FSM Components

- XST features:
  - Specific inference capabilities for synchronous Finite State Machine (FSM) components.
  - Built-in FSM encoding strategies to accommodate your optimization goals.
- You may also instruct XST to follow your own encoding scheme.
- FSM extraction is *enabled* by default.
- Use Automatic FSM Extraction to *disable* FSM extraction.

FSM Description

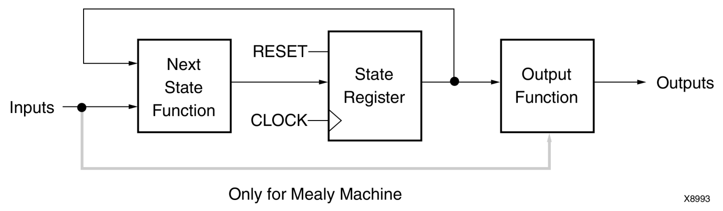
- XST supports specification of Finite State Machine (FSM) in both Moore and Mealy form.
- An FSM consists of:
  - State register
  - Next state function
  - Outputs function

HDL Coding Methods

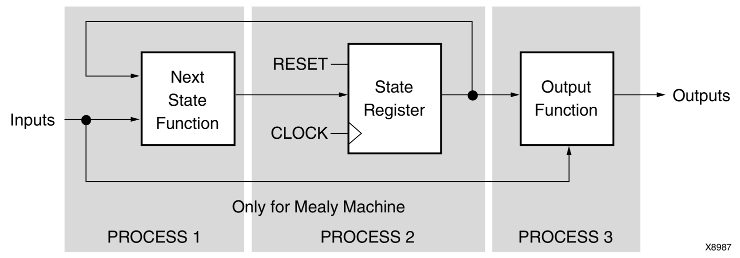
- You can choose among many HDL coding methods. Your choice depends on your goals with respect to code compactness and readability.
- The following HDL coding methods:
  - Ensure maximum readability.
  - Maximize the ability of XST to identify the FSM.
- Method One

  Describe all three components of the FSM in a single sequential process or **always** block.
- Method Two
  1. Describe the state register and next state function together in a sequential process or **always** block.
  2. Describe the outputs function in a separate combinatorial process or **always** block.
- Method Three
  1. Describe the state register in a sequential process or **always** block.
  2. Describe the next state and outputs functions together in a separate combinatorial process or **always** block.
- Method Four
  1. Describe the state register in a sequential process or **always** block.
  2. Describe the next state function in a first combinatorial process or **always** block.
  3. Describe the outputs function in a second separate combinatorial process or **always** block.

FSM Representation Incorporating Mealy and Moore Machines Diagram



FSM With Three Processes Diagram



State Registers

- Specify a reset or power-up state for XST to identify a Finite State Machine (FSM).
- The State Register can be asynchronously or synchronously reset to a particular state.
- Xilinx® recommends using synchronous reset logic over asynchronous reset logic for an FSM.

Specifying State Registers in VHDL

You can specify a State Register in VHDL with:

- Standard Type
- Enumerated Type

Standard Type

Specify the State Register with a Standard Type such as:

- integer
- bit_vector
- std_logic_vector

Enumerated Type

1. Define an Enumerated Type containing all possible state values.
2. Declare the state register with that type.

```
type state_type is (state1, state2, state3, state4);
signal state : state_type;
```

Specifying State Registers in Verilog

- A State Register type in Verilog is:
  - An integer, or
  - A set of defined parameters.

    ```
    parameter [3:0]
        s1 = 4'b0001,
        s2 = 4'b0010,
        s3 = 4'b0100,
        s4 = 4'b1000;
    reg [3:0] state;
    ```

- Modify these parameters to represent different state encoding schemes.

Next State Equation

- Next state equations can be described:
  - Directly in the sequential process, or
  - In a separate combinatorial process
- The sensitivity list of a separate combinatorial process contains:
  - The state signal
  - All Finite State Machine (FSM) inputs
- The simplest coding example is based on a **case** statement, the selector of which is the current state signal.

Unreachable States

XST detects and reports unreachable states.

FSM Outputs

- Non-registered outputs are described in:
  - The combinatorial process, or
  - Concurrent assignments
- Registered outputs must be assigned in the sequential process.

FSM Inputs

- Registered inputs are described using internal signals.
- Internal signals are assigned in the sequential process.

State Encoding Techniques

- XST state encoding techniques accommodate different optimization goals, and different Finite State Machine (FSM) patterns.
- Use FSM Encoding Algorithm to select the state encoding technique.
- For more information, see Chapter 9, Design Constraints.

Auto State Encoding

XST tries to select the best suited encoding method for a given FSM.

One-Hot State Encoding

- Is the default encoding scheme.
- Is usually a good choice for optimizing speed or reducing power dissipation.
- Assigns a distinct bit of code to each FSM state.
- Implements the State Register with one flip-flop for each state.
  - In a given clock cycle during operation, one and only one bit of the State Register is asserted.
  - Only two bits toggle during a transition between two states.

Gray State Encoding

- Guarantees that only one bit switches between two consecutive states.
- Is appropriate for controllers exhibiting long paths without branching.
- Minimizes hazards and glitches.
- Gives good results when implementing the State Register with T Flip-Flops.
- Can be used to minimize power dissipation.

Compact State Encoding

- Minimizes the number of bits in the state variables and flip-flops. This technique is based on hypercube immersion.
- Is appropriate when trying to optimize area.

Johnson State Encoding

Beneficial when using state machines containing long paths with no branching (as in Gray State Encoding).

Sequential State Encoding

- Identifies long paths
- Applies successive radix two codes to the states on these paths.
- Minimizes next state equations.

Speed1 State Encoding

- Is oriented for speed optimization.
- The number of bits for a State Register depends on the specific FSM, but is generally greater than the number of FSM states.

User State Encoding

XST uses the original encoding specified in the HDL file.

User State Encoding Example

If the State Register is described based on an enumerated type:

- Use Enumerated Encoding to assign a specific binary value to each state.
- Select User State Encoding to instruct XST to follow your coding scheme.

Implementing FSM Components on Block RAM Resources

- Finite State Machine (FSM) components are implemented on slice logic.
  - To save slice logic resources, instruct XST to implement FSM components in block RAM.
  - Implementing FSM components in block RAM can enhance the performance of large FSM components.
- To select the implementation for slice logic, use FSM Style to choose between:
  - default implementation
  - block RAM implementation
- The values for FSM Style are:
  - lut (default)
  - bram
- If XST cannot implement an FSM in block RAM:
  - XST implements the state machine in slice logic.
  - XST issues a warning during Advanced HDL Synthesis.
- The failure to implement an FSM in block RAM usually occurs when the FSM has an asynchronous reset.

FSM Safe Implementation

Safe Finite State Machine (FSM) design is a subject of debate. There is no single perfect solution. Xilinx® recommends that you carefully review the following sections before deciding on your implementation strategy.

Optimization

- Optimization is standard for the great majority of applications. Most applications operate in normal external conditions. Their temporary failure due to a single event upset does not have critical consequences.
- XST detects and optimizes the following by default:
  - Unreachable states (both logical and physical)
  - Related transition logic
- Optimization ensures implementation of a state machine that:
  - Uses minimal device resources.
  - Provides optimal circuit performance.

Preventing Optimization

- Some applications operate in external conditions in which the potentially catastrophic impact of soft errors cannot be ignored. Optimization is not appropriate for these applications.
- These soft errors are caused primarily by:
  - Cosmic rays, or
  - Alpha particles from the chip packaging
- State machines are sensible to soft errors. A state machine may never resume normal operation after an external condition sends it to an illegal state. For the circuit to be able to detect and recover from those errors, unreachable states must not be optimized away.
- Use Safe Implementation to prevent optimization. XST creates additional logic allowing the state machine to:
  - Detect an illegal transition.
  - Return to a valid recovery state.
- XST selects the reset state as the recovery state by default. If no reset state is available, XST selects the power-up state. Use Safe Recovery State to manually define a specific recovery state.

One-Hot Encoding Versus Binary Encoding

- With binary State Encoding Techniques (such as Compact, Sequential, and Gray), the state register is implemented with a minimum number of Flip-Flops. One-Hot Encoding implies a larger number of Flip-Flops (one for each valid state). This increases the likelihood of a single event upset affecting the State Register.
- Despite this drawback, One-Hot Encoding has a significant topological benefit. A Hamming distance of **2** makes all single bit errors easily detectable. An illegal transition resulting from a single bit error always sends the state machine to an invalid state. The XST safe implementation logic ensures that any such error is detected and cleanly recovered from.
- An equivalent binary coded state machine has a Hamming distance of **1**. As a result, a single bit error may send the state machine to an unexpected but valid state. If the number of valid states is a power of **2**, all possible code values correspond to a valid state, and a soft error always produces such an outcome. In that event, the circuit does not detect that an illegal transition has occurred, and that the state machine has not executed its normal state sequence. Such a random and uncontrolled recovery may not be acceptable.

Recovery-Only States

- Xilinx recommends that you define a recovery state that is *none* of the normal operating states of your state machine.
- Defining a recovery-only state allows you to:
  - Detect that the state machine has been affected by a single event upset.
  - Perform specific actions before resuming normal operation. Such actions include flagging the recovery condition to the rest of the circuit or to a circuit output.
- Directly recovering to a normal operation state is sufficient, provided that the faulty state machine does not need to:
  - Inform the rest of the circuit of its temporary condition, or
  - Perform specific actions following a soft error.

FSM Safe Implementation VHDL Coding Example

```
--
-- Finite State Machine Safe Implementation VHDL Coding Example
--   One-hot encoding
--   Recovery-only state
--
-- Download: http://www.xilinx.com/txpatches/pub/documentation/misc/xstug_examples.zip
-- File: HDL_Coding_Techniques/state_machines/safe_fsm.vhd
--
library ieee;
use ieee.std_logic_1164.all;

entity safe_fsm is

  port(
    clk : in  std_logic;
    rst : in  std_logic;
    c   : in  std_logic_vector(3 downto 0);
    d   : in  std_logic_vector(3 downto 0);
    q   : out std_logic_vector(3 downto 0));

end safe_fsm;

architecture behavioral of safe_fsm is

  type state_t is ( idle, state0, state1, state2, recovery );
  signal state, next_state : state_t;

  attribute fsm_encoding : string;
  attribute fsm_encoding of state : signal is "one-hot";
  attribute safe_implementation : string;
  attribute safe_implementation of state : signal is "yes";
  attribute safe_recovery_state : string;
  attribute safe_recovery_state of state : signal is "recovery";

begin

  process(clk)
  begin
    if rising_edge(clk) then
      if rst = '1' then
        state <= idle;
      else
        state <= next_state;
      end if;
    end if;
  end process;

  process(state, c, d)
  begin

    next_state <= state;

    case state is
      when idle =>
        if c(0) = '1' then
          next_state <= state0;
        end if;
        q <= "0000";

      when state0 =>
        if c(0) = '1' and c(1) = '1' then
          next_state <= state1;
        end if;
        q <= d;

      when state1 =>
        next_state <= state2;
        q <= "1100";

      when state2 =>
        if c(1) = '0' then
          next_state <= state1;
        elsif c(2) = '1' then
          next_state <= state2;
        elsif c(3) = '1' then
          next_state <= idle;
        end if;
        q <= "0101";

      when recovery =>
        next_state <= state0;
        q <= "1111";

    end case;

  end process;

end behavioral;
```

Verilog Support for FSM Safe Implementation

- Because Verilog does not provide enumerated types, Verilog support for FSM safe implementation is more restrictive than VHDL.
- **Recommendation** Follow these coding guidelines for proper implementation of the state machine:
  - Manually enforce the desired encoding strategy.
    - Explicitly define the code value for each valid state.
    - Set FSM Encoding Algorithm to **User**.
  - Use **localparam** or **'define** for readability to symbolically designate the various states in the state machine description.
  - Hard code the recovery state value as one of the following, since it cannot be referred to symbolically in a Verilog attribute specification:
    - A string, directly in the attribute statement, or
    - A **'define**, as shown in the following coding example

FSM Safe Implementation Verilog Coding Example

```
//
// Finite State Machine Safe Implementation Verilog Coding Example
//   One-hot encoding
//   Recovery-only state
//
// Download: http://www.xilinx.com/txpatches/pub/documentation/misc/xstug_examples.zip
// File: HDL_Coding_Techniques/state_machines/safe_fsm.v
//
module v_safe_fsm (clk, rst, c, d, q);

  input              clk;
  input              rst;
  input       [3:0]  c;
  input       [3:0]  d;
  output reg  [3:0]  q;

  localparam [4:0]
    idle     = 5'b00001,
    state0   = 5'b00010,
    state1   = 5'b00100,
    state2   = 5'b01000,
    recovery = 5'b10000;

  'define recovery_attr_val "10000"

  (* fsm_encoding = "user",
     safe_implementation = "yes",
     safe_recovery_state = 'recovery_attr_val *)
     // alternatively:  safe_recovery_state = "10000" *)
  reg   [4:0]  state;
  reg   [4:0]  next_state;

  always @ (posedge clk)
  begin
      if (rst)
        state <= idle;
      else
        state <= next_state;
  end

  always @(*)
  begin

    next_state <= state;

    case (state)

      idle: begin
          if (c[0])
            next_state <= state0;
          q <= 4'b0000;
      end

      state0: begin
          if (c[0] && c[1])
            next_state <= state1;
          q <= d;
      end

      state1: begin
          next_state <= state2;
          q <= 4'b1100;
      end

      state2: begin
          if (~c[1])
            next_state <= state1;
          else
            if (c[2])
              next_state <= state2;
            else
              if (c[3])
                next_state <= idle;
              q <= 4'b0101;
      end

      recovery: begin
          next_state <= state0;
          q <= 4'b1111;
      end

      default: begin
          next_state <= recovery;
          q <= 4'b1111;
      end

    endcase

  end

endmodule
```

FSM Related Constraints

- Automatic FSM Extraction
- FSM Style
- FSM Encoding Algorithm
- Enumerated Encoding
- Safe Implementation
- Safe Recovery State

FSM Reporting

The XST log provides detailed information about Finite State Machine (FSM) components and their encoding.

FSM Reporting Example

```
=========================================================================
*                          HDL Synthesis                                *
=========================================================================

Synthesizing Unit <fsm_1>.
    Found 1-bit register for signal <outp>.
    Found 2-bit register for signal <state>.
    Found finite state machine <FSM_0> for signal <state>.
    -----------------------------------------------------------------------
    | States             | 4                                              |
    | Transitions        | 5                                              |
    | Inputs             | 1                                              |
    | Outputs            | 2                                              |
    | Clock              | clk (rising_edge)                              |
    | Reset              | reset (positive)                               |
    | Reset type         | asynchronous                                   |
    | Reset State        | s1                                             |
    | Power Up State     | s1                                             |
    | Encoding           | gray                                           |
    | Implementation     | LUT                                            |
    -----------------------------------------------------------------------
    Summary:
 inferred   1 D-type flip-flop(s).
 inferred   1 Finite State Machine(s).
Unit <fsm_1> synthesized.

=========================================================================
HDL Synthesis Report

Macro Statistics
# Registers                                            : 1
 1-bit register                                        : 1
# FSMs                                                 : 1

=========================================================================

=========================================================================
*                      Advanced HDL Synthesis                           *
=========================================================================

=========================================================================
Advanced HDL Synthesis Report

Macro Statistics
# FSMs                                                 : 1
# Registers                                            : 1
 Flip-Flops                                            : 1
# FSMs                                                 : 1

=========================================================================

=========================================================================
*                        Low Level Synthesis                            *
=========================================================================
Optimizing FSM <state> on signal <state[1:2]> with gray encoding.
-------------------
 State | Encoding
-------------------
 s1    | 00
 s2    | 11
 s3    | 01
 s4    | 10
-------------------
```

FSM Coding Examples

For update information, see "Coding Examples" in the Introduction.

FSM Described with a Single Process VHDL Coding Example

```
--
-- State Machine described with a single process
--
-- Download: http://www.xilinx.com/txpatches/pub/documentation/misc/xstug_examples.zip
-- File: HDL_Coding_Techniques/state_machines/state_machines_1.vhd
--
library IEEE;
use IEEE.std_logic_1164.all;

entity fsm_1 is
    port ( clk, reset, x1 : IN std_logic;
           outp           : OUT std_logic);
end entity;

architecture behavioral of fsm_1 is
    type state_type is (s1,s2,s3,s4);
    signal state : state_type ;
begin

    process (clk)
    begin
        if rising_edge(clk) then
            if (reset ='1') then
         state <= s1;
                outp <= '1';
     else
                case state is
                    when s1 =>  if x1='1' then
                                    state <= s2;
                                    outp <= '1';
                                else
                                    state <= s3;
                                    outp <= '0';
                                end if;
                    when s2 => state <= s4; outp <= '0';
                    when s3 => state <= s4; outp <= '0';
                    when s4 => state <= s1; outp <= '1';
                end case;
            end if;
        end if;
    end process;

end behavioral;
```

FSM with Three Always Blocks Verilog Coding Example

```
//
// State Machine with three always blocks.
//
// Download: http://www.xilinx.com/txpatches/pub/documentation/misc/xstug_examples.zip
// File: HDL_Coding_Techniques/state_machines/state_machines_3.v
//
module v_fsm_3 (clk, reset, x1, outp);
    input clk, reset, x1;
    output outp;
    reg outp;
    reg [1:0] state;
    reg [1:0] next_state;

    parameter s1 = 2'b00; parameter s2 = 2'b01;
    parameter s3 = 2'b10; parameter s4 = 2'b11;

    initial begin
        state = 2'b00;
    end

    always @(posedge clk or posedge reset)
    begin
        if (reset) state <= s1;
        else state <= next_state;
    end

    always @(state or x1)
    begin
        case (state)
            s1: if (x1==1'b1)
                    next_state = s2;
                else
                    next_state = s3;
            s2: next_state = s4;
            s3: next_state = s4;
            s4: next_state = s1;
        endcase
    end

    always @(state)
    begin
        case (state)
            s1: outp = 1'b1;
            s2: outp = 1'b1;
            s3: outp = 1'b0;
            s4: outp = 1'b0;
        endcase
    end

endmodule
```