INTRODUCTION TO FPGA PROGRAMMING

Lesson 09: Finite State Machines

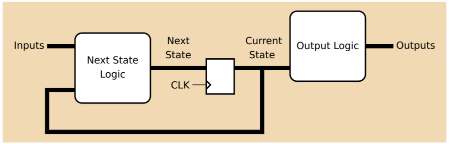
Dr. Davide Cieri[1]

[1]Max-Planck-Institut für Physik, Munich

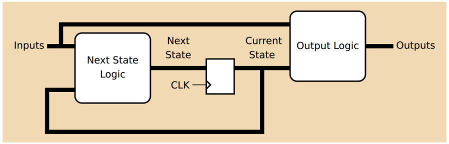
September 2024

FINITE STATE MACHINES

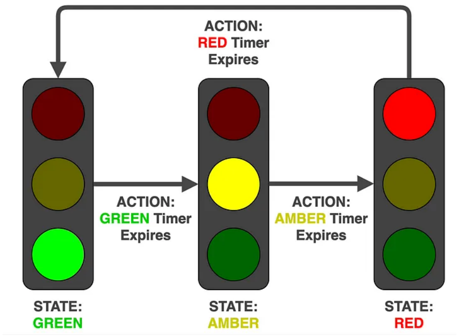
- A Finite State Machine (FSM) is a computational model used to design sequential logic circuits.
- Consists of a finite number of states, transitions between those states, and actions.
- **Moore FSM**: Outputs depend only on the state



- **Mealy FSM**: Outputs depend on state and inputs



STATES, TRANSITIONS AND ACTIONS

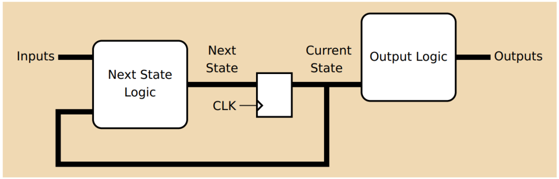

ACTION: RED Timer Expires
ACTION: GREEN Timer Expires
ACTION: AMBER Timer Expires
STATE: GREEN
STATE: AMBER
STATE: RED

- **State**: describe the status of the system.
- **Transition**: The action of moving from one state to the other.
- **Action**: The event triggering a transition
  - The same action can have different effect depending on the current state

MOORE VS MEALY

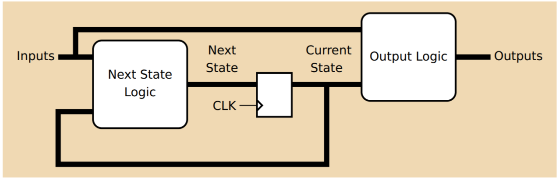
Moore

- Advantages:
  - Simpler circuit, Faster Clock Frequency
  - No combinational input to output
  - Can have output encoded FSM states
- Disadvantages:
  - Slower to react to input changes



Mealy

- Advantages:
  - Faster to react to inputs
- Disadvantages:
  - Combinational inputs to Outputs
  - More complex circuit (slower clock)



DEFINE FSM STATES IN VHDL

- States can me encoded as constants or enumerated types
- Enumerated types are more intuitive
- Constant allows you to access each state vector bits
- States and types must be defined in the declarative part of your architecture (before `begin`)

```vhdl
-- Constant States
constant RED : std_logic_vector(1 downto 0) := "00";
constant AMBER : std_logic_vector(1 downto 0) := "01";
constant GREEN : std_logic_vector(1 downto 0) := "10";
signal state : std_logic_vector(1 downto 0);
-- Enumerated type
type fsm_state is (RED, AMBER, GREEN);
signal state : fsm_state := RED;
```

STATE ENCODING IMPLEMENTATION

- When using enumerated type, the synthesis tool will encode the states into binary vectors
- Three main possibilities:
  - *Binary Encoding* (default): States are encoded in the minimum number of bits needed to represent all states
  - *One-hot Encoding*: Each state is represented by one bit and the FSM is encoded so only one bit set at any time.
  - *Gray code Encoding*: The states are encoded in such a way that any state transition only has one bit change at a time.
- You can force Vivado using one of three encoding using the attribute `fsm_encoding` (link to documentation)

```
attribute fsm_encoding : string;
attribute fsm_encoding of state : signal is "one_hot";
```

BINARY ENCODING

- States are assigned binary values.
- Number of bits required: $\log_2(\text{Number of states})$.
- **Advantages:** Minimal bit width, efficient use of state register.
- **Disadvantages:** Complex state transition logic, higher risk of glitches.

Example:

- For 4 states: S0 = "00", S1 = "01", S2 = "10", S3 = "11".

ONE-HOT ENCODING

- Each state is represented by a single bit set to ’1’, all others are ’0’.
- Number of bits required = Number of states.
- **Advantages:** Simple state transition logic, faster operation.
- **Disadvantages:** Higher resource usage, less efficient for many states.

Example:

- For 4 states: S0 = "0001", S1 = "0010", S2 = "0100", S3 = "1000".

GRAY ENCODING

- Adjacent states differ by only one bit.
- Number of bits required: $\log_2(\text{Number of states})$.
- **Advantages:** Minimizes switching noise, ideal for asynchronous systems.
- **Disadvantages:** More complex to implement, less common.

Example:

- For 4 states: S0 = "00", S1 = "01", S2 = "11", S3 = "10".

IMPLEMENTING FSM IN VHDLS

Single-Process Implementation

- **Structure:** Combines all logic (state transitions, output, sequential) in one process block.
- **Advantages:**
  - Simpler design with fewer processes.
  - Easier to manage for small FSMs.
- **Disadvantages:**
  - Difficult to debug due to mixed logic.
  - Lower readability for complex FSMs.

Multiple-Process Implementation

- **Structure:** Separates FSM into distinct processes:
  - State Register Process (sequential)
  - Next State Logic Process (combinational)
  - Output Logic Process (optional)
- **Advantages:**
  - Higher clarity and modularity.
  - Easier to debug and modify.
- **Disadvantages:**
  - More verbose code.
  - Requires careful synchronization.

SINGLE SEQUENTIAL PROCESS EXAMPLE

```
if rising_edge(clk) then
  if reset = '1' then
      state <= S0;
  else
      case state is
          when S0 =>
              if input = '1' then
                  state <= S1;
              end if;
              output <= '0';
          when S1 =>
              state <= S0;
              output <= '1';
          when others =>
              state <= S0;
      end case;
  end if;
end if;
```

MULTIPLE-PROCESS FSM EXAMPLE

```vhdl
-- State register process
process (clk)
begin
    if rising_edge(clk) then
        if reset = '1' then
            state <= S0;
        else
            state <= next_state;
        end if;
    end if;
end process;
```

```vhdl
-- Next state logic process
process (state, input)
begin
    case state is
        when S0 =>
            if input = '1' then
                next_state <= S1;
            else
                next_state <= S0;
            end if;
        when S1 =>
            next_state <= S0;
        when others =>
            next_state <= S0;
    end case;
end process;
```

```vhdl
-- Output logic process
process (state)
begin
    case state is
        when S0 =>
            output <= '0';
        when S1 =>
            output <= '1';
        when others =>
            output <= '0';
    end case;
end process;
```

- N.B. The Output logic can be placed in any of the other two processes, depending if we want registered or combinatorial outputs.

STATE MACHINE BEST PRACTICES

- **One state machine per file:** Improves readability and maintainability.
- **Use more than one process with complex FSMs:** At least one clocked for the states, and one combinational for next state.
- **Meaningful state names:** Improves code clarity.
- **Draw flow diagrams:** Visualize FSM before coding.

LAB 14: IMPROVE THE TRAFFIC LIGHT

LAB 15: DESIGN A STOP WATCH

The figures in these slides are taken from:

- Digital Design: Principles and Practices, Fourth Edition, John F. Wakerly, ISBN 0-13- 186389-4.
©2006, Pearson Education, Inc, Upper Saddle River, NJ. All rights reserved
- allaboutfpga.com
- nandland.com
- docs.amd.com
- https://www.symmetryelectronics.com/
- https://www.edn.com/
- Stephen A. Edwards, Columbia University, Fundamentals of Computer Systems, Spring 2012
- https://medium.com/well-red/state-machines-for-everyone-part-1-introduction-b7ac9aaf482e