```
--------------------------------------------------------------------------------
--                                                                            --
--  RLLEncoder Logic Design                                                   --
--                                                                            --
--  This is an entity declaration and architecture definition for an          --
--  RLL(2, 7) econding state machine. The system takes in a serial bit        --
--  stream at half of the system clock rate and outputs a serial bit stream   --
--  giving the RLL encoded input data at the system clock rate. The output    --
--  data is delayed by 2 bits from the input data, meaning that on input of   --
--  an input bit, it will take 4 clock cycles to see the corresponding        --
--  output.                                                                   --
--                                                                            --
--  This encoder will work with the RLLEncoder test bench without edit.       --
--                                                                            --
--  Inputs:                                                                   --
--           DataIn  -  Input data bit                                        --
--           Reset   -  Active low reset signal                               --
--           CLK     -  System clock signal                                   --
--                                                                            --
--  Outputs:                                                                  --
--           RLLOut  -  RLL encoded output data bit                           --
--                                                                            --
--  Revision History:                                                         --
--      11/18/2024  Edward Speer  Initial Revision                            --
--      11/20/2024  Edward Speer  Enhance documentation                       --
--                                                                            --
--------------------------------------------------------------------------------

--
-- Imports
--

library ieee;
use ieee.std_logic_1164.all;

--
-- RLLEncoder entity declaration
--

entity RLLEncoder is
    port (
        DataIn  : in  std_logic;  -- Input data bit
        Reset   : in  std_logic;  -- Active low asynchronous reset signal
        CLK     : in  std_logic;  -- System clock signal
        RLLOut  : out std_logic   -- RLL encoded output data bit
    );
end RLLEncoder;

--
-- RLLEncoder behavioral architecture
--
```

```
architecture behavioral of RLLEncoder is

    -- The state machine has 15 states, 1-14, without any straightforward
    -- interpretation associated with each state
    type states is (S1, S2, S3, S4, S5, S6, S7, S8, S9, S10, S11, S12, S13,
                    S14);

    -- Create a signal for the current state
    signal current_state : states;

    -- Create a signal for the next state
    signal next_state : states;

begin

    process(CLK, Reset)            -- UPDATE_CURRENT_STATE
    begin
        if Reset = '0' then
            current_state <= S5;
        elsif rising_edge(CLK) then
            current_state <= next_state;
        end if;
    end process;                   -- UPDATE_CURRENT_STATE

    process(current_state, DataIn) -- NEXT_STATE_LOGIC
    begin
        case current_state is
            when S1 => next_state <= S2;
            when S2 => next_state <= S3;
            when S3 =>
                if DataIn = '1' then
                    next_state <= S4;
                else
                    next_state <= S5;
                end if;
            when S4 => next_state <= S6;
            when S5 =>
                if DataIn = '1' then
                    next_state <= S1;
                else
                    next_state <= S7;
                end if;
            when S6 =>
                if DataIn = '1' then
                    next_state <= S8;
                else
                    next_state <= S9;
                end if;
            when S7 => next_state <= S10;
            when S8 => next_state <= S3;
```

```
            when S9 => next_state <= S11;
            when S10 => next_state <= S11;
            when S11 =>
                if DataIn = '1' then
                    next_state <= S12;
                else
                    next_state <= S13;
                end if;
            when S12 =>
                if DataIn = '1' then
                    next_state <= S3;
                else
                    next_state <= S14;
                end if;
            when S13 =>
                if DataIn = '1' then
                    next_state <= S11;
                else
                    next_state <= S3;
                end if;
            when S14 => next_state <= S5;
        end case;
    end process;                    -- NEXT_STATE_LOGIC

    process(current_state)          -- OUTPUT_LOGIC
    begin
        case current_state is
            when S2 | S6 | S10 | S14 =>
                 RLLOut <= '1';
            when others => RLLOut <= '0';
        end case;
    end process;                    -- OUTPUT_LOGIC

end behavioral;
```

```
--------------------------------------------------------------------------------
--                                                                            --
--  TAPController Logic Design                                                --
--                                                                            --
--  This is an entity declaration and architecture definition for a JTAG TAP  --
--  controller state machine. The system take in the JTAG test mode select    --
--  signal, JTAG test clock, and JTAG test data input signals and outputs     --
--  the JTAG test data output signal. The controller maintains two internal   --
--  queues for the JTAG data register (32 bits) and the JTAG instruction      --
--  register (7 bits).                                                        --
--                                                                            --
--  This controller will work with the TAPController test bench without       --
--  edit.                                                                     --
--                                                                            --
--  Inputs:                                                                   --
--           TRST  -  JTAG test reset signal                                  --
--           TMS   -  JTAG test mode select signal                            --
--           TDI   -  JTAG test data input signal                             --
--           TCK   -  JTAG test clock signal                                  --
--                                                                            --
--  Outputs:                                                                  --
--           TDO   -  JTAG test data output signal                            --
--                                                                            --
--  Revision History:                                                         --
--      11/19/2024  Edward Speer  Initial Revision                            --
--      11/20/2024  Edward Speer  Enhance documentation                       --
--                                                                            --
--------------------------------------------------------------------------------

--
-- Imports
--

library ieee;
use ieee.std_logic_1164.all;

--
-- TAPController entity declaration
--

entity TAPController is
    port(
        TRST  :  in  std_logic; -- JTAG synchronous reset signal
        TMS   :  in  std_logic; -- JTAG test mode select signal
        TDI   :  in  std_logic; -- Test data in
        TCK   :  in  std_logic; -- Test clock
        TDO   :  out std_logic  -- Test data out
    );
end TAPController;

--
```

```
-- TAPController behavioral architecture
--

architecture behavioral of TAPController is

    --
    -- Type declaration of the 10 states in the controller state machine
    --

    type states is (
        RESET,       -- Reset state
        IDLE,        -- Idle state
        DSELECT,     -- Select data register state
        ISELECT,     -- Select instruction register state
        CAPTURE,     -- Begin capture on selected register
        SHIFT,       -- Shift data from TDI into selected register and out TDO
        EXIT1,       -- Either exit to IDLE or PAUSE
        PAUSE,       -- Pause state - Can either wait or exit to EXIT2
        EXIT2,       -- Either exit to update or resume shifting
        UPDATE       -- Save shifted data to selected register and exit to IDLE
    );

    --
    -- Registers
    --

    signal DR : std_logic_vector(31 downto 0);  -- Data register
    signal IR : std_logic_vector(6 downto 0);   -- Instruction register

    --
    -- State signals
    --

    signal current_state : states;     -- Signal for the current state
    signal next_state    : states;     -- Signal for the next state
    signal DR_selected   : std_logic;  -- high if DR selected, low for IR

begin

    process(TCK)                -- UPDATE_CURRENT_STATE
    begin
        if rising_edge(TCK) then
            if TRST = '1' then  -- Test reset signal is synchronous
                current_state <= RESET;
            else
                current_state <= next_state;

            end if;
        end if;
    end process;                -- UPDATE_CURRENT_STATE
```

```
process(current_state, TMS)  -- NEXT_STATE_LOGIC
begin
    case current_state is
        when RESET =>
            if TMS = '1' then
                next_state <= RESET;
            else
                next_state <= IDLE;
            end if;
        when IDLE =>
            if TMS = '1' then
                next_state <= DSELECT;
            else
                next_state <= IDLE;
            end if;
        when DSELECT =>
            if TMS = '1' then
                next_state <= ISELECT;
            else
                next_state <= CAPTURE;
            end if;
        when ISELECT =>
            if TMS = '1' then
                next_state <= RESET;
            else
                next_state <= CAPTURE;
            end if;
        when CAPTURE =>
            if TMS = '1' then
                next_state <= EXIT1;
            else
                next_state <= SHIFT;
            end if;
        when SHIFT =>
            if TMS = '1' then
                next_state <= EXIT1;
            else
                next_state <= SHIFT;
            end if;
        when EXIT1 =>
            if TMS = '1' then
                next_state <= UPDATE;
            else
                next_state <= PAUSE;
            end if;
        when PAUSE =>
            if TMS = '1' then
                next_state <= EXIT2;
            else
                next_state <= PAUSE;
            end if;
```

```
            when EXIT2 =>
                if TMS = '1' then
                    next_state <= UPDATE;
                else
                    next_state <= SHIFT;
                end if;
            when UPDATE =>
                if TMS = '1' then
                    next_state <= DSELECT;
                else
                    next_state <= IDLE;
                end if;
        end case;
    end process;                -- NEXT_STATE_LOGIC

    process(TCK)                -- SHIFT_LOGIC
    begin
        if rising_edge(TCK) then
            if current_state = DSELECT then
                DR_selected <= '1';     -- DR selected for scan
            elsif current_state = ISELECT then
                DR_selected <= '0';     -- DR not selected, so IR selected
            elsif current_state = SHIFT then
                -- Shift data into selected register according to DR_selected
                if DR_selected = '1' then
                    DR <= TDI & DR(31 downto 1);
                else
                    IR <= TDI & IR(6 downto 1);
                end if;
            end if;
        end if;
    end process;                -- SHIFT_LOGIC

    process(DR, IR, DR_selected) -- OUTPUT_LOGIC
    begin
        -- Output the LSB of the selected register
        if DR_selected = '1' then
            TDO <= DR(0);
        else
            TDO <= IR(0);
        end if;
    end process;                -- OUTPUT_LOGIC

end behavioral;
```