Report: Analysis of the FSM VHDL Code and Testbench

Table of Contents

1. Introduction

This report analyzes the provided VHDL code, which implements a Finite State Machine (FSM) with four states: idle, proc_1, halt, and final. The code also includes a testbench designed to validate the FSM's functionality.

2. FSM VHDL Code

2.1 Entity Declaration

The entity FSM defines the inputs and outputs of the FSM module:

- clk: Clock signal (input).
- reset: Asynchronous reset signal (input).
- a: Input signal that controls state transitions.
- b: Another input signal that also controls state transitions.
- fsm_out: Output signal that indicates the state of the FSM.
- main: A 3-bit output vector that provides a coded representation of the current state.

2.2 Architecture - Behavioral

The FSM is implemented in the Behavioral architecture. The architecture includes the following components:

2.2.1 State Definitions

- A state type state_type is defined, with four states: idle, proc_1, halt, and final.
- Two signals, current_state and next_state, are defined to keep track of the FSM's current and next state, respectively.

2.2.2 Process 1: State Register

The first process, labeled proc1, is responsible for updating the current_state on the rising edge of the clock or resetting it to idle if the reset signal is asserted.

```
proc1: process(clk, reset) begin
   if reset = '1' then
      current_state <= idle;
   elsif(rising_edge(clk)) then
      current_state <= next_state;
   end if;
end process;
```

2.2.3 Process 2: Next State Logic and Output Logic

The second process, labeled proce2, determines the next state and the output values based on the current state and input signals (a and b).

```
proce2: process(a, b, current_state)
```

```
begin
   next_state <= current_state;
   fsm_out <= '0';
   main <= "000";

   case current_state is
      when idle =>
         fsm_out <= '1';
         main <= "000";
         if (a = '1') then
            next_state <= proc_1;
         elsif (a='0' and b='1') then
            next_state <= halt;
         end if;

      when proc_1 =>
         fsm_out <= '0';
         main <= "100";
         if a='0' then
            next_state <= halt;
         end if;

      when halt =>
         fsm_out <= '0';
         main <= "010";
         if a='0' then
            next_state <= halt;
         elsif a = '1' then
            next_state <= final;
         end if;

      when final =>
         next_state <= idle;
         fsm_out <= '1';
         main <= "001";

      when others =>
         next_state <= idle;
         fsm_out <= '0';
         main <= "000";

   end case;
end process;
```

State Transition and Output Behavior:

- **idle**: fsm_out is set to '1' and main to "000". The FSM transitions to proc_1 if a = '1' or to halt if a = '0' and b = '1'.
- **proc_1**: fsm_out is '0', and main is "100". It transitions to halt if a = '0'.
- **halt**: fsm_out is '0', and main is "010". It stays in halt if a = '0' or moves to final if a = '1'.
- **final**: The FSM transitions back to idle with fsm_out set to '1' and main to "001".

3. Testbench Analysis

3.1 Entity Declaration

The testbench entity tb_FSM has no ports since it is designed to stimulate the FSM internally.

3.2 Architecture - Behavior

The architecture defines:

- **Inputs and Outputs**: The signals clk, reset, a, and b serve as inputs to the FSM, while fsm_out and main are outputs.
- **Clock Process**: A process is defined to generate a clock signal with a period of 10 ns.

3.3 Stimulus Process

The stimulus process is responsible for applying test vectors to the FSM:

- The reset signal is asserted for the first 20 ns to initialize the FSM to the idle state.
- Various combinations of a and b are then applied to verify the state transitions.

Test Scenarios:

1. **Test 1**: The FSM is reset, a is set to '1', transitioning from idle to proc_1. a is toggled to check the transition to halt.
2. **Test 2**: Inputs a and b are manipulated to check transitions from halt to final, ensuring the FSM returns to idle.

4. Conclusion

The FSM VHDL code implements a simple state machine with well-defined state transitions controlled by input signals a and b. The output signals fsm_out and main indicate the current state of the FSM. The testbench successfully stimulates the FSM and verifies that it transitions correctly between states. The architecture is modular, with clear separation between the state register, next-state logic, and output logic, making the design robust and easy to understand.