Synthesis
From code to hardware

Design Methodology

An Introduction

—

Application Specific Integrated Circuits [ASICs] and Field Programmable Gate Arrays [FPGAs] are designed systematically to maximize the likelihood that a design will be correct and will be fabricated without fatal flaws. Design follow a "*design flow*" which specifies a sequence of major steps that will be taken to design, verify, synthesize and test a digital circuit. ASIC design flows involve several activities, from specification and design entry, to place-and-route and timing closure of the circuit in silicon. Timing closure is attained when all of the signal paths in the design satisfy the timing constraints imposed by the interface circuitry, the circuit's sequential elements and the system clock. Although the design flow appears to be linear, in practice is not. Various steps might be revisited as design errors are discovered, requirements change, or performance and designs constraints are violated.

Design flow for HDL-based ASICs

—

1. Design specification
2. Design partition
3. Design entry: Verilog/VHDL behavioral modeling
4. Simulation/functional verification
5. Design integration and verification
6. Presynthesis sign-off
7. Synthesize and map gate-level netlist
8. Postsynthesis design validation
9. Postsynthesis timing verification
10. Test generation and fault simulation
11. Cell placement, scan chain and clock tree insertion, cell routing
12. Verify physical and electrical design rules
13. Extract parasitics
14. Design sign-off

Design flow for HDL-based ASICs

—

The design flow begins with a written specification for the design. The specification document can be very elaborate statement of functionality, timing, silicon area, power consumption, testability, fault coverage, and other criteria that govern the design.

At a minimum, the specification describes the functional characteristics that are to be implemented in a design. Typically, state transition graphs, timing charts, and algorithmic-state machine [*ASM*] charts are used to describe sequential machines.

Design specification

—

In today’s methodologies for designing ASICs and FPGAs, large circuits are partitioned to form an *architecture*; a configuration of interacting functional units, such that each is described by a behavioral model of its functionality. The process by which a complex design is progressively partitioned into smaller and simpler functional units is called *top-down design* or *hierarchical design*. Hardware Description Languages [*HDLs*] support top-down design with mixed levels of abstraction by providing a common framework for partitioning, synthesizing and verifying large, complex systems. Parts of large designs can be linked together for verification of overall functionality and performance. The partitioned architecture consists of functional units that are simpler that the whole, and each can be described by an HDL-based model

Design Partition

—

Design entry means composing a language-based description of the design and storing it in an electronic format in a computer. Modern designs are described by HDLs [Velilog, VHDL,...] because it takes significantly less time to write a behavioral description and synthesize a gate-level realization of a large circuit that it does develop the gate-level realization by other means, such as *bottom-up* manual entry. This saves time that can be put to better use in other parts of the design cycle. The ease of writing, changing or substituting HDL descriptions encourages architectural exploration; moreover, a synthesis tool itself will find alternative realizations of the same functionality and generate reports describing the attributes of the design.
Synthesis tools create an optimal internal representation of a circuit before mapping the description into the target technology. The internal database at this stage is generic, which allows it to be mapped into a variety of technologies.

Design Entry

—

HDL-based designs are easier to debug than schematics. A behavioral description encapsulating complex functionality hides underlying gate-level detail, so there is less information to cope with in trying to isolate problems in the functionality of the design. Furthermore, if the behavioral description is functionally correct, it is a gold standard for subsequent gate-level realizations.
HDL-based designs incorporate documentation within the design by using descriptive names, by including comments to clarify intent and by explicitly specifies the functionality of the design. Since the language is standard, documentation of a design can be decoupled from a particular vendor's tools.

Behavioral modeling is the predominant descriptive style used by the industry; it describes the functionality of a design by specifying what the designed circuit will do, not how to build it in hardware. It specifies the input-output model of a logic circuit and suppresses details about physical, gate-level implementation.

Design Entry

—

Behavioral modeling encourages designers to:

1. Rapidly create a behavioral prototype of a design [without binding it to hardware details].
2. Verify its functionality.
3. Use a synthesis tool to optimize and map the design into a selected physical technology.

If the model has been written in a synthesis-ready style, the synthesis tool will remove redundant logic, perform tradeoffs between alternative architectures and/or multilevel equivalent circuits and ultimately achieve a design that is compatible with are or timing constraints. By focusing the designer’s attention on the functionality that is to be implemented rather than on individual logic gates and their interconnections, behavioral modeling provides the freedom to explore alternatives to a design before committing it to production.

Design Entry

—

The verification process is threefold, it includes:

1. Development of a test plan - A carefully documented test plan is developed to specify what functional features are to be tested and how they will be tested. A test plan identifies the stimulus generators, response monitors and the gold standard response against which the model will be tested.
2. Development of a testbench - The *testbench* is a HDL module in which the *unit under test* [UUT] has been instantiated, together with the pattern generators that are to be applied to the inputs of the model during simulation. The testbench is documented to identify the goals and sequential activity that will be observed during simulation. If a design is formed as an architecture of multiple modules, each must be verified separately, beginning with the lowest level of the design hierarchy, then the integrated design must be tested to verify that the modules interact correctly. In this case, the test plan must describe the functional features of each module and the process by which they will be tested

Simulation and functional verification

—

3. Test execution and model verification - The testbench is exercised according to the test plan and the response is verified against the original specification for the design. This step is intended to reveal errors in the design, confirm the syntax of the description, verify style conventions and eliminate barriers to synthesis. Verification of a model requires a systematic, thorough demonstration of its behavior. *There is no point in proceeding further into the design flow until the model has been verified.*

Simulation and functional verification

—

After each of the functional subunits of a partitioned design have been verified to have correct functionality, the architecture must be integrated and verified to have the correct functionality. This requires development of a separate testbench whose stimulus generators exercise the input-output functionality of the top level module, monitor port and bus activity across module boundaries and observe state activity in any embedded state machines. *This step in the design flow is crucial* and must be excited thoroughly to ensure that the design that is being signed off for synthesis is correct.

Design Integration and Verification

—

A demonstration of full functionality is to be provided by the testbench and any discrepancies between the functionality of the HDL behavioral model and the design specification must be resolved. *Sign-off* occurs after all known functional errors have been eliminated

Presynthesis sing-off

—

After all syntax and functional errors have been eliminated from the design and sign-off has occurred, a synthesis tool is used to create an optimal Boolean description and compose it in an available technology. In general, a synthesis tools removes redundant logic and seeks to reduce the area of the logic needed to complement the functionality and satisfy performance (speed) specifications. This step produces a *netlist of standard cells* or a database that will configura a target FPGA

Gate-level synthesis and technology mapping

—

Design validation compares the response of the synthesized gate-level description to the response of the behavioral model. This can be done by a testbench that instantiates both models and drives them with a common stimulus. The response can be monitored by software and/or by visual/graphical mens to see whether they have identical functionality. For synchronous designs that match must hold at the boundaries of the machine’s cycle-intermediate activity is of no consequence. If the functionality of the behavioral description and the synthesized realization do not match, painstaking work must be done to understand and resolve the discrepancy. Post-synthesis design validation can reveal software race conditions in the behavioral model that cause events to occur in a different clock cycle that expected

Postsynthesis design validation

—

Although the synthesis process is intended to produce a circuit that meets timing specification, the circuit's timing margins must be checked to verify that speeds are adequate on critical paths. This step is repeated after *parasitic extraction* [step 13] because synthesis tools do not accurately anticipate the effect of the capacitive delays inducted by interconnect metalization in the layout. Ultimately, these delays must be extracted from the properties of the materials and the geometric details of the fabrication masks. The extracted delays are used by a static timing analyzer to verify that the longest paths do not violate timing constraints. The circuit might have to be resynthesized or re-placed and rerouted to meet specifications. Resynthesis might require [1] transistor resizing [2] architectural modification/substitutions and [3] device substitution

Postsynthesis timing verification

—

After fabrication, integrated circuits must be tested to verify that they are free of defects and operate correctly. Contaminants in the clean-room environment can cause defects on the circuit and render it useless. In this step of the design flow a set of test vectors is applied to the circuit and response of the circuit is measured. Testing considers process-inducted faults, not design errors. Design errors should be detected before presynthesis sign-off. Testing is daunting, for an ASIC chip might have millions of transistors, but only a few hundred package pins that can be used to probe the internal circuits. The designer might have to embed additional, special circuits that will enable a tester to use only a few external pins to test the entire internal circuitry of the ASIC, either alone or on a printed circuit board.

Test generation and fault simulation

The patterns that are used to verify a behavioral model can be used to test the fabricated part that results from synthesis, but they might not be robust enough to detect a sufficiently high level of manufacturing defects. Combinational logic can be tested for faults exhaustively, but sequential machines present special challenges. Fault simulation questions whether the chips that come off the fabrication line can, in fact, be tested to verify that they operate correctly. Fault simulation is conducted to determine whether a set of test vectors will detect a set of faults. The result of fault simulation guide the use of software tools for generating additional test patterns. To eliminate the possibility that a part could be produced but not tested, test patterns are generated before the device is fabricated, to allow for possible changes in the design such as a scan path.

Test generation and fault simulation

—

The placement and routing step of the ASIC design flow arranges the cells on the die and connects their signal paths. In cell-based technology the individual cells are integrated to form a global mask that will be used to pattern the silicon wafer with gates. This step also might involve inserting a clock tree into the layout, to provide a skew free distribution of the clock signal to the sequential elements of the design. If a scan path is to be used, it will be inserted in this step too.

Placement and routing

—

The physical layout of a design must be checked to verify that constraints on material widths, overlaps and separation is satisfied. Electrical rules are checked to verify that fanout constraints are met and that signal integrity is not compromised by electrical crosstalk and power-grip drop. Noise levels are also checked to determine whether electrical transients are problematic. Power dissipation is modeled and analyzed in this step to verify that the generated by the chip will not damage the circuitry

Physical and electrical design rule checks

—

Parasitic capacitance induces by the layout is extracted by a software tool and then used to produce a more accurate verification of the electrical characteristics and timing performance of the design. The result of the extraction step are used to update the loading models that are used in timing calculations. Then the timing constraints are checked again to confirm that the design, as laid out, will function at the specified clock speed.

Parasitic extraction

—

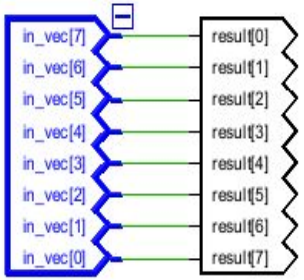
Final sign-off occurs after all of the design constraints have been satisfied and timing-closure has been achieved. The mask set is ready for fabrication. The description consists of the geometric data (usually in GDS-II format) that will determine the photomasking steps of the fabrication process. At this point significant resources have been expended to ensure that the fabricated chip will meet the specifications for its functionality and performance.

Design sign-off

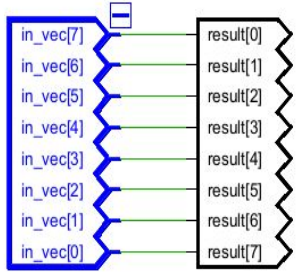
RTL and Synthesis

Examples for typical components

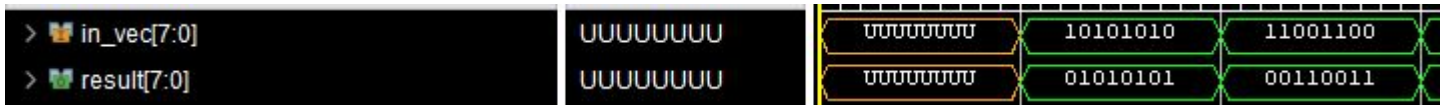
```vhdl
entity loops_1 is
  port (
    in_vec : in std_logic_vector(7 downto 0);
    result : out std_logic_vector(7 downto 0)
  );
end loops_1;

architecture rtl of loops_1 is
begin
   REVERSE_PROC : process(in_vec)
  begin
    for i in 0 to 7 loop
      result(i) <= in_vec(7 - i);
    end loop;
  end process;
end architecture;
```



RTL Schematic



Synthesized schematic

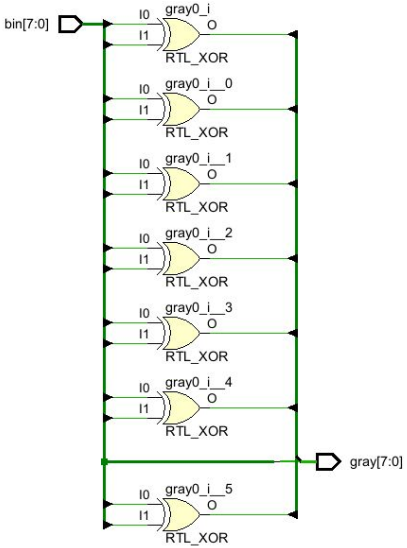


Simulation

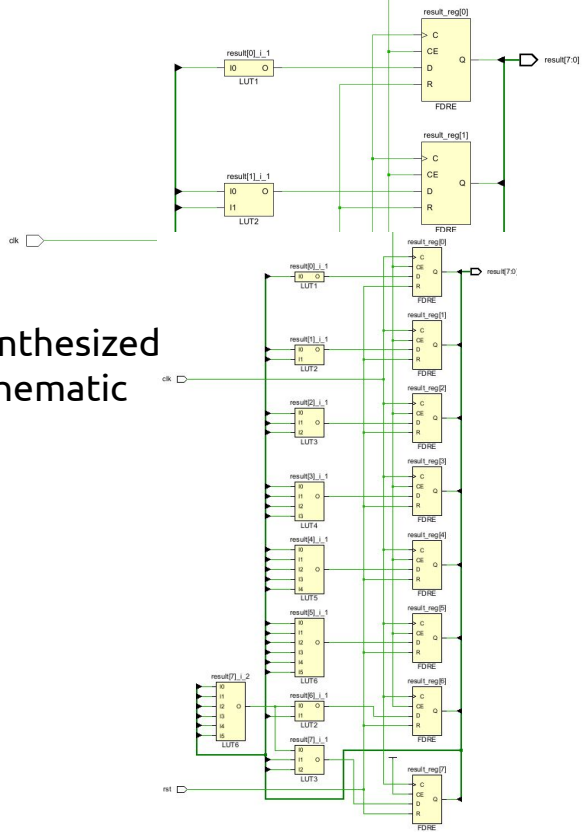
For loop vector reverser

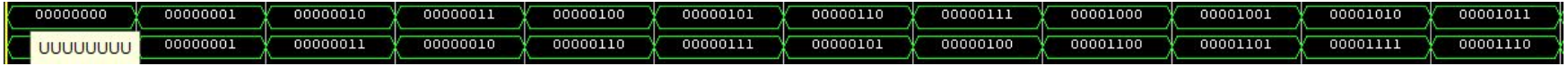
```
entity loops_2 is
  port (
    bin    : in std_logic_vector(7 downto 0);
    gray   : out std_logic_vector(7 downto 0)
  );
end loops_2;
architecture rtl of loops_2 is
begin
  GRAY_PROC : process(bin) is
  begin
    gray(7) <= bin(7);
    for i in 6 downto 0 loop
      gray(i) <= bin(i + 1) xor bin(i);
    end loop;
  end process;
end architecture;
```

RTL Schematic



Synthesized schematic



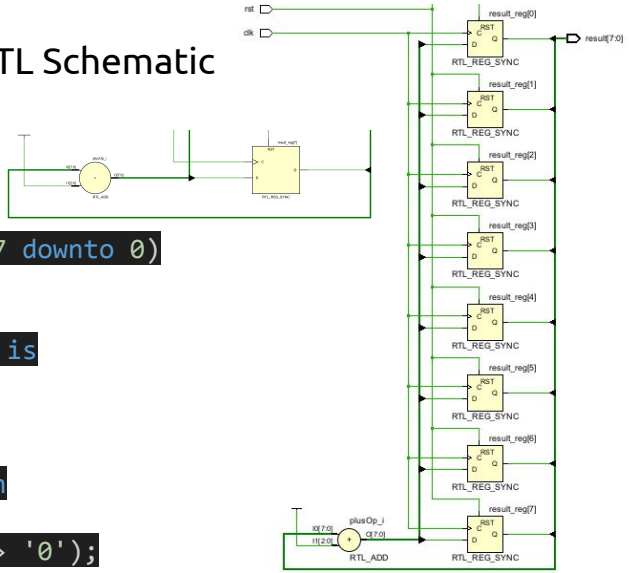


For loop gray converter

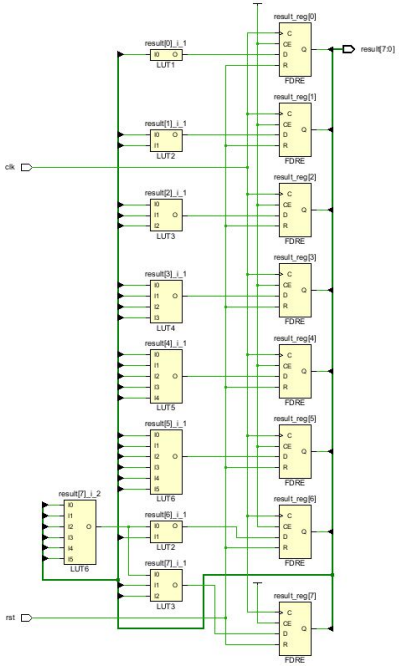
Simulation

RTL Schematic

```
entity loops_3 is
  port (
    clk    : in std_logic;
    rst    : in std_logic;
    result : out unsigned(7 downto 0)
  );
end loops_3;
architecture rtl of loops_3 is
begin
  SUM_PROC : process(clk)
  begin
    if rising_edge(clk) then
      if rst = '1' then
        result <= (others => '0');
      else
        -- It just produces an adder that adds 7 to result
        for i in 0 to 7 loop   -- Same as:
          result <= result + i;-- result <= result + 7;
        end loop;
            end if;
    end if;
  end process;
```



Synthesized schematic

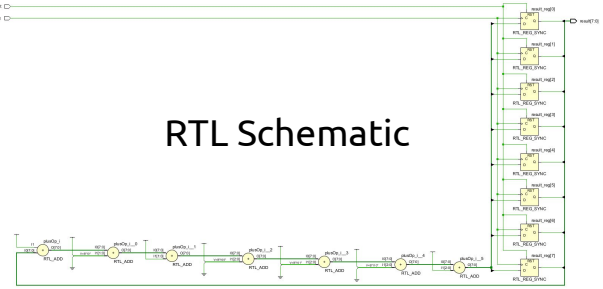


For loop

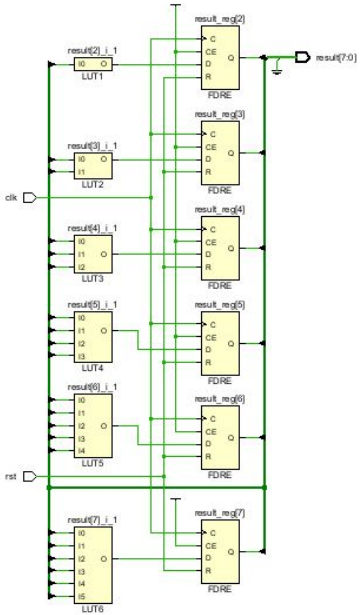
Simulation

```vhdl
entity loops_4 is
  port (
    clk      : in std_logic;
    rst      : in std_logic;
    result   : out unsigned(7 downto 0)
  );
end loops_4;
architecture rtl of loops_4 is
begin
  SUM_PROC : process(clk)
    variable tmp : unsigned(7 downto 0);
  begin
    if rising_edge(clk) then
      if rst = '1' then
        result <= (others => '0');
      else
        tmp := result;
        for i in 0 to 7 loop
          tmp := tmp + i;
        end loop;
        result <= tmp;
      end if;
    end if;
  end process;
end architecture;
```

RTL Schematic



Synthesized schematic [optimized]

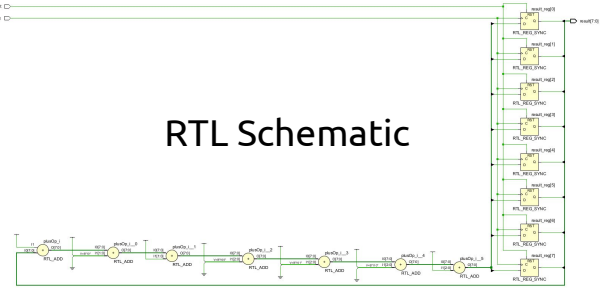


Simulation

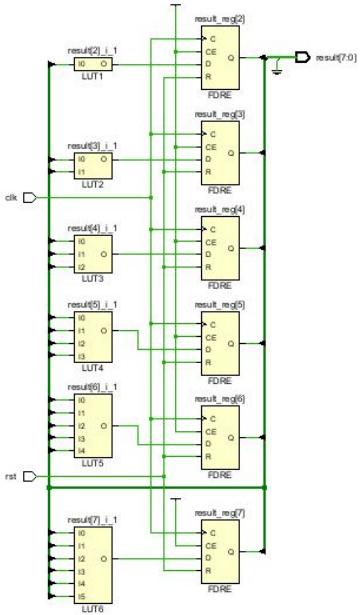
For loop adder chain

```vhdl
entity loops_5 is
  port (
    clk    : in std_logic;
    rst    : in std_logic;
    result : out unsigned(7 downto 0)
  );
end loops_5;
architecture rtl of loops_5 is
begin
  SUM_PROC : process(clk)
    variable tmp  : unsigned(7 downto 0);
    variable i    : integer;
  begin
    if rising_edge(clk) then
      if rst = '1' then
        result <= (others => '0');
      else
        tmp := result; i := 0;
        while i < 8 loop
          tmp := tmp + i;
          i := i + 1;
        end loop;
        result <= tmp;
      end if;
    end if;
  end process;
end architecture;
```

RTL Schematic



Synthesized schematic [optimized]

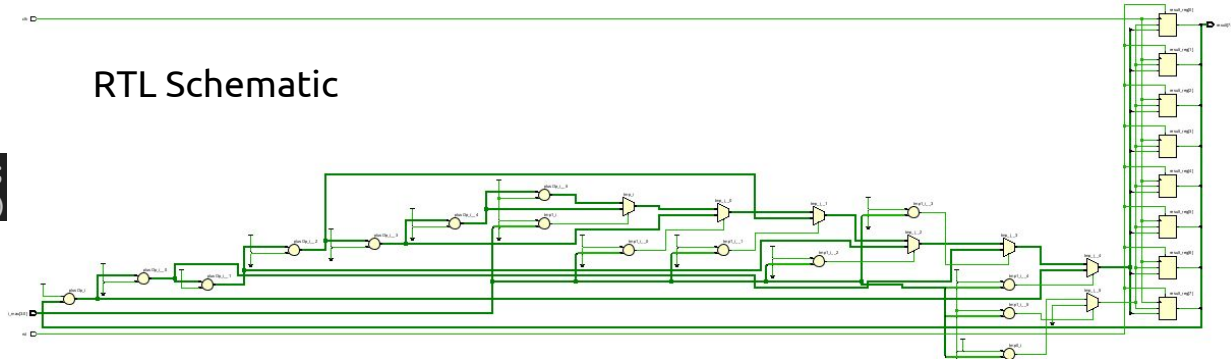


Simulation

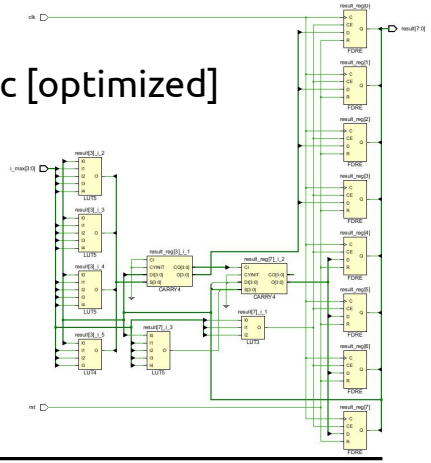
While loop adder chain

```vhdl
entity loops_6 is
  port (
    clk      : in std_logic;
    rst      : in std_logic;
    i_max    : in integer range 0 to 8;
    result   : out unsigned(7 downto 0)
  );
end loops_6;
architecture rtl of loops_6 is
begin
  SUM_PROC : process(clk)
    variable tmp  : unsigned(7 downto 0);
    variable i    : integer;
  begin
    if rising_edge(clk) then
      if rst = '1' then
        result <= (others => '0');
      else
        tmp := result; i := 0;
        while i < i_max loop
          tmp := tmp + i;
          i := i + 1;
        end loop;
        result <= tmp;
      end if;
    end if;
  end process;
end architecture;
```

RTL Schematic



Synthesized schematic [optimized]

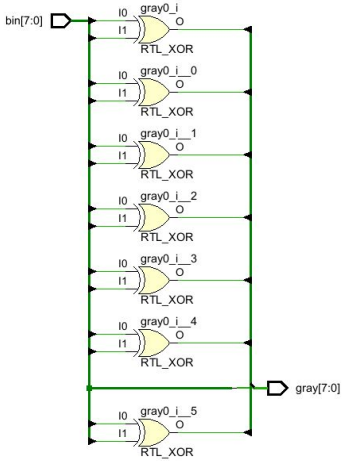


Simulation

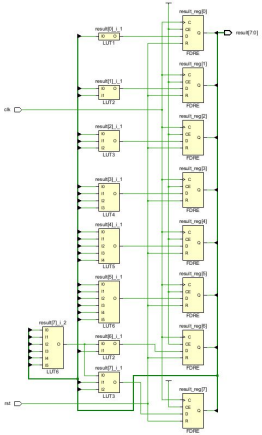
Loop unrolling in VHDL

```
entity loops_7 is
  port (
    bin    : in std_logic_vector(7 downto 0);
    gray   : out std_logic_vector(7 downto 0)
  );
end loops_7;
architecture rtl of loops_7 is
begin
  gray(7) <= bin(7);
  GRAY_GENERATOR : for i in 6 downto 0 generate
  begin
    gray(i) <= bin(i + 1) xor bin(i);
  end generate;
```

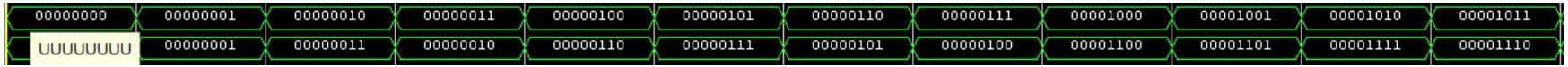
RTL Schematic



Synthesized schematic [optimized]



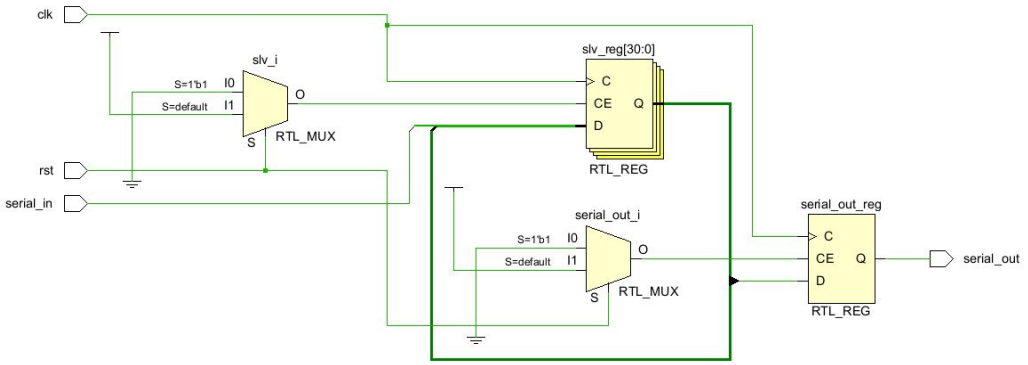
Simulation



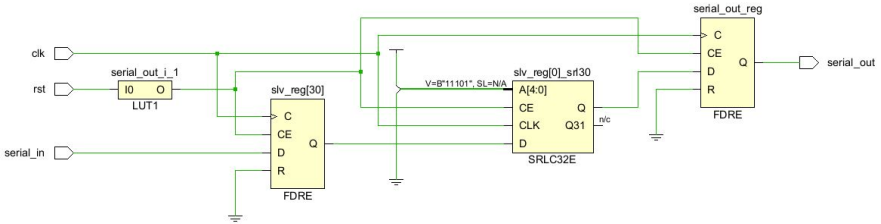
Generate loops

```vhdl
entity sreg_8 is
  port (
    clk          : in std_logic;
    rst          : in std_logic;
    serial_in    : in std_logic;
    serial_out   : out std_logic
  );
end sreg_8;
architecture rtl of sreg_8 is
  constant sreg_length : integer := 32;
  signal slv : std_logic_vector(sreg_length - 2 downto 0);
begin
  SREG_PROC : process(clk)
  begin
    if rising_edge(clk) then
      if rst = '1' then
        -- slv <= (others => '0');
      else
        slv <= serial_in & slv(slv'high downto 1);
        serial_out <= slv(0);
      end if;
    end if;
  end process;
```

RTL Schematic



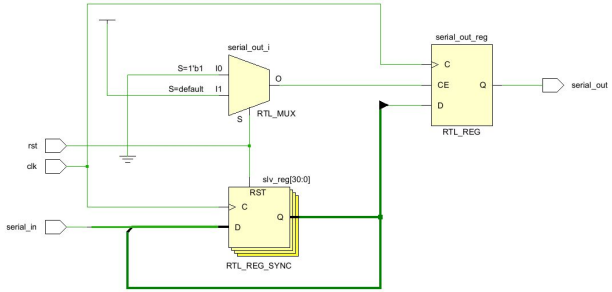
Synthesized schematic



Simulation

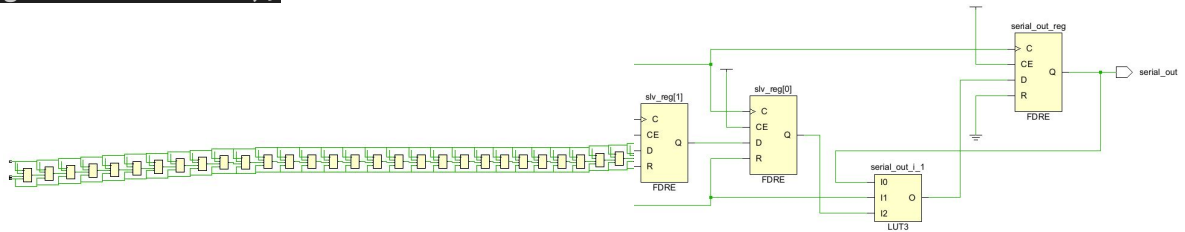
Inferring shift register LUTs [SRLs]

```vhdl
entity sreg_8 is
  port (
    clk         : in std_logic;
    rst         : in std_logic;
    serial_in   : in std_logic;
    serial_out  : out std_logic
  );
end sreg_8;
architecture rtl of sreg_8 is
  constant sreg_length : integer := 32;
  signal slv : std_logic_vector(sreg_length - 2 downto 0);
begin
  SREG_PROC : process(clk)
  begin
    if rising_edge(clk) then
      if rst = '1' then
        slv <= (others => '0');
      else
        slv <= serial_in & slv(slv'high downto 1);
        serial_out <= slv(0);
      end if;
    end if;
  end process;
```



RTL Schematic

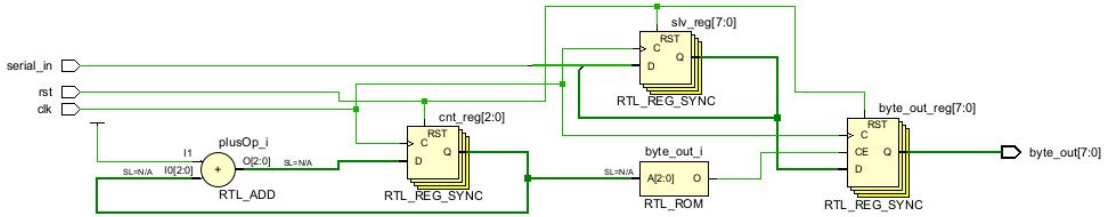
Synthesized schematic



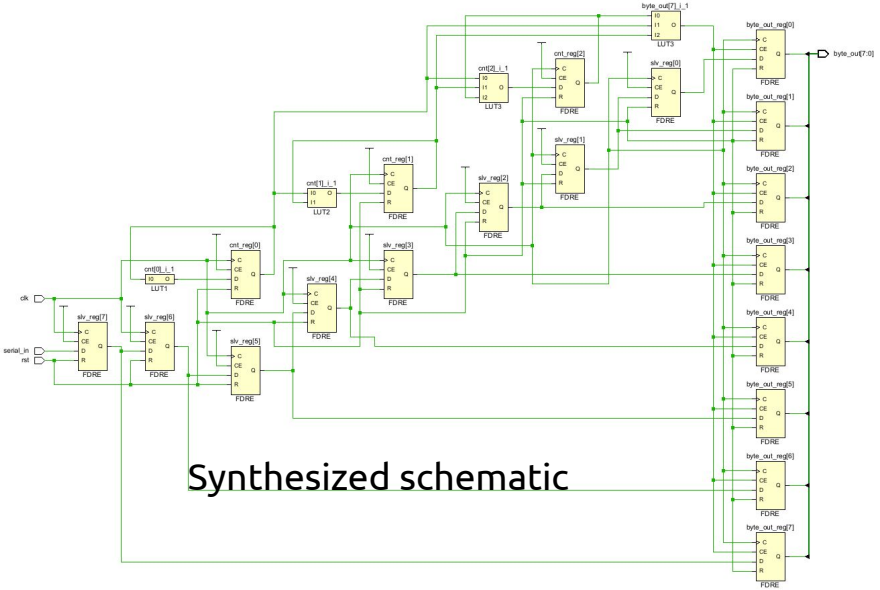
Simulation

Inferring shift register LUTs [SRLs]

```vhdl
entity sreg_9 is
  port (
    clk       : in std_logic;
    rst       : in std_logic;
    serial_in : in std_logic;
    byte_out  : out std_logic_vector(7 downto 0)
  );
end sreg_9;
architecture rtl of sreg_9 is
  signal slv : std_logic_vector(7 downto 0);
  signal cnt : unsigned(2 downto 0);
begin
  SREG_PROC : process(clk)
  begin
    if rising_edge(clk) then
      if rst = '1' then
        byte_out  <= (others => '0');
        slv       <= (others => '0');
        cnt       <= (others => '0');
      else
        slv <= serial_in & slv(slv'high downto 1);
        cnt <= cnt + 1;
        if cnt = 0 then
          byte_out <= slv;
        end if;
      end if;
    end if;
  end process;
end architecture;
```



RTL Schematic

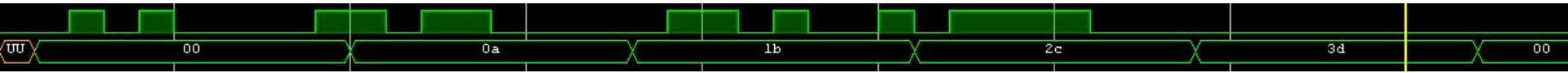


Synthesized schematic

Inferring shift register LUTs [SRLs]
8 bit serializer

```
SEQUENCER_PROC : process
  constant test_pattern : std_logic_vector(31 downto 0) := x"3D2C1B0A";
begin
  wait for 10 ns;
  rst <= '0';
  for i in 0 to 31 loop
    serial_in <= test_pattern(i);
    wait for clk_period * 1;
  end loop;
  wait;
end process;
```

Simulation

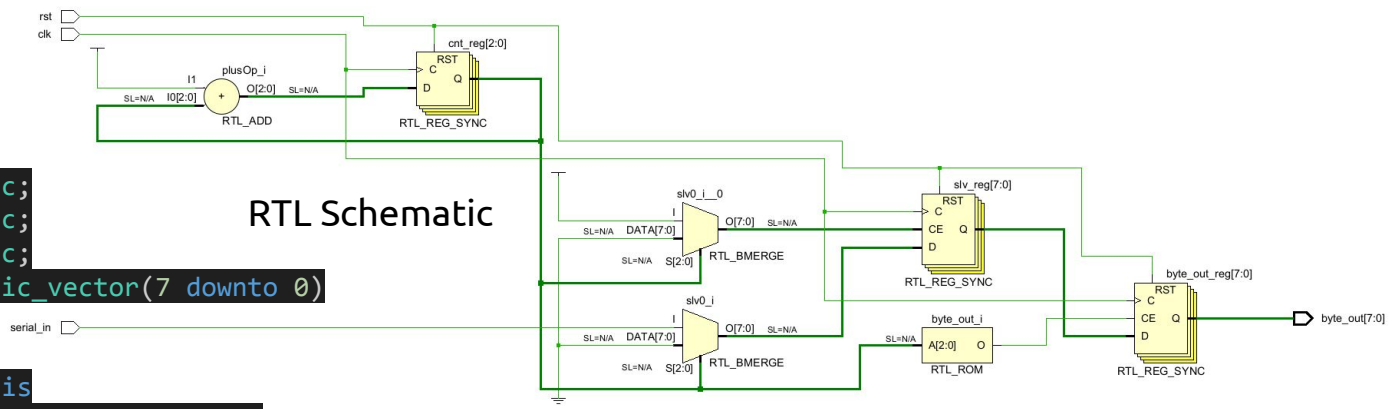


Inferring shift register LUTs [SRLs]

8 bit serializer

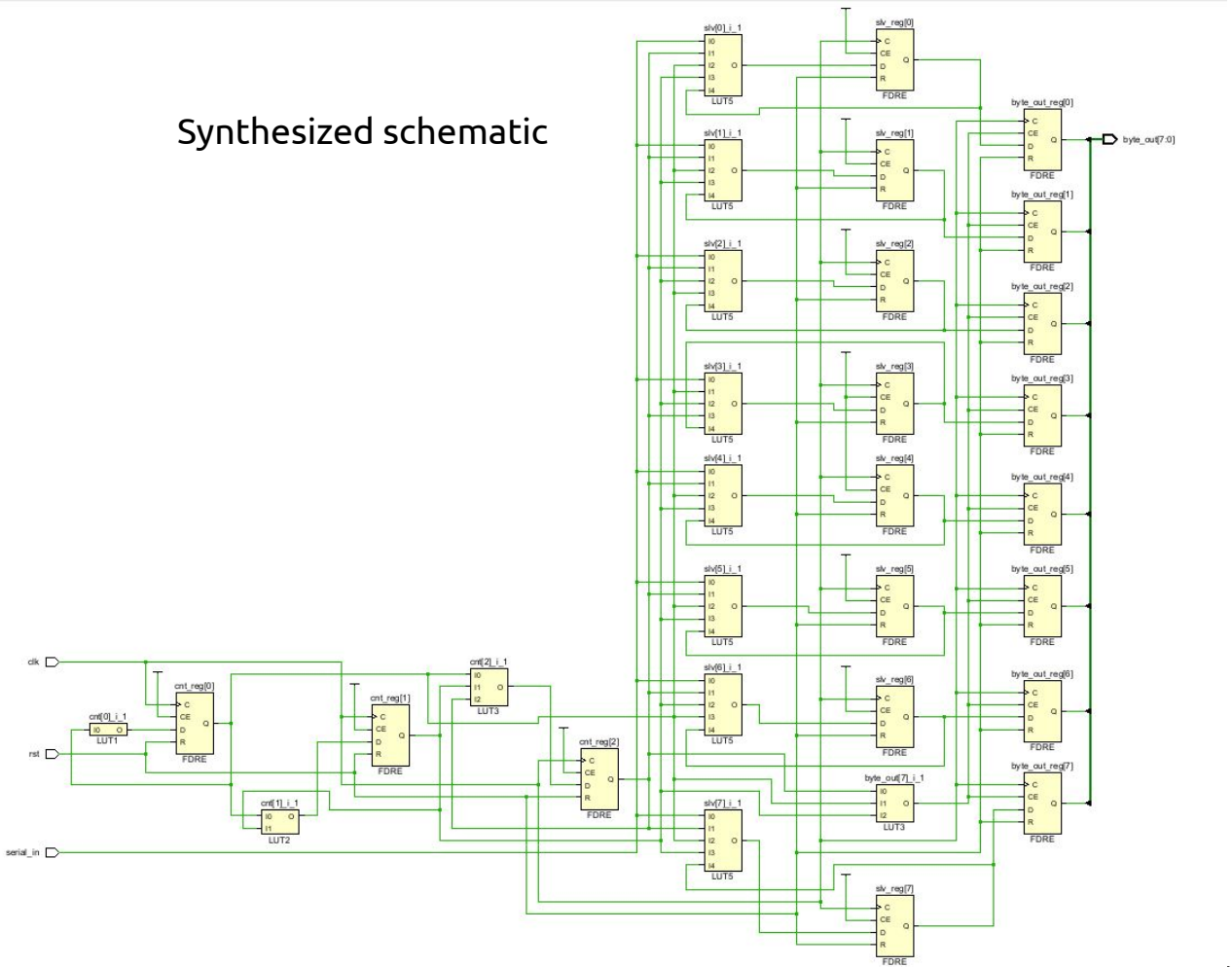
```
entity mux_10 is
  port (
    clk       : in std_logic;
    rst       : in std_logic;
    serial_in : in std_logic;
    byte_out  : out std_logic_vector(7 downto 0)
  );
end mux_10;
architecture rtl of mux_10 is
  signal slv : std_logic_vector(7 downto 0);
  signal cnt : unsigned(2 downto 0);
begin
  MUX_PROC : process(clk)
  begin
    if rising_edge(clk) then
      if rst = '1' then
        byte_out  <= (others => '0');
        slv       <= (others => '0');
        cnt       <= (others => '0');
      else
        slv(to_integer(cnt)) <= serial_in;
        cnt <= cnt + 1;
        if cnt = 0 then
          byte_out <= slv;
        end if;
      end if;
    end if;
  end process;
end architecture;
```

RTL Schematic



Multiplexers [MUX]

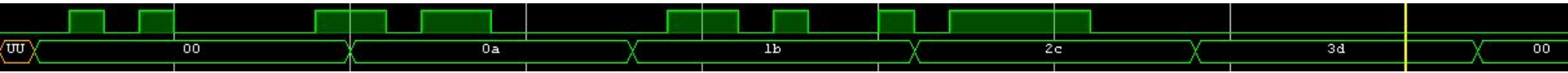
Synthesized schematic



Multiplexers [MUX]

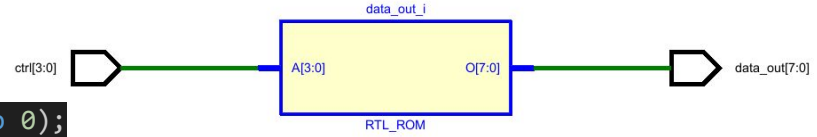
```
SEQUENCER_PROC : process
  constant test_pattern : std_logic_vector(31 downto 0) := x"3D2C1B0A";
begin
  wait for 10 ns;
  rst <= '0';
  for i in 0 to 31 loop
    serial_in <= test_pattern(i);
    wait for clk_period * 1;
  end loop;
  wait;
end process;
```

Simulation

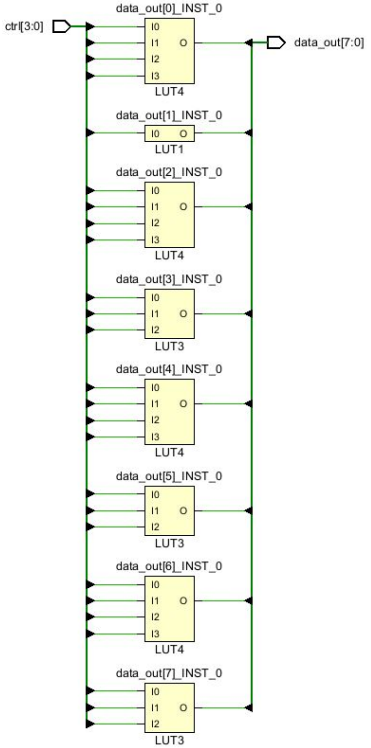


Multiplexers [MUX]

```
entity rom_11 is
  port (
    ctrl      : in std_logic_vector (3 downto 0);
    data_out  : out std_logic_vector (7 downto 0)
  );
end rom_11;
architecture rtl of rom_11 is
begin
  process(ctrl)
  begin
    case ctrl is
      when "0000" =>
        data_out <= "00000011";
      when "0001" =>
        data_out <= "00000111";
      when "0010" =>
        data_out <= "00001111";
      when "0011" =>
        data_out <= "00011111";
      when "0100" =>
        data_out <= "00111111";
      when "0101" =>
        data_out <= "01111111";
      when "0110" =>
        data_out <= "11111111";
      when "0111" =>
        data_out <= "11111110";
      when "1000" =>
        data_out <= "11111100";
```



RTL Schematic



Synthesized schematic

Simulation

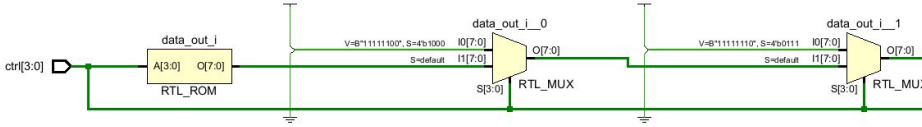
Case-When

```
SEQUENCER_PROC : process
begin
  wait for 10 ns;
  ctrl <= "0000";
  wait for 10 ns;
  ctrl <= "0001";
  wait for 10 ns;
  ctrl <= "0010";
  wait for 10 ns;
  ctrl <= "0011";
  wait for 10 ns;
  ctrl <= "0100";
  wait for 10 ns;
  ctrl <= "0101";
  wait for 10 ns;
  ctrl <= "0110";
  wait for 10 ns;
  ctrl <= "0111";
  wait for 10 ns;
  ctrl <= "1000";
  wait for 10 ns;
  ctrl <= "1001";
  wait for 10 ns;
  -- Unspecified value
  ctrl <= "1011";
  wait for 10 ns;
  wait;
end process;
```

Simulation

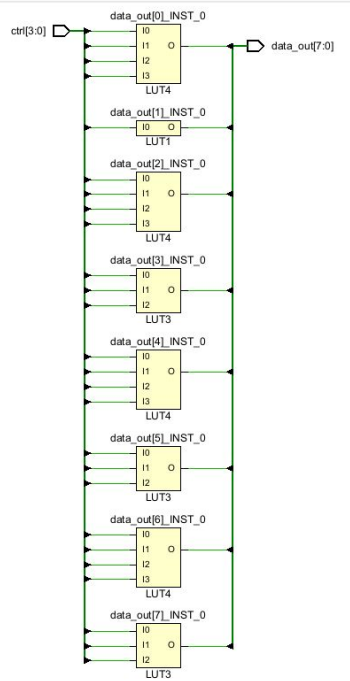
Case-When

RTL Schematic




```
entity mux_12 is
  port (
    ctrl      : in std_logic_vector (3 downto 0);
    data_out  : out std_logic_vector (7 downto 0)
  );
end mux_12;
architecture rtl of mux_12 is
begin
  process(ctrl)
  begin
    if ctrl = "0000" then
      data_out <= "00000011";
    elsif ctrl = "0001" then
      data_out <= "00000111";
    elsif ctrl = "0010" then
      data_out <= "00001111";
    elsif ctrl = "0011" then
      data_out <= "00011111";
    elsif ctrl = "0100" then
      data_out <= "00111111";
    elsif ctrl = "0101" then
      data_out <= "01111111";
    elsif ctrl = "0110" then
      data_out <= "11111111";
    elsif ctrl = "0111" then
```

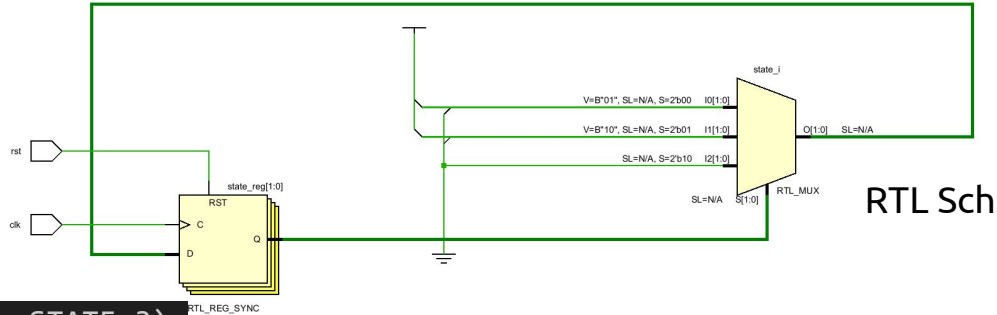
Synthesized Schematic



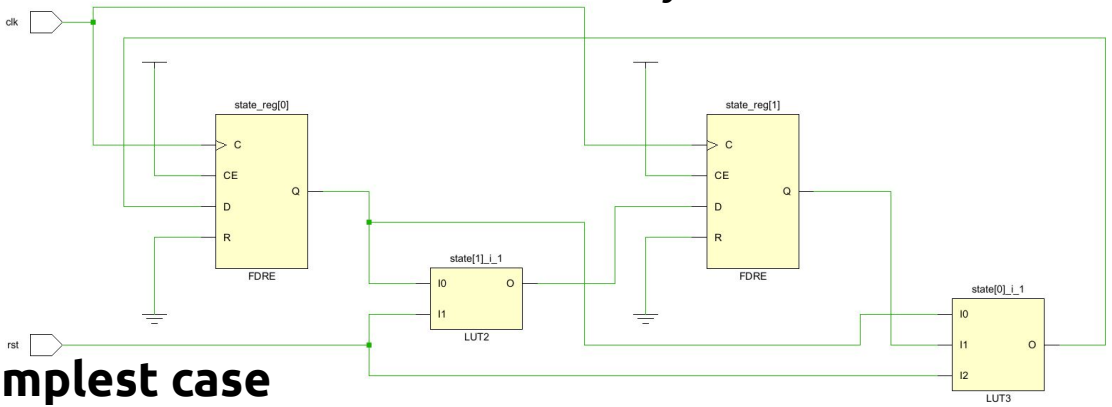
If-elsif-else // Multiplexers [MUX]

```vhdl
entity fsm_13 is
  port (
    clk : in std_logic;
    rst : in std_logic
  );
end fsm_13;
architecture rtl of fsm_13 is
  type state_type is (STATE_1, STATE_2, STATE_3);
  signal state : state_type;
  -- Keep state signal and do not optimize
  attribute keep : string;
  attribute keep of state : signal is "true";
begin
  FSM_PROC : process(clk)
  begin
    if rising_edge(clk) then
      if rst = '1' then
        state <= STATE_1;
      else
        case state is
          when STATE_1 =>
            state <= STATE_2;
          when STATE_2 =>
            state <= STATE_3;
          when STATE_3 =>
```

RTL Schematic



Synthesized schematic

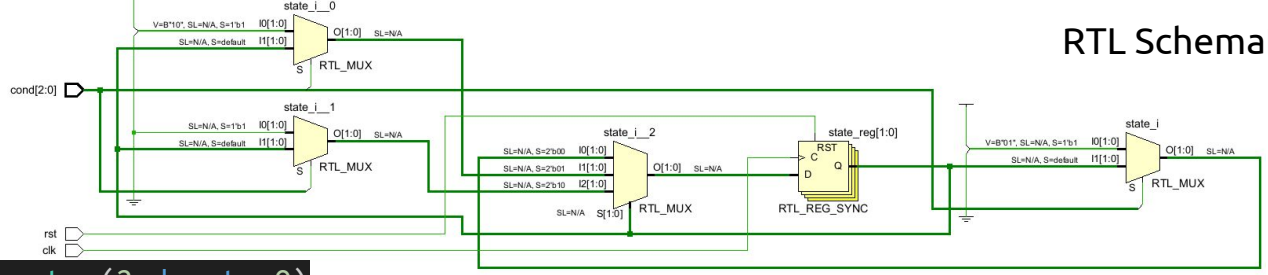


FSM - Simplest case

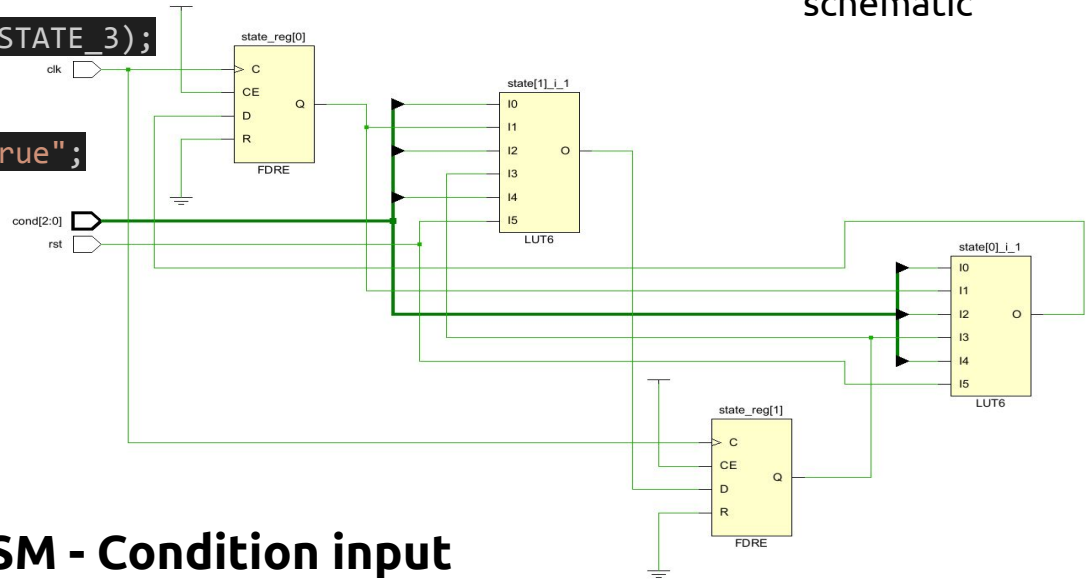
FSM - Condition input

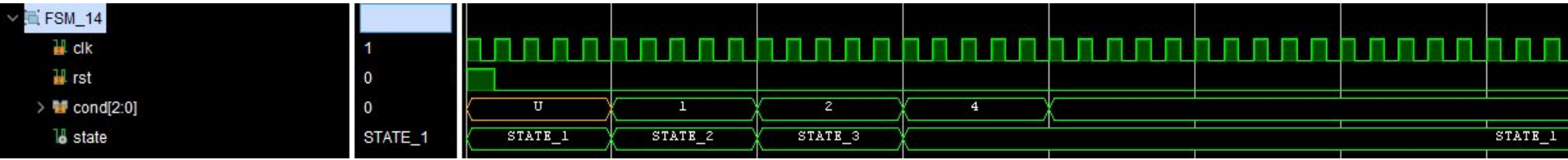
```
entity fsm_14 is
  port (
    clk   : in std_logic;
    rst   : in std_logic;
    cond  : in std_logic_vector(2 downto 0)
  );
end fsm_14;
architecture rtl of fsm_14 is
  type state_type is (STATE_1, STATE_2, STATE_3);
  signal state : state_type;
  attribute keep : string;
  attribute keep of state : signal is "true";
begin
  FSM_PROC : process(clk)
  begin
    if rising_edge(clk) then
      if rst = '1' then
        state <= STATE_1;
      else
        case state is
          when STATE_1 =>
            if cond(0) = '1' then
              state <= STATE_2;
            end if;
          when STATE_2 =>
```

RTL Schematic



Synthesized schematic





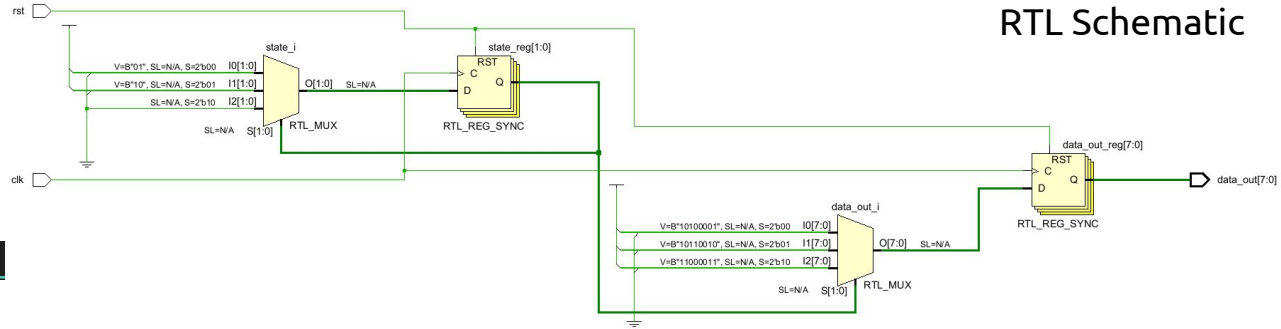
Simulation

FSM - Condition input

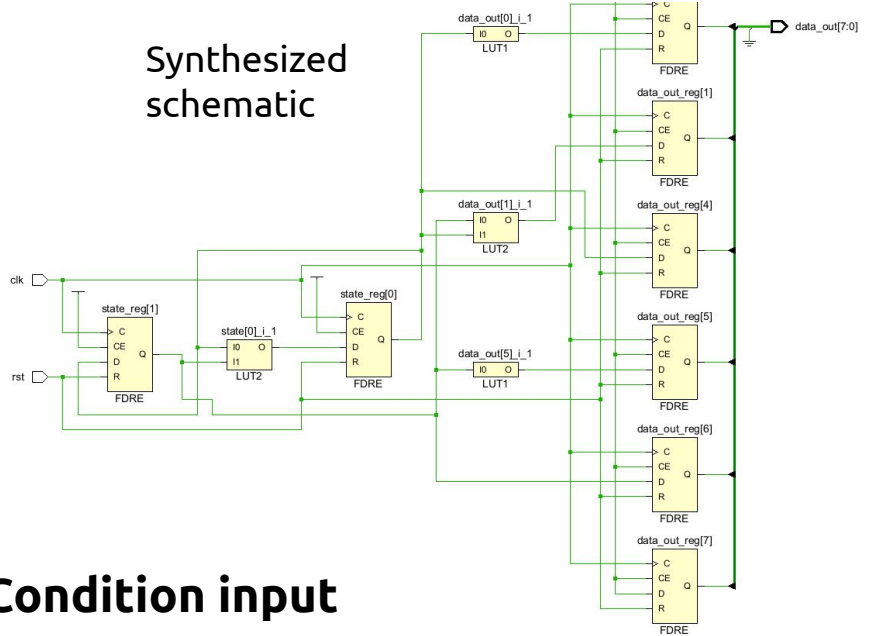
FSM - Condition input

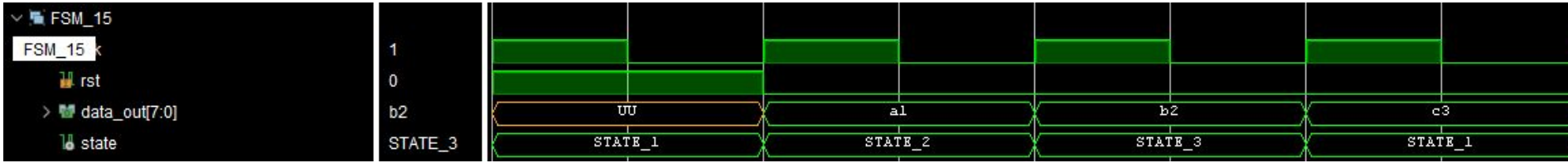
```
entity fsm_15 is
  port (
    clk       : in std_logic;
    rst       : in std_logic;
    data_out  : out std_logic
  );
end fsm_15;
architecture rtl of fsm_15 is
  type state_type is (STATE_1, STATE_2, STATE_3
  signal state : state_type;
  -- Keep state signal and do not optimize
  attribute keep : string;
  attribute keep of state : signal is "true";
begin
  FSM_PROC : process(clk)
  begin
    if rising_edge(clk) then
      if rst = '1' then
        data_out  <= (others => '0');
        state     <= STATE_1;
      else
        case state is
          when STATE_1 =>
            data_out  <= x"A1";
            state     <= STATE_2;
```

RTL Schematic



Synthesized schematic





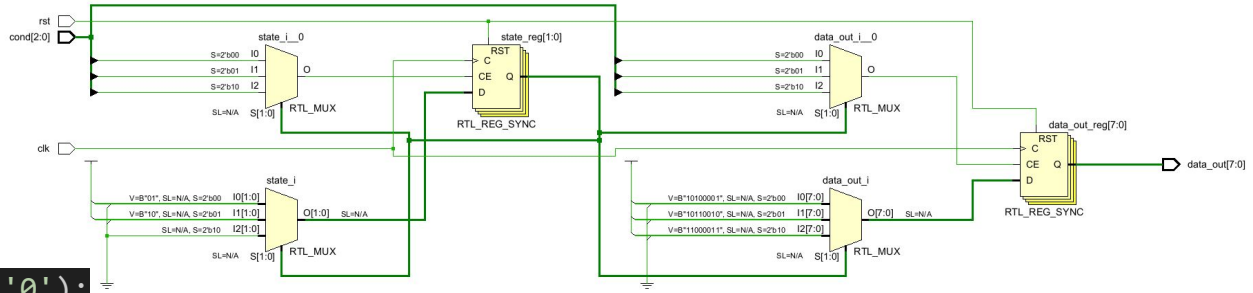
Simulation

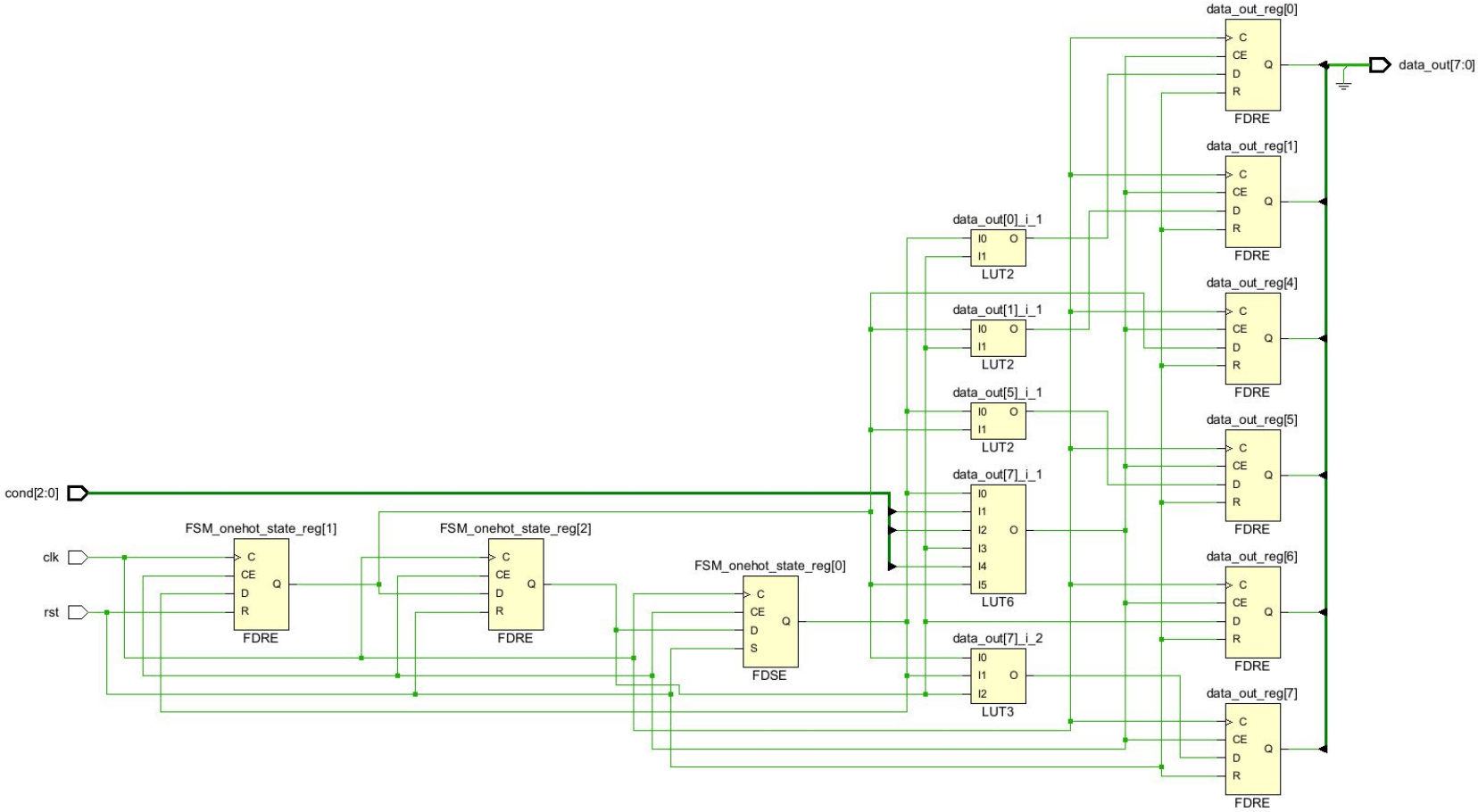
FSM - Condition input

FSM - Condition input and output

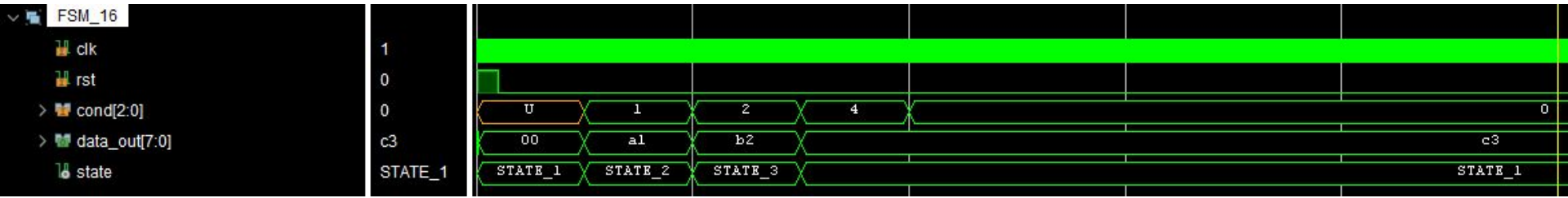
```
entity fsm_16 is
  port (
    clk       : in std_logic;
    rst       : in std_logic;
    cond      : in std_logic_vector(2 downto 0);
    data_out  : out std_logic_vector(7 downto 0)
  );
end fsm_16;
architecture rtl of fsm_16 is
  type state_type is (STATE_1, STATE_2, STATE_3);
  signal state : state_type;
begin
  FSM_PROC : process(clk)
  begin
    if rising_edge(clk) then
      if rst = '1' then
        state <= STATE_1;
        data_out <= (others => '0');
      else
        case state is
          when STATE_1 =>
            if cond(0) = '1' then
              state     <= STATE_2;
              data_out  <= x"A1";
            end if;
```

RTL Schematic





FSM - Synthesized schematic

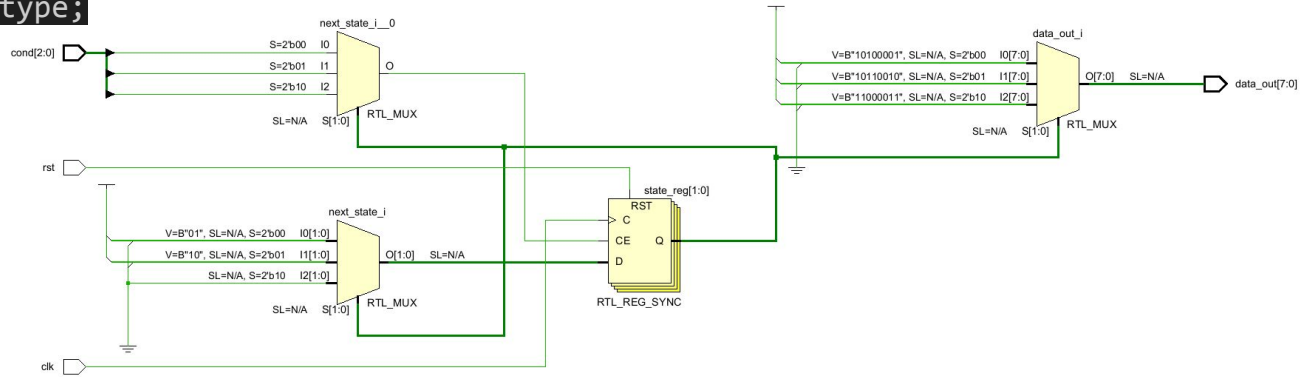


Simulation

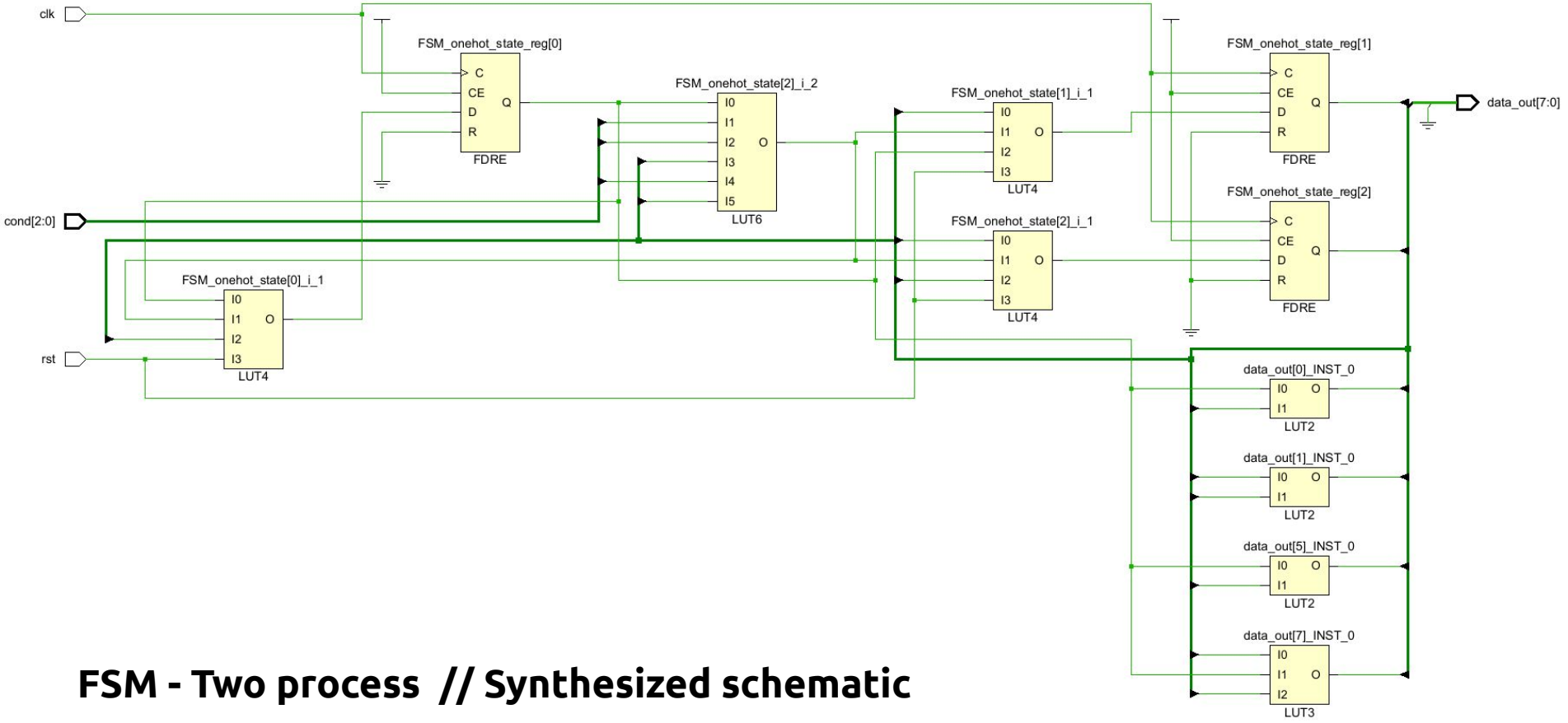
FSM - Condition input and output

```
entity fsm_17 is
  port (
    clk       : in std_logic;
    rst       : in std_logic;
    cond      : in std_logic_vector(2 downto 0);
    data_out  : out std_logic_vector(7 downto 0)
  );
end fsm_17;
architecture rtl of fsm_17 is
  type state_type is (STATE_1, STATE_2, STATE_3);
  signal state      : state_type;
  signal next_state : state_type;
begin
  SYNC_PROC : process(clk)
  begin
    if rising_edge(clk) then
      if rst = '1' then
        state <= STATE_1;
      else
        state <= next_state;
      end if;
    end if;
  end process;
  COMB_PROC : process(all)
  begin
```

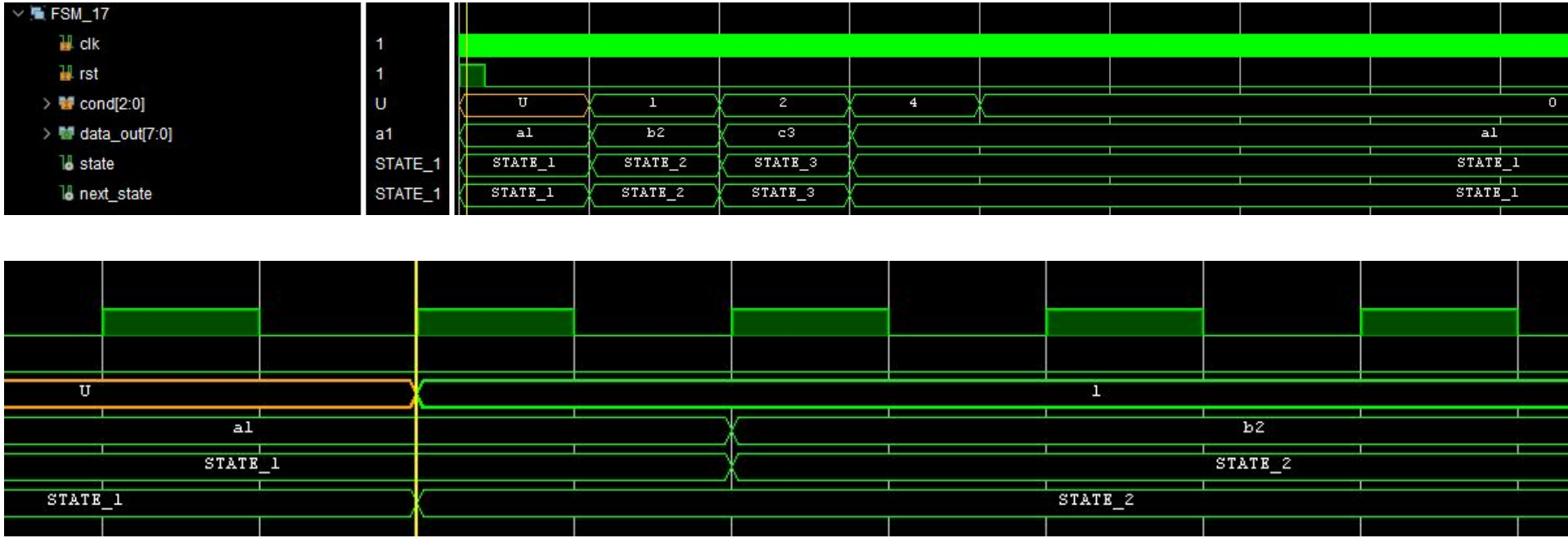
RTL Schematic



FSM - Two process



FSM - Two process // Synthesized schematic

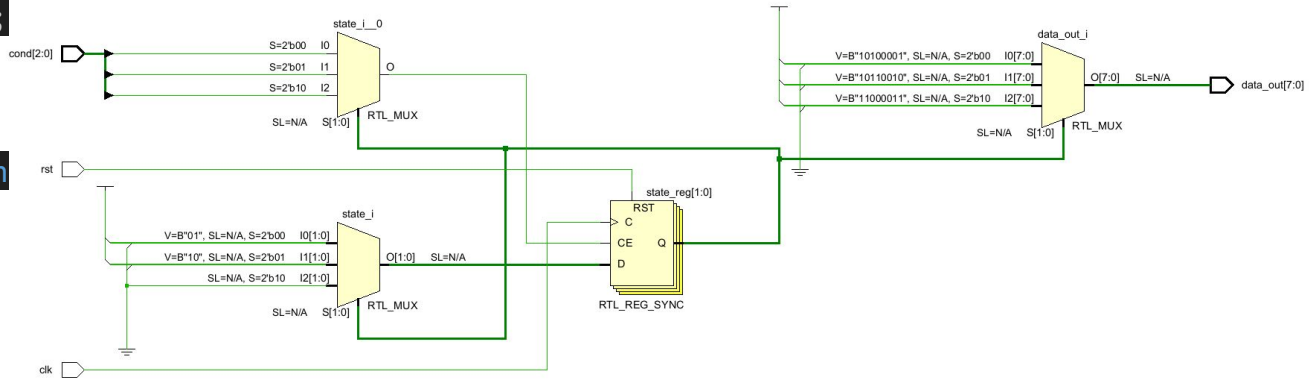


Simulation

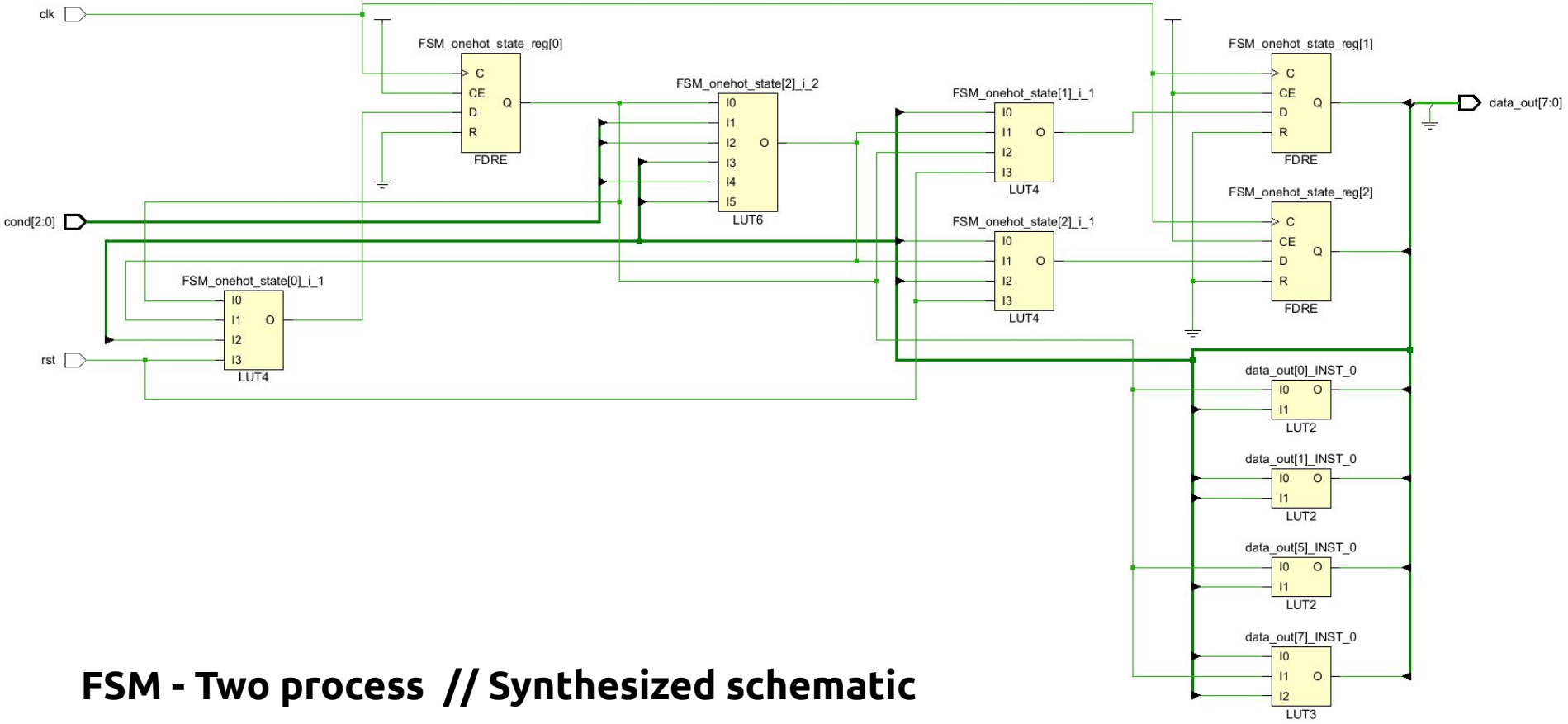
FSM - Two process

```
entity fsm_18 is
  port (
    clk       : in std_logic;
    rst       : in std_logic;
    cond      : in std_logic_vector(2 downto 0);
    data_out  : out std_logic_vector(7 downto 0)
  );
end fsm_18;
architecture rtl of fsm_18 is
  type state_type is (STATE_1, STATE_2, STATE_3);
  signal state : state_type;
begin
  SYNC_PROC : process(clk)
  begin
    if rising_edge(clk) then
      if rst = '1' then
        state <= STATE_1;
      else
        case state is
          when STATE_1 =>
            if cond(0) = '1' then
              state <= STATE_2;
            end if;
          when STATE_2 =>
            if cond(1) = '1' then
```

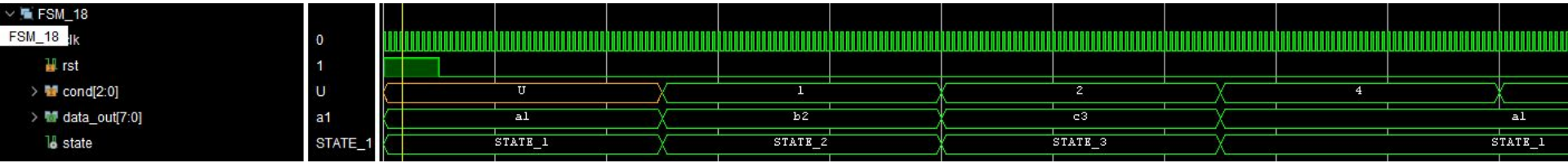
RTL Schematic



FSM - Two process



FSM - Two process // Synthesized schematic

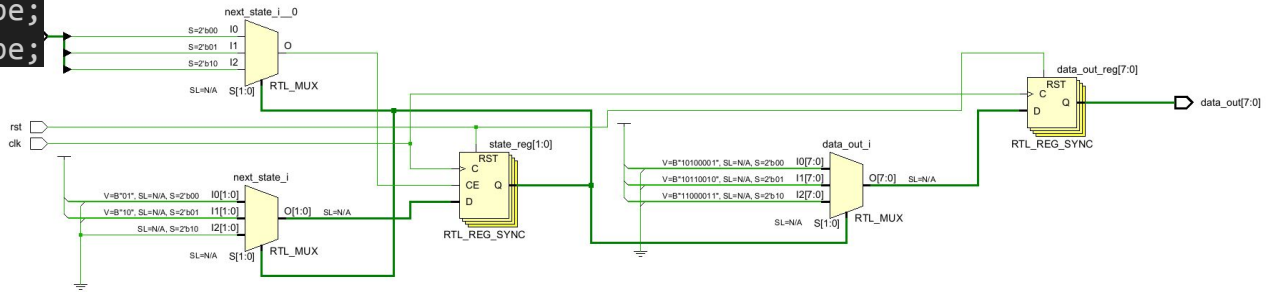


Simulation

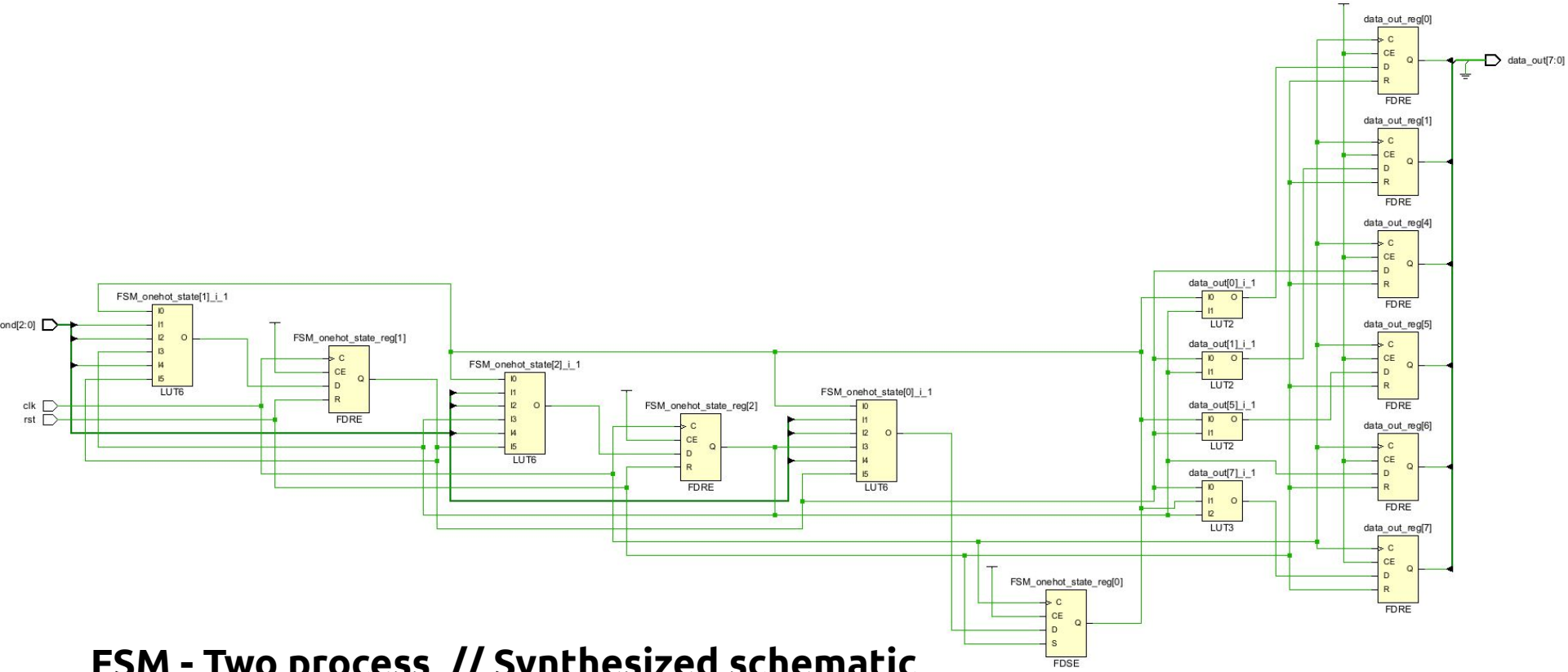
FSM - Two process

```
entity fsm_19 is
  port (
    clk       : in std_logic;
    rst       : in std_logic;
    cond      : in std_logic_vector(2 downto 0);
    data_out  : out std_logic_vector(7 downto 0)
  );
end fsm_19;
architecture rtl of fsm_19 is
  type state_type is (STATE_1, STATE_2, STATE_3);
  signal state      : state_type;
  signal next_state : state_type;
begin
  SYNC_PROC : process(clk)
  begin
    if rising_edge(clk) then
      if rst = '1' then
        state     <= STATE_1;
        data_out  <= (others => '0');
       else
        state <= next_state;
        case state is
          when STATE_1 =>
            data_out <= x"A1";
          when STATE_2 =>
```

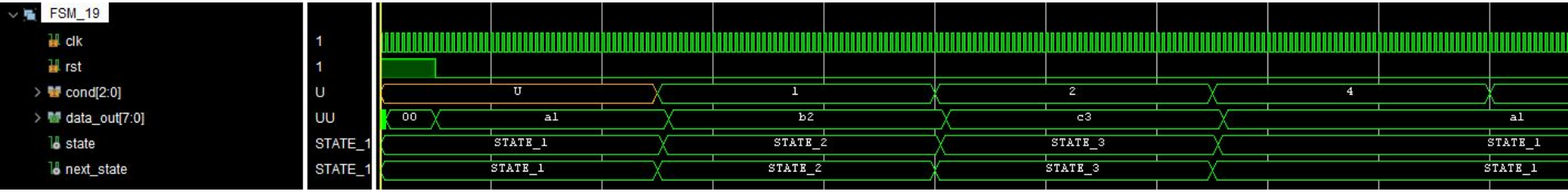
RTL Schematic

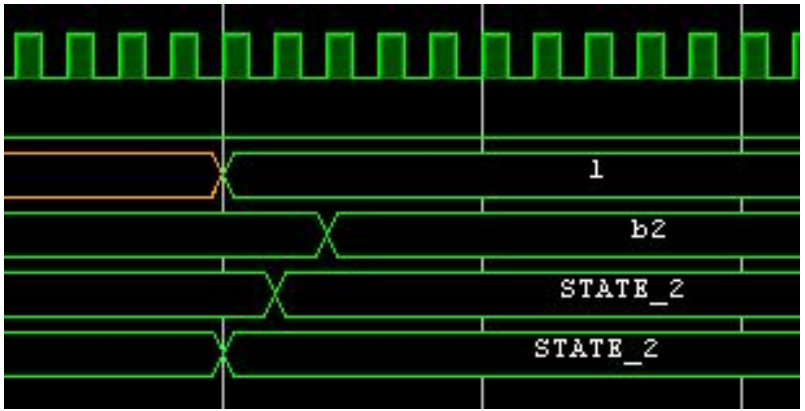


FSM - Two process



FSM - Two process // Synthesized schematic



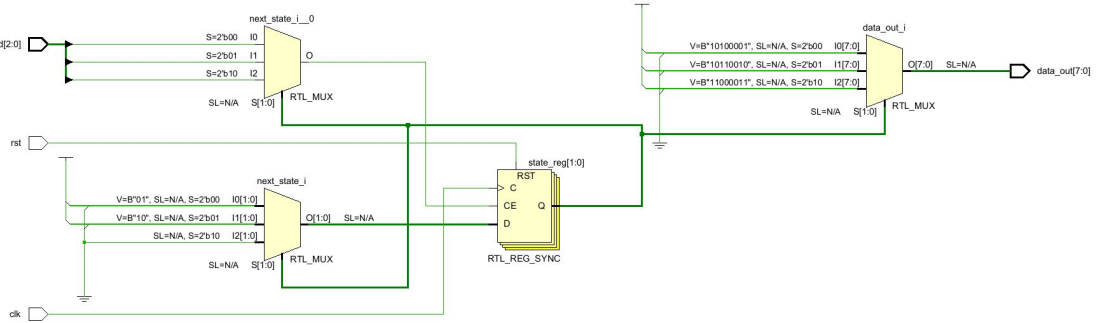


Simulation

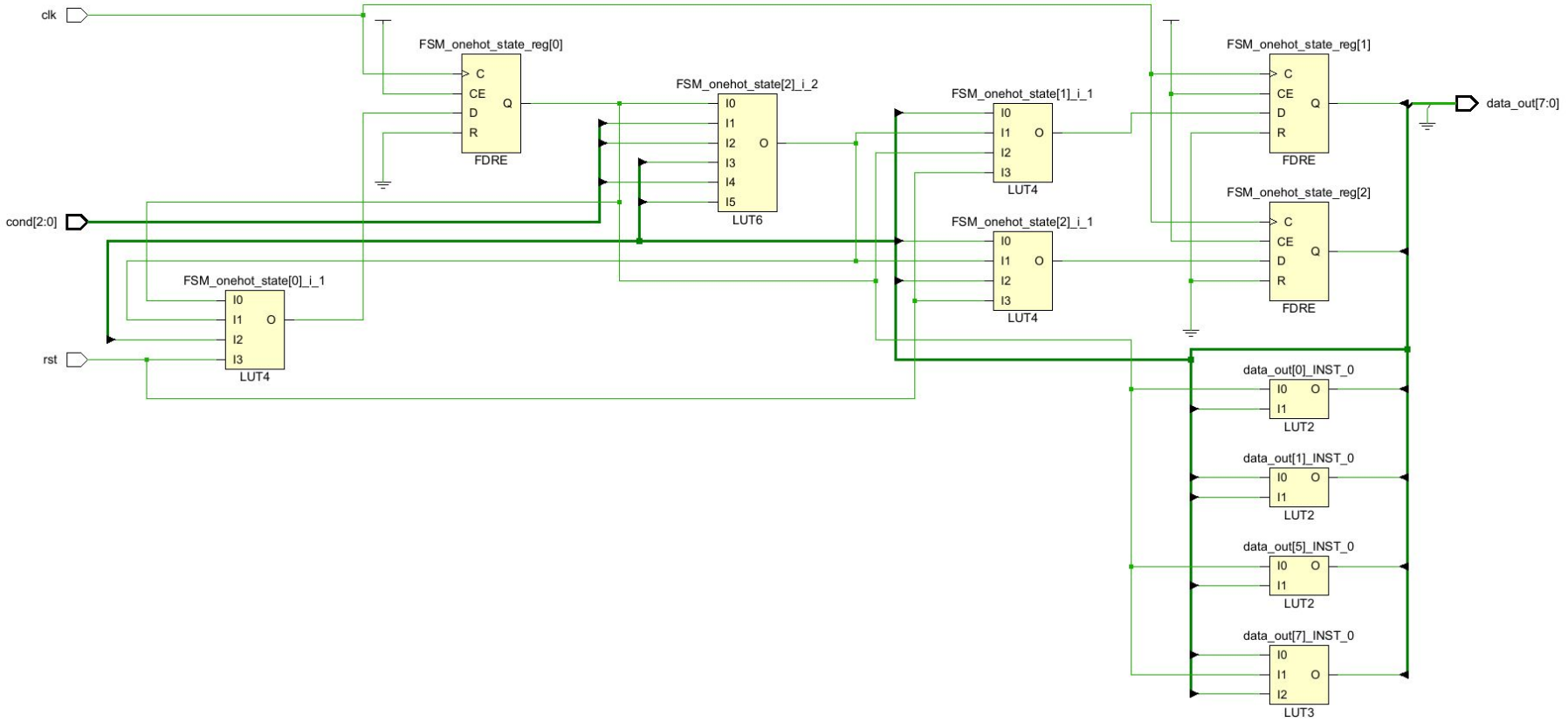
FSM - Two process

```
entity fsm_20 is
  port (
    clk       : in std_logic;
    rst       : in std_logic;
    cond      : in std_logic_vector(2 downto 0);
    data_out  : out std_logic_vector(7 downto 0)
  );
end fsm_20;
architecture rtl of fsm_20 is
  type state_type is (STATE_1, STATE_2, STATE_3);
  signal state      : state_type;
  signal next_state : state_type;
begin
  SYNC_PROC : process(clk)
  begin
    if rising_edge(clk) then
      if rst = '1' then
        state <= STATE_1;
      else
        state <= next_state;
      end if;
    end if;
  end process;
  DOUT_PROC : process(all)
  begin
```

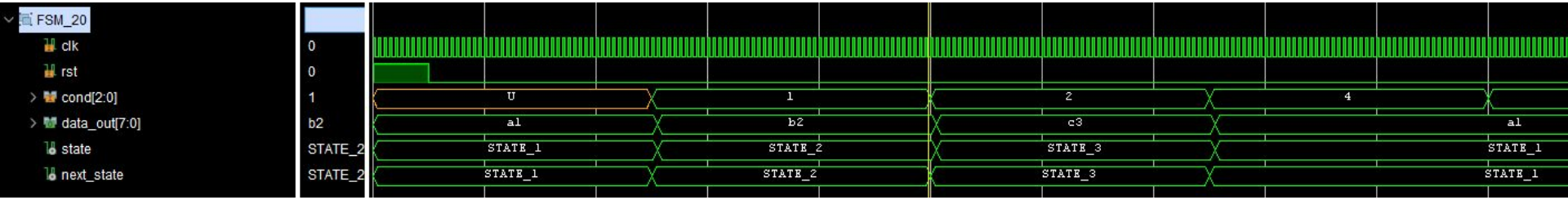
RTL Schematic

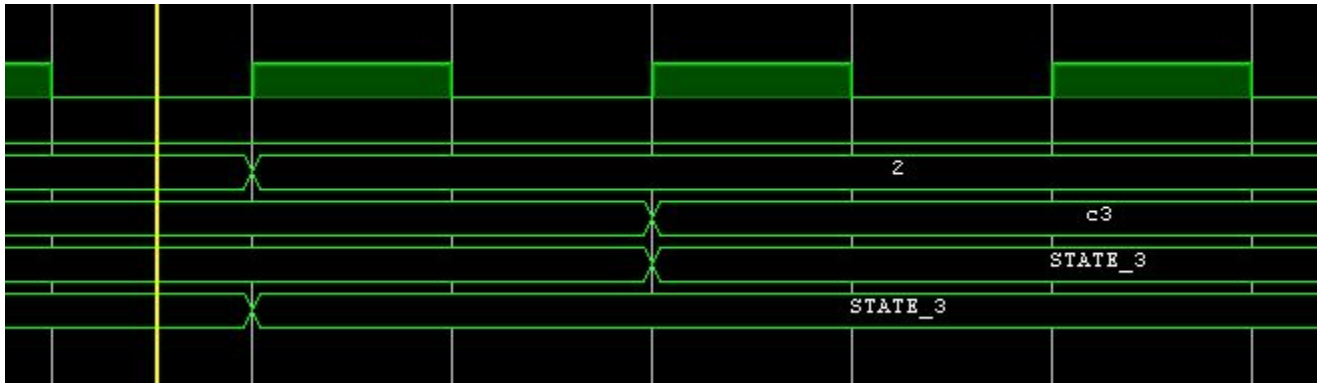


FSM - Three process



FSM - Three process // Synthesized schematic





Simulation

FSM - Three process

Synthesis
From code to hardware