precedence over all other transitions.

Your friend also provides the following illustration that demonstrates the points above.



When `switch` is off



When `switch` is on

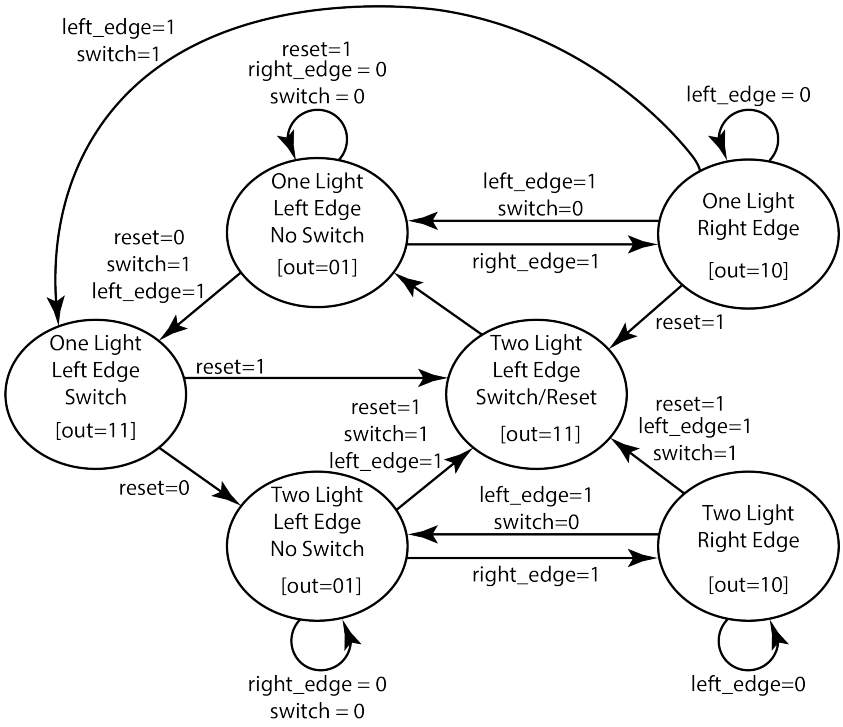
The lights will be controlled by a configurable shift register, which can change both the shifting direction as well as load in a set value, which will be used when switching between 1 and 2 lights. The shift register has a 2-bit configuration input and will shift right if the input is `01`, left if it is `10`, stop if the input is `00`, and load a new value if the input is `11`. The shift register also generates a `left_edge` and `right_edge` flag when the leftmost and rightmost bit are 1, respectively. (*You do not need to generate the loaded values, just the configuration bits of the shift register.*)

For this problem:

(a) Draw and label the State Transition Diagram for a Moore Machine that performs the above behavior by setting the configuration bits for the shift register.

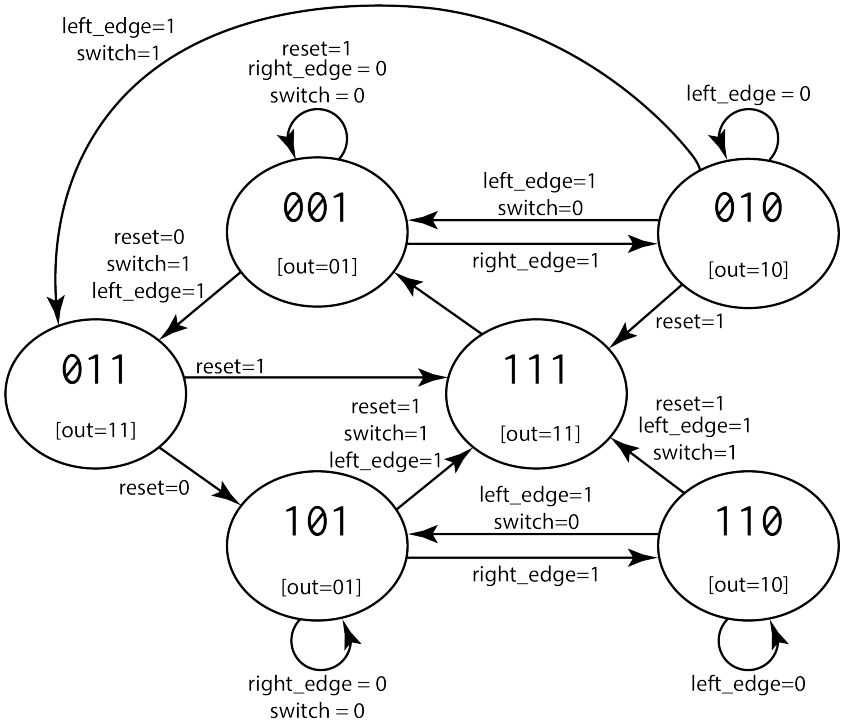
(b) Write a behavioral Verilog module of the state machine. **Remember to use the `EECS151.v` library!**

Solution:



State Transition Diagram

Here I choose a binary encoding for the Verilog implementation,



State Transition Diagram With Encoding

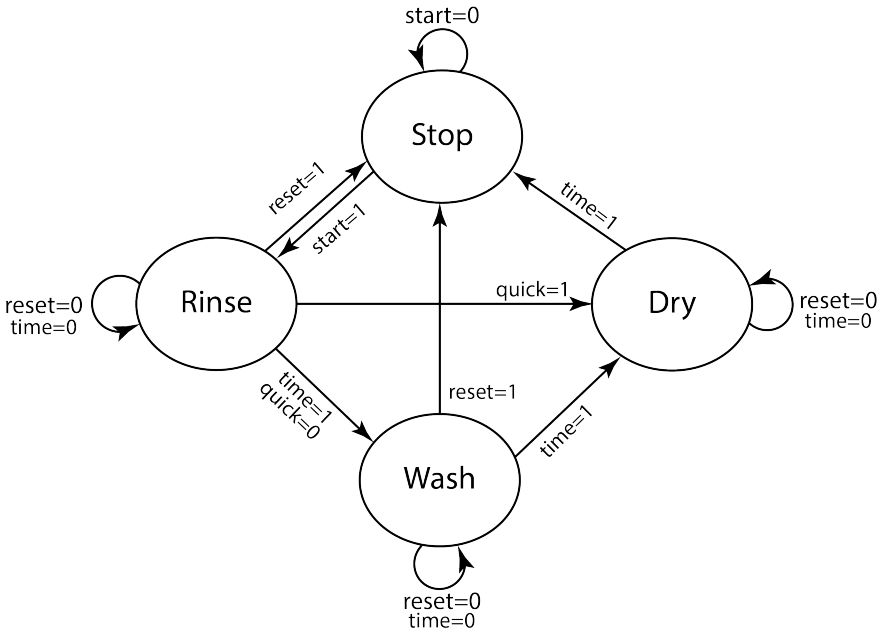
Note that I have chosen my encoding so that the output logic is extremely simple (just the 2 LSBs of the state encoding).

```
`include "EECS151.v"

module lights_fsm(
  input clk, reset, switch, left_edge, right_edge,
  output [1:0] out
  );

  reg [2:0] next_state, cur_state;

  localparam  LEFT_NO_SWITCH_1 = 3'b001;
  localparam  RIGHT_1 = 3'b010;
  localparam  LEFT_NO_SWITCH_2 = 3'b101;
  localparam  RIGHT_2 = 3'b110;
  localparam  LEFT_SWITCH_1 = 3'b011;
  localparam  LEFT_SWITCH_2 = 3'b111;

  assign out = cur_state[1:0];

  REGISTER #(.N(3)) (.d(next_state), .q(cur_state), .clk(clk));
```

```
    always @(*) begin
      case (cur_state)
        LEFT_NO_SWITCH_1: // This is our reset state
          if (reset || ~right_edge || (left_edge && ~switch)) next_state =
          ↪  cur_state;
          else if (right_edge) next_state = RIGHT_1;
          else if (switch && left_edge) next_state = LEFT_SWITCH_1;
          else next_state = cur_state;
        LEFT_NO_SWITCH_2:
          if (reset) next_state = LEFT_SWITCH_2;
          else if (~right_edge || (left_edge && ~switch)) next_state =
          ↪  cur_state;
          else if (right_edge) next_state = RIGHT_2;
          else if (switch && left_edge) next_state = LEFT_SWITCH_2;
          else next_state = cur_state;
        RIGHT_1:
          if (reset) next_state = LEFT_SWITCH_2;
          else if (~left_edge) next_state = cur_state;
          else if (switch) next_state = LEFT_SWITCH_1;
          else next_state = LEFT_NO_SWITCH_1;
        RIGHT_2:
          if (reset) next_state = LEFT_SWITCH_2;
          else if (~left_edge) next_state = cur_state;
          else if (switch) next_state = LEFT_SWITCH_2;
          else next_state = LEFT_NO_SWITCH_2;
        LEFT_SWITCH_1:
          if (reset) next_state = LEFT_SWITCH_2;
          else next_state = LEFT_NO_SWITCH_2;
        LEFT_SWITCH_2:
          next_state = LEFT_NO_SWITCH_1;
        default:
          next_state = LEFT_NO_SWITCH_1;
      endcase
    end
  endmodule
```

Problem 2. FSM Design

Consider a simple dishwasher that has an option for a quick rinse cycle or a full wash cycle. The dishwasher will first rinse the dishes with hot water, then decide whether to wash with detergent or skip straight to drying depending on the setting. If you open the door of the dishwasher prematurely, a door switch will trigger a `reset` and the dishwasher will stop and go back to the initial state. A timer raises a `time` flag when the dishwasher is ready to move to the next state. The dishwasher has three outputs: `water`, `detergent`, `heat`. During the rinse cycle, it will request water. During the wash cycle, it will request water and detergent. During the dry cycle it will request only heat. Here is a conceptual state transition diagram for the dishwasher:



For each of the following scenarios, please provide:

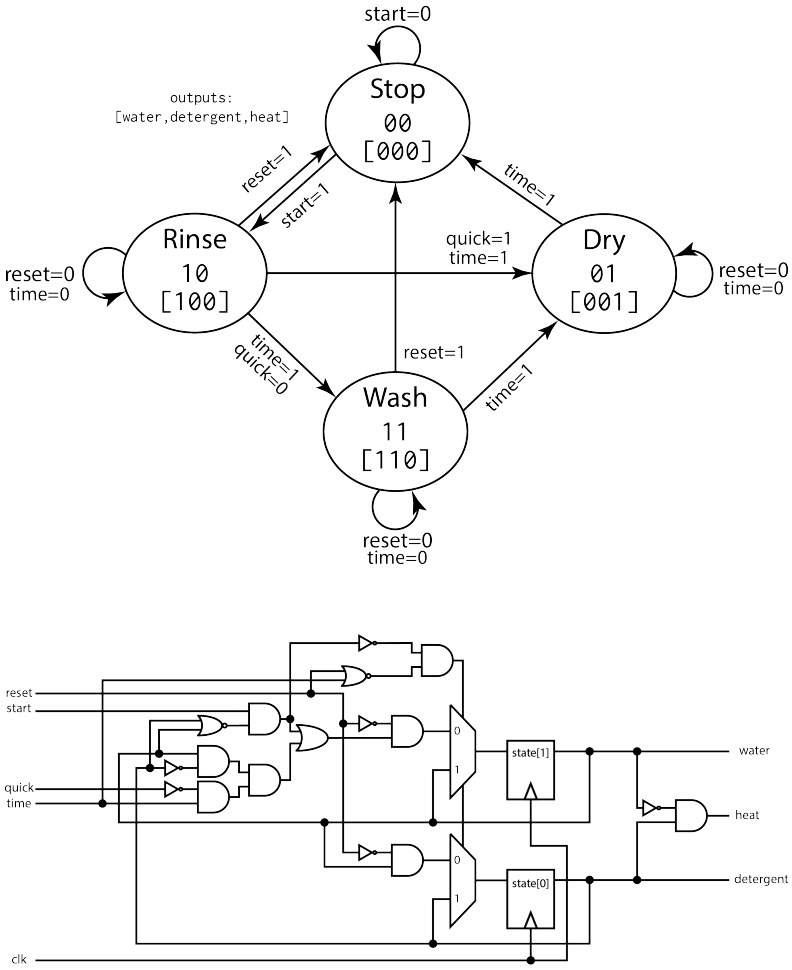
- A state transition diagram with the state names and encoding (e.g. `STOP (00)`), as well as outputs labeled appropriately.
- A circuit diagram of the state machine. The state machine will receive the inputs `reset`, `start`, `quick`, and `time`. The state machine must have the outputs `water`, `detergent`, and `heat`. You may use logic gates of 4 inputs or fewer as well as multiplexers to implement your next state logic.
- A quick (1-2 sentence) summary of your design process and decisions.

Design the FSM for each of these scenarios:

(a) Binary-encoded Moore Machine

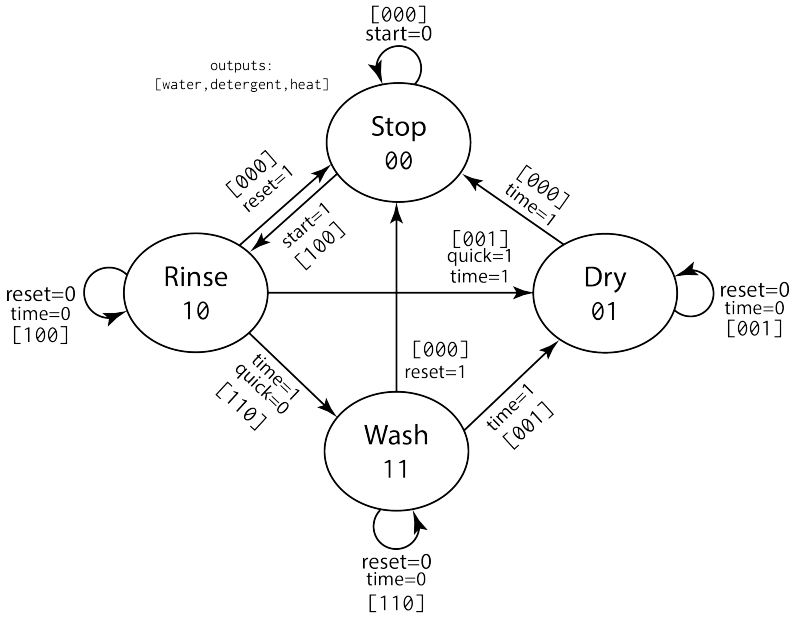
Solution:

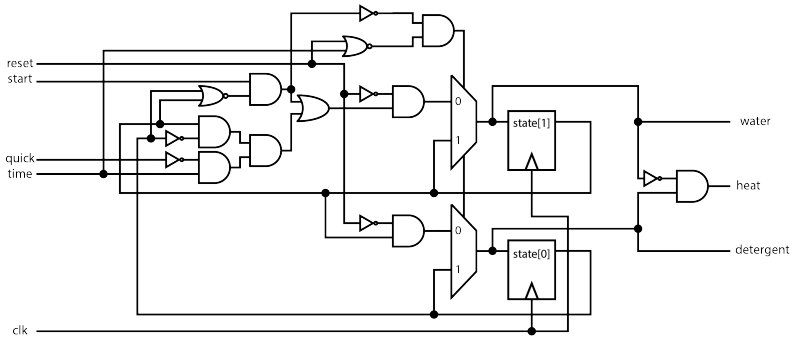
There are many approaches to this problem. In this case, I have purposely encoded my states so that the output logic is relatively simple.



(b) Binary-encoded Mealy Machine

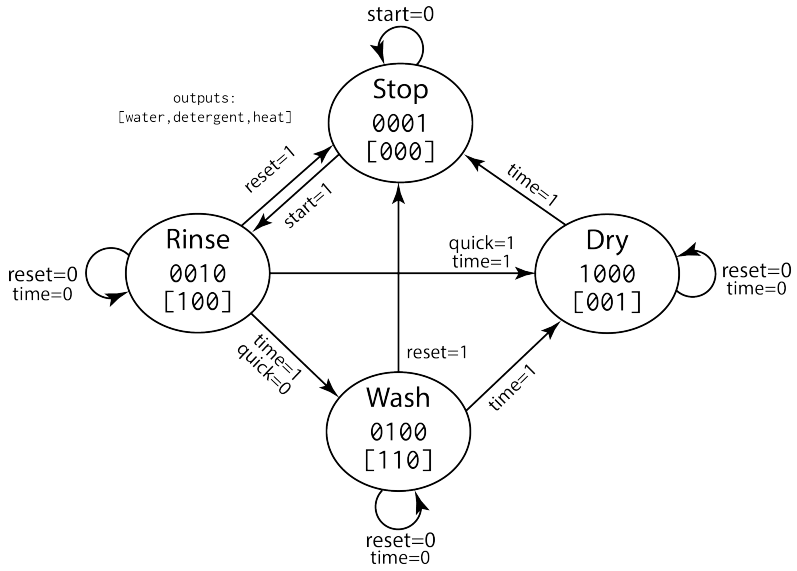
Solution:

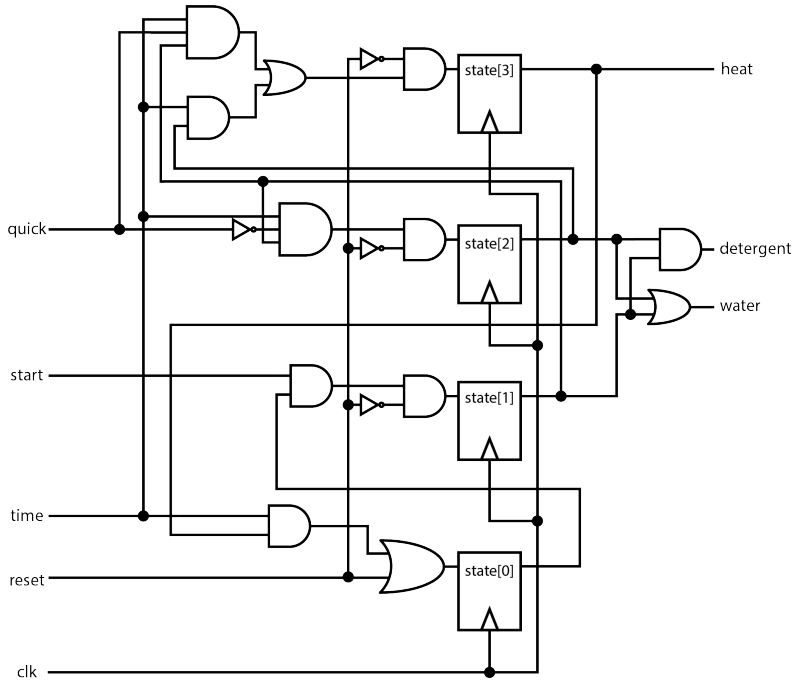




(c) One-hot Moore Machine

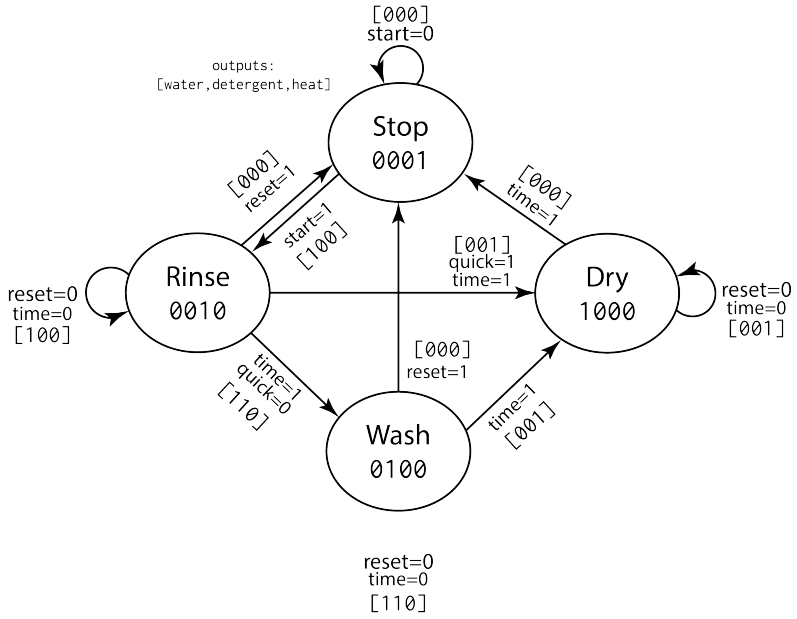
Solution:

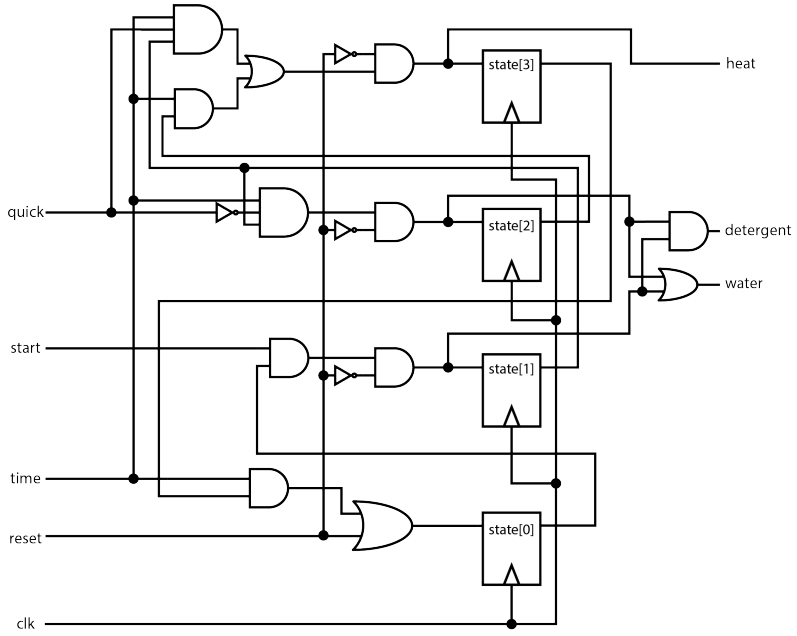




(d) One-hot Mealy Machine

Solution:


[000]
start=0
outputs:
[water,detergent,heat]
Stop
0001
[000]
reset=1
start=1
[100]
[000]
time=1
[001]
quick=1
time=1
Rinse
0010
Dry
1000
reset=0
time=0
[100]
reset=0
time=0
[001]
[000]
reset=1
time=1
quick=0
[110]
time=1
[001]
Wash
0100
reset=0
time=0
[110]


state[3]
heat
quick
state[2]
detergent
water
start
state[1]
time
reset
state[0]
clk

Problem 3: Bit-Serial Network Interface

Design an FSM that will act as a receiver for a bit-serial network interface. The interface receives each packet serially, and the packets follow a special format:

```
<src address (4 Bytes)><dst address (4B)><payload (512B)><checksum(1B)>
```

Where `src` is the address of the sending node, `dst` is the address of the intended recipient, `payload` is the data content being send, and `checksum` is the check value to detect bit corruption in the received packet. This packet arrives `src` field first, with bits following afterwards (i.e. the packet is "read" left to right).

The sender node will compute the checksum of each packet by taking the sum of all the previous bits of the packet (i.e. the `src`, `dst`, and `payload` fields) modulo 256, so the result is 1 byte long. It then takes the *1's complement* of this sum and appends it to the packet as the checksum value. The receiving node then checks the contents of the package by summing the `src`, `dst`, and `payload` fields with the checksum field. Therefore, if the bits are all correct, the final value of this sum should total to 0. If this is not 0, then the check fails and the receiver knows that the packet it has received has somehow been corrupted.

In this design, the address of the receiving node is stored in a 4B wide register, which we will name `my_addr`. Your FSM should check for both the destination address as well as the packet fidelity, each with their respective outputs. The checks and input/output specification of the receiver is as follows:

- Check that we are the intended recipient (i.e. the `dst` matches `my_addr`), and if the match succeeds, set output `packet_match` true.
- Check that the packet is uncorrupted, and if the test succeeds, set output `check_match` true.
- **The two `match` signals each have a `packet_match_valid` and `check_match_valid` signal associated with them. This `valid` signal should be set to True synchronously with the completion of their respective checks.**
- The receiver should pause the input from the serial interface until it is done processing the current packet. To achieve this, the interface will not feed new bits to the receiver until the receiver sets a `input_ready` signal high. Similarly, the interface presents a signal to the `input_valid` input of the receiver. **The receiver should only accept new bits when these signals are both high.**

You are given access to several auxilliary blocks to use as part of this state machine. There is a parametrizable clock-enabled shift register and counter that you may use to assist the state machine by generating flags to be used as additional inputs to the FSM (Verilog module definitions available here). You may use Verilog comparison operators as well as the addition operator to implement the checks.

You are free to choose between a binary-encoded SM or one-hot SM, and between Moore or Mealy machines. Please discuss this choice in the summary. For this problem, turn in:

- A state transition diagram with outputs labeled.
- A block diagram of your FSM. You may abstract the counters, adders, comparators, and shift register as black box modules, but the state registers and next state logic for the state machine must be elaborated in full.
- A behavioral Verilog implementation of the state machine.
- A short (3-4 sentence) description of your design procedure and decisions.

Solution:

```
// Problem 3
module bit_serial_receiver #(
  parameter MY_ADDR = 10
) (
  input clk,
  input rst,

  input  bits_in,
  input  input_valid,
  output input_ready,

  output packet_match,
  output packet_match_valid,
  output check_match,
  output check_match_valid
);

  localparam PLD_WIDTH = 512 * 8;
  localparam AWIDTH    = 4 * 8;
  localparam CHK_WIDTH = 8;

  localparam PKT_WIDTH = AWIDTH * 2 + CHK_WIDTH + PLD_WIDTH;

  localparam CHK_BITS_START = 0;
  localparam CHK_BITS_END   = CHK_BITS_START + CHK_WIDTH - 1;
  localparam PLD_BITS_START = CHK_BITS_END + 1;
  localparam PLD_BITS_END   = PLD_BITS_START + PLD_WIDTH - 1;
  localparam DST_BITS_START = PLD_BITS_END + 1;
  localparam DST_BITS_END   = DST_BITS_START + AWIDTH - 1;
  localparam SRC_BITS_START = DST_BITS_END + 1;
  localparam SRC_BITS_END   = SRC_BITS_START + AWIDTH - 1;

  wire [PKT_WIDTH-1:0] pkt_value;

  wire [AWIDTH-1:0] my_addr;
  REGISTER #(.N(AWIDTH)) my_addr_reg (
```

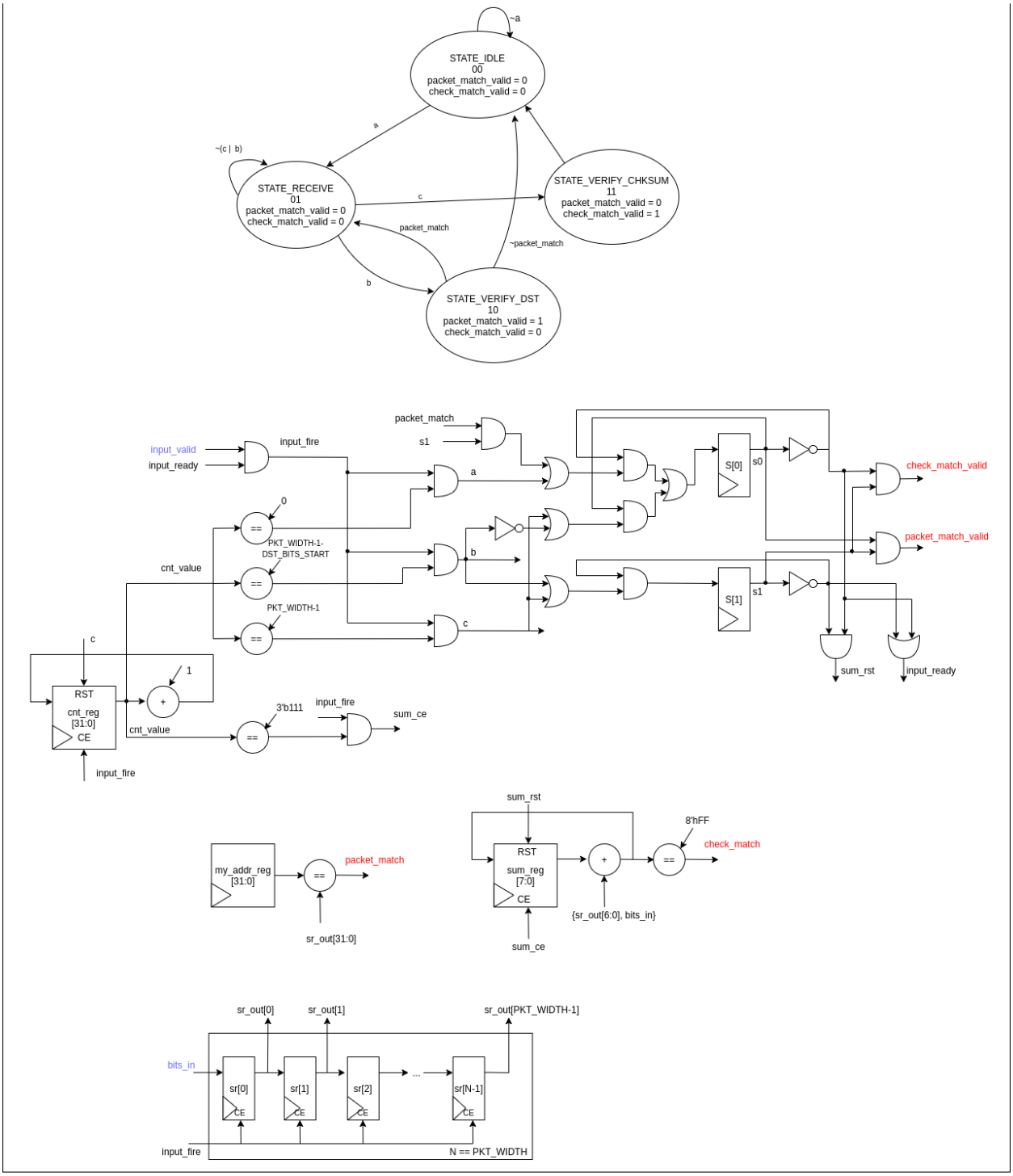
```
    .clk(clk),
    .d(MY_ADDR),
    .q(my_addr)
  );

  wire [31:0] cnt_next, cnt_value;
  wire cnt_ce, cnt_rst;
  REGISTER_R_CE #(.N(32), .INIT(0)) cnt_reg (
    .clk(clk),
    .rst(cnt_rst),
    .ce(cnt_ce),
    .d(cnt_next),
    .q(cnt_value)
  );

  wire [7:0] sum_next, sum_value;
  wire sum_ce, sum_rst;
  REGISTER_R_CE #(.N(8), .INIT(0)) sum_reg (
    .clk(clk),
    .rst(sum_rst),
    .ce(sum_ce),
    .d(sum_next),
    .q(sum_value)
  );

  wire [PKT_WIDTH-1:0] sr_out;
  wire sr_ce;
  SHIFT_REG_CE #(.N(PKT_WIDTH)) shift_reg (
    .clk(clk),
    .ce(sr_ce),
    .bit_in(bits_in),
    .bit_out(),
    .Q(sr_out)
  );

  localparam STATE_IDLE          = 2'b00;
  localparam STATE_RECEIVE       = 2'b01;
  localparam STATE_VERIFY_DST    = 2'b10;
  localparam STATE_VERIFY_CHKSUM = 2'b11;

  reg [1:0]  state_next;
  wire [1:0] state_value;

  REGISTER_R #(.N(2), .INIT(STATE_IDLE)) state_reg (
    .clk(clk),
    .rst(rst),
    .d(state_next),
```

```
    .q(state_value)
  );

  assign input_ready = (state_value != STATE_VERIFY_CHKSUM);
  wire input_fire = input_valid & input_ready;

  always @(*) begin
    state_next = state_value;
    case (state_value)
      STATE_IDLE: begin
        if (input_fire && cnt_value == 0)
          state_next = STATE_RECEIVE;
      end

      STATE_RECEIVE: begin
        if (input_fire && cnt_value == PKT_WIDTH - 1 - DST_BITS_START)
          state_next = STATE_VERIFY_DST;
        else if (input_fire && cnt_value == PKT_WIDTH - 1)
          state_next = STATE_VERIFY_CHKSUM;
      end

      STATE_VERIFY_DST: begin
        if (~packet_match)
          state_next = STATE_IDLE;
        else
          state_next = STATE_RECEIVE;
      end

      STATE_VERIFY_CHKSUM: begin
        state_next = STATE_IDLE;
      end
    endcase
  end

  assign sr_ce = input_fire;

  assign cnt_next = cnt_value + 1;
  assign cnt_ce   = input_fire;
  assign cnt_rst  = (input_fire && cnt_value == PKT_WIDTH - 1) | rst;

  assign sum_next = sum_value + {sr_out[6:0], bits_in};
  assign sum_ce   = (cnt_value[2:0] == 3'b111) && input_fire;
  assign sum_rst  = (state_value == STATE_IDLE) | rst;

  assign packet_match       = (sr_out[AWIDTH-1:0] == my_addr);
  assign packet_match_valid = (state_value == STATE_VERIFY_DST);
```

```
  assign check_match        = (sum_value[7:0] == 8'hFF);
  assign check_match_valid  = (state_value == STATE_VERIFY_CHKSUM);

endmodule
```

Source files with testbench are here

Block Diagram and State Transition Diagram as follows.


~a
STATE_IDLE
00
packet_match_valid = 0
check_match_valid = 0
a
~(c | b)
STATE_RECEIVE
01
packet_match_valid = 0
check_match_valid = 0
c
STATE_VERIFY_CHKSUM
11
packet_match_valid = 0
check_match_valid = 1
packet_match
~packet_match
b
STATE_VERIFY_DST
10
packet_match_valid = 1
check_match_valid = 0
input_valid
input_ready
input_fire
packet_match
s1
a
b
c
0
PKT_WIDTH-1-DST_BITS_START
PKT_WIDTH-1
cnt_value
S[0]
S[1]
s0
s1
check_match_valid
packet_match_valid
sum_rst
input_ready
RST
cnt_reg
[31:0]
CE
1
3'b111
sum_ce
my_addr_reg
[31:0]
sr_out[31:0]
8'hFF
check_match
sum_reg
[7:0]
{sr_out[6:0], bits_in}
bits_in
sr_out[0]
sr_out[1]
sr_out[2]
sr_out[N-1]
sr_out[PKT_WIDTH-1]
N == PKT_WIDTH