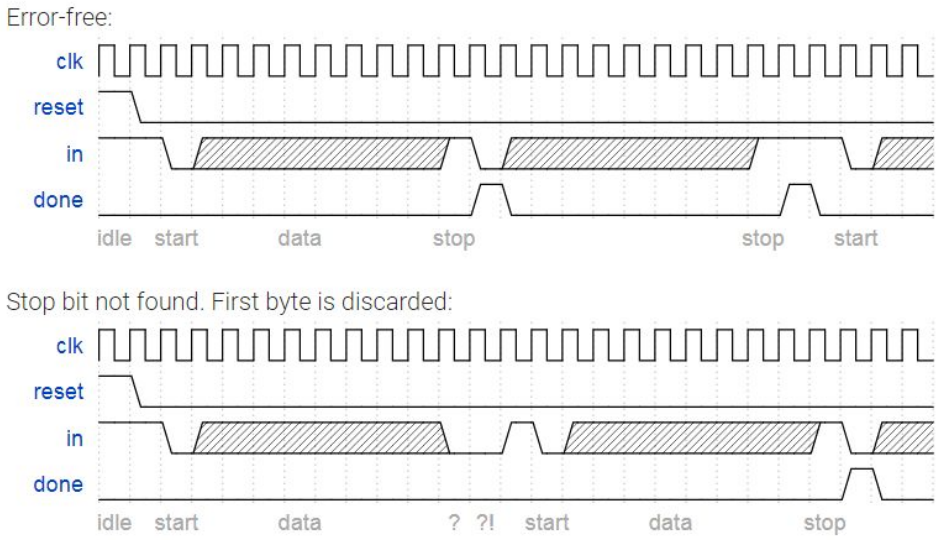
Fsm serial

In many (older) serial communications protocols, each data byte is sent along with a start bit and a stop bit, to help the receiver delimit bytes from the stream of bits. One common scheme is to use one start bit (0), 8 data bits, and 1 stop bit (1). The line is also at logic 1 when nothing is being transmitted (idle).

Design a finite state machine that will identify when bytes have been correctly received when given a stream of bits. It needs to identify the start bit, wait for all 8 data bits, then verify that the stop bit was correct. If the stop bit does not appear when expected, the FSM must wait until it finds a stop bit before attempting to receive the next byte.

Some timing diagrams




```
module top_module(
    input clk,
    input in,
    input reset,    // Synchronous reset
    output done
);
    parameter      WAIT  = 4'b0001;
    parameter      DATA  = 4'b0010;
    parameter      CHECK = 4'b0100;
    parameter      ERR   = 4'b1000;
    reg [3:0] curr_state;
    reg [3:0] next_state;
    wire      data_to_check;
    wire      check2wait;
    always @ (*) begin
        case(curr_state)
```

```
        WAIT :next_state=(~in)?DATA:WAIT;
        DATA :next_state=(data_to_check)?CHECK:DATA;
        CHECK:next_state=( in)?WAIT:ERR;
        ERR  :next_state=( in)?WAIT:ERR;
        default:next_state=curr_state;
      endcase
   end
   always@(posedge clk)begin
      if(reset)begin
         curr_state<=WAIT;
      end
      else begin
         curr_state<=next_state;
      end
   end



   reg [2:0] cnt;
   assign    data_to_check = (curr_state==DATA) & (cnt==3'd7);
   always@(posedge clk)begin
      if(reset)begin
         cnt<=3'd0;
      end
      else if(data_to_check)begin
         cnt<=3'd0;
      end
      else if(curr_state==DATA)begin
         cnt<= cnt+3'd1;
      end
   end

   assign check2wait = (curr_state==CHECK) & in;


   always@(posedge clk)begin
      if(reset)begin
         done<=1'b0;
      end
      else if(done)begin
         done<=1'b0;
      end
      else if(check2wait)begin
         done<=1'b1;
      end
   end

endmodule
```

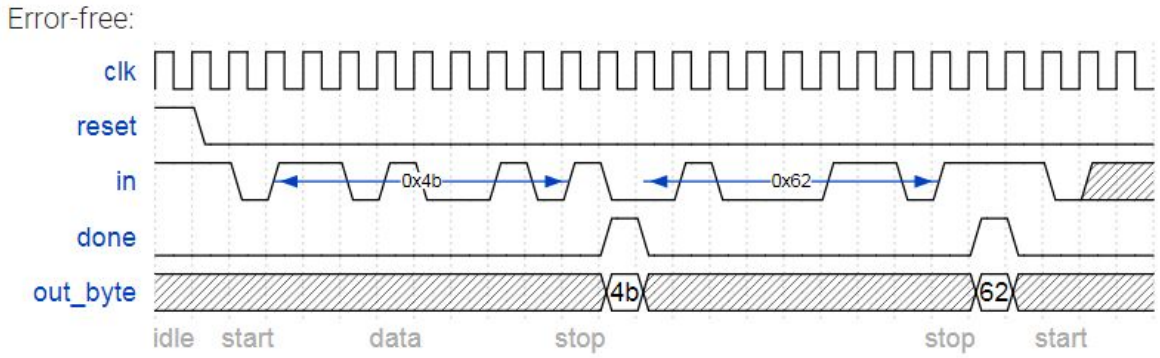
Fsm serialdata

See also: Serial receiver

Now that you have a finite state machine that can identify when bytes are correctly received in a serial bitstream, add a datapath that will output the correctly-received data byte. `out_byte` needs to be valid when done is 1, and is don't-care otherwise.

Note that the serial protocol sends the *least* significant bit first.

Some timing diagrams




```
module top_module(
    input clk,
    input in,
    input reset,    // Synchronous reset
    output [7:0] out_byte,
    output done
); //

    parameter     WAIT  = 4'b0001;
    parameter     DATA  = 4'b0010;
    parameter     CHECK = 4'b0100;
    parameter     ERR   = 4'b1000;
    reg [3:0] curr_state;
    reg [3:0] next_state;
    wire      data_to_check;
    wire      check2wait;
    always @ (*) begin
        case(curr_state)
            WAIT :next_state=(~in)?DATA:WAIT;
            DATA :next_state=(data_to_check)?CHECK:DATA;
            CHECK:next_state=( in)?WAIT:ERR;
            ERR  :next_state=( in)?WAIT:ERR;
            default:next_state=curr_state;
        endcase
    end
    always@(posedge clk)begin
        if(reset)begin
```

```
            curr_state<=WAIT;
        end
        else begin
            curr_state<=next_state;
        end
    end



    reg [2:0] cnt;
    assign    data_to_check = (curr_state==DATA) & (cnt==3'd7);
    always@(posedge clk)begin
        if(reset)begin
            cnt<=3'd0;
        end
        else if(data_to_check)begin
            cnt<=3'd0;
        end
        else if(curr_state==DATA)begin
            cnt<= cnt+3'd1;
        end
    end

    assign check2wait = (curr_state==CHECK) & in;


    always@(posedge clk)begin
        if(reset)begin
            done<=1'b0;
        end
        else if(done)begin
            done<=1'b0;
        end
        else if(check2wait)begin
            done<=1'b1;
        end
    end

    reg [7:0] shift_reg;
    always @(posedge clk )begin
        if(reset)begin
            shift_reg <= 8'b0;
        end
        else if(curr_state==DATA)begin
            shift_reg = {in,shift_reg[7:1]};
        end
    end

            assign out_byte = shift_reg;
    // New: Datapath to latch input bits.

endmodule
```

Fsm serialdp

See also: Serial receiver and datapath

We want to add parity checking to the serial receiver. Parity checking adds one extra bit after each data byte. We will use odd parity, where the number of 1s in the 9 bits received must be odd. For example, `101001011` satisfies odd parity (there are 5 1s), but `001001011` does not.

Change your FSM and datapath to perform odd parity checking. Assert the done signal only if a byte is correctly received *and* its parity check passes. Like the serial receiver FSM, this FSM needs to identify the start bit, wait for all 9 (data and parity) bits, then verify that the stop bit was correct. If the stop bit does not appear when expected, the FSM must wait until it finds a stop bit before attempting to receive the next byte.

You are provided with the following module that can be used to calculate the parity of the input stream (It's a TFF with reset). The intended use is that it should be given the input bit stream, and reset at appropriate times so it counts the number of 1 bits in each byte.

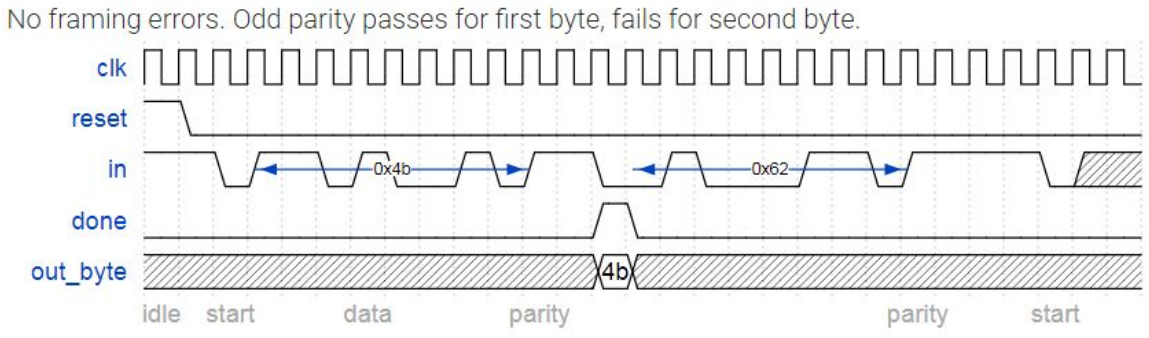
```
module parity (
    input clk,
    input reset,
    input in,
    output reg odd);

    always @(posedge clk)
        if (reset) odd <= 0;
        else if (in) odd <= ~odd;

endmodule
```

Note that the serial protocol sends the least significant bit first, and the parity bit after the 8 data bits.

Some timing diagrams




```
module top_module(
    input clk,
```

```
    input in,
    input reset,    // Synchronous reset
    output [7:0] out_byte,
    output done
); //
          parameter      WAIT  = 5'b00001;
    parameter      DATA  = 5'b00010;
    parameter      CHECK = 5'b00100;
    parameter      PAR_CK= 5'b01000;
    parameter      ERR   = 5'b10000;
    reg [4:0] curr_state;
    reg [4:0] next_state;
    wire      data_to_parcheck;
    wire      par_ok;
    wire      check2wait;
    always @ (*) begin
       case(curr_state)
          WAIT  :next_state=(~in)?DATA:WAIT;
          DATA  :next_state=(data_to_parcheck)?PAR_CK:DATA;
          PAR_CK:next_state=(par_ok)?CHECK:ERR;
          CHECK :next_state=( in)?WAIT:ERR;
          ERR   :next_state=( in)?WAIT:ERR;
          default:next_state=curr_state;
       endcase
    end
    always@(posedge clk)begin
       if(reset)begin
          curr_state<=WAIT;
       end
       else begin
          curr_state<=next_state;
       end
    end



    reg [2:0] cnt;
    assign    data_to_parcheck = (curr_state==DATA) & (cnt==3'd7);
    always@(posedge clk)begin
       if(reset)begin
          cnt<=3'd0;
       end
       else if(data_to_parcheck)begin
          cnt<=3'd0;
       end
       else if(curr_state==DATA)begin
          cnt<= cnt+3'd1;
       end
    end

    assign check2wait = (curr_state==CHECK) & in;


    always@(posedge clk)begin
       if(reset)begin
          done<=1'b0;
       end
       else if(done)begin
```

```
            done<=1'b0;
        end
        else if(check2wait)begin
            done<=1'b1;
        end
    end

    reg [7:0] shift_reg;
    always @(posedge clk )begin
        if(reset)begin
            shift_reg <= 8'b0;
        end
        else if (curr_state==WAIT & ~in)begin
            shift_reg <= 8'b0;
        end
        else if(curr_state==DATA)begin
            shift_reg = {in,shift_reg[7:1]};
        end
    end

    assign par_ok = (curr_state==PAR_CK) & (^{in,shift_reg[7:0]});

            assign out_byte = shift_reg;


endmodule
```