Finite State Machine

1. Overview on FSM
   - Contain “random” logic in next-state logic
   - Used mainly used as a controller in a large system
   - Mealy vs Moore output

2. Representation of FSM
   - State diagram

E.g. a memory controller

3. ASM (algorithmic state machine) chart
   - Flowchart-like diagram
   - Provide the same info as an FSM
   - More descriptive, better for complex description
   - ASM block
     - One state box
     - One or more optional decision boxes: with T or F exit path
     - One or more conditional output boxes: for Mealy output

Outline
1. Overview
2. FSM representation
3. Timing and performance of an FSM
4. Moore machine versus Mealy machine
5. VHDL description of FSMs
6. State assignment
7. Moore output buffering
8. FSM design examples
State diagram and ASM chart conversion

- E.g. 1.

- E.g. 2.

- E.g. 3.

- E.g. 4.

- E.g. 6.
• Difference between a regular flowchart and ASM chart:
  – Transition governed by clock
  – Transition done between ASM blocks

• Basic rules:
  – For a given input combination, there is one unique exit path from the current ASM block
  – The exit path of an ASM block must always lead to a state box. The state box can be the state box of the current ASM block or a state box of another ASM block.

3. Performance of FSM
• Similar to regular sequential circuit

\[ T_c = T_{cq} + T_{next(x)} + T_{setup} \]
\[ T_{co(x)} = T_{cq} + T_{output(xo)} \]

• Incorrect ASM charts:

4. Sample timing diagram
4. Moore vs Mealy output

- **Moore machine:**
  - output is a function of state
- **Mealy machine:**
  - output function of state and output
- **From theoretical point of view**
  - Both machines have similar “computation capability”
- **Implication of FSM as a controller?**

  - E.g., edge detection circuit
    - A circuit to detect the rising edge of a slow “strobe” input and generate a “short” (about 1-clock period) output pulse.

  - Three designs:

  ![Diagram of three designs](image)

  - Comparison
    - Mealy machine uses fewer states
    - Mealy machine responds faster
    - Mealy machine may be transparent to glitches
  - Which one is better?
  - Types of control signal
    - Edge sensitive
      - E.g., enable signal of counter
      - Both can be used but Mealy is faster
    - Level sensitive
      - E.g., write enable signal of SRAM
      - Moore is preferred

VHDL Description of FSM

- Follow the basic block diagram
- Code the next-state/output logic according to the state diagram/ASM chart
- Use enumerate data type for states
library ieee;
use ieee.std_logic_1164.all;
entity mem_ctrl is
port ( 
   clk, reset: in std_logic;
   mem, rw, burst: in std_logic;
   we, we_xo: out std_logic);
edn mem_ctrl;
architecture multi_reg_arch of mem_ctrl is
   type nc_state_type is
      (idle, read1, read2, read3, read4, write);
   signal state_reg, state_next: nc_state_type;
begin
   -- state register
   process(clk, reset)
   begin
      if (reset='1') then
         state_reg <= idle;
      elsif (clk'event and clk='1') then
         state_reg <= state_next;
      end if;
   end process;

   when read1 =>
      if (burst='1') then
         state_next <= read2;
      else
         state_next <= idle;
      end if;
   when read2 =>
      state_next <= read3;
   when read3 =>
      state_next <= read4;
   when read4 =>
      state_next <= idle;
   end case;
end process;

-- next_state logic
process(state_reg, mem, rw, burst)
begin
   case state_reg is
      when idle =>
         if mem='1' then
            if rw='1' then
               state_next <= read1;
            else
               state_next <= write;
            end if;
         else
            state_next <= idle;
         end if;
      when write =>
         state_next <= idle;
   end case;
end process;

-- moore output logic
process(state_reg)
begin
   we <= '0'; -- default value
   ow <= '0'; -- default value
   case state_reg is
      when idle =>
         when write =>
            we <= '1';
         when read1 =>
            ow <= '1';
         when read2 =>
            ow <= '1';
         when read3 =>
            ow <= '1';
         when read4 =>
            ow <= '1';
      when idle =>
         ow <= '1';
   end case;
end process;
6. State assignment

- State assignment: assign binary representations to symbolic states
- In a synchronous FSM
  - All assignments work
  - Good assignment reduce the complexity of next-state/output logic
- Typical assignment
  - Binary, Gray, one-hot, almost one-hot

<table>
<thead>
<tr>
<th>Table 10.1</th>
<th>State assignment example</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Binary assignment</td>
</tr>
<tr>
<td>idle</td>
<td>000</td>
</tr>
<tr>
<td>read1</td>
<td>001</td>
</tr>
<tr>
<td>read2</td>
<td>010</td>
</tr>
<tr>
<td>read3</td>
<td>011</td>
</tr>
<tr>
<td>read4</td>
<td>100</td>
</tr>
<tr>
<td>write</td>
<td>101</td>
</tr>
</tbody>
</table>

```vhdl
process(state_reg, mem, rw, burst)
begin
  oe <= '0';  -- default values
  we <= '1';
  we_me <= '0';
  case state_reg is
    when idle =>
      if mem='1' then
        if rw='1' then
          state_next <= read1;
        else
          state_next <= write;
          we_me <= '1';
        end if;
      else
        state_next <= idle;
      end if;
    when write =>
      state_next <= idle;
      we <= '1';
    when read1 =>
      if burst='1' then
        state_next <= read2;
      else
        state_next <= idle;
      end if;
    when read2 =>
      state_next <= read3;
      oe <= '1';
    when read3 =>
      state_next <= read4;
      oe <= '1';
    when read4 =>
      state_next <= idle;
      oe <= '1';
    end case;
  end case;
end process;
```
State assignment in VHDL

- Implicit: use user attributes `enum_encoding`

```vhdl
type mo.state_type is (idle, write, read1, read2, read3, read4);
attribute enum_encoding: string;
attribute enum_encoding of mo.state_type:
    type is "0000 0100 1000 1010 1011";
```

- Explicit: use `std_logic_vector` for the register

```vhdl
architecture state_assign_arch of mo fsm is
    constant idle: std_logic_vector(3 downto 0) := "0000";
    constant write: std_logic_vector(3 downto 0) := "0100";
    constant read1: std_logic_vector(3 downto 0) := "1000";
    constant read2: std_logic_vector(3 downto 0) := "1010";
    constant read3: std_logic_vector(3 downto 0) := "1011";
    constant read4: std_logic_vector(3 downto 0) := "1011";
    signal state_reg, state_next: std_logic_vector(3 downto 0);
begin
    state_reg <= idle;
    begin
        if (reset='1') then
            state_reg <= idle;
        elsif (clk'event and clk='1') then
            state_reg <= state_next;
        end if;
    end process;
end;
```

6. Moore output buffering

- FSM as control circuit
  - Sometimes fast, glitch-free signal is needed
  - An extra output buffer can be added, but introduce one-clock delay
- Special schemes can be used for Moore output
  - Clever state assignment
  - Look-ahead output circuit

Handling the unused state

- Many binary representations are not used
- What happens if the FSM enters an unused state?
  - Ignore the condition
  - Safe (Fault-tolerant) FSM: got to an error state or return to the initial state.
- Easy for the explicit state assignment
- No portable code for the enumerated data type

Potential problems of the Moore output logic:

- Potential hazards introduce glitches
- Increase the Tco delay (Tco = Tcq + Toutput)
- Can we get control signals directly from the register?
Clever state assignment

- Assigning state according to output signal patterns
- Output can be obtained from register directly
- Extra register bits may be needed
- Must use explicit state assignment in VHDL code to access individual register bit
- Difficult to revise and maintain

VHDL code

```vhdl
architecture state_assign_arch of mem fsm is
  constant idle: std_logic_vector(3 downto 0) := "0000";
  constant wait: std_logic_vector(3 downto 0) := "0100";
  constant read1: std_logic_vector(3 downto 0) := "1000";
  constant read2: std_logic_vector(3 downto 0) := "1010";
  constant read3: std_logic_vector(3 downto 0) := "1011";
  constant read4: std_logic_vector(3 downto 0) := "1111";
  signal state_reg, state_next: std_logic_vector(3 downto 0);

-- Moore output logic
  oe <= state_reg(3);
  we <= state_reg(2);
```

Look-ahead output circuit

- Output buffer introduces one-clock delay
- The "next" value of Moore output can be obtained by using state_next signal
- Buffer the next value cancel out the one-clock delay
- More systematic and easier to revise and maintain

Modification over original VHDL code:
- Add output buffer
- Use state_next to replace state_reg in Moore output logic

```vhdl
-- output buffer
  process(clk, reset)
  begin
    if (reset='1') then
      oe_buf_reg <= '0';
      we_buf_reg <= '0';
    elsif (clk'event and clk='1') then
      if ( oe_next = '1' ) then
        oe_buf_reg <= '0';
        we_buf_reg <= '0';
      elsif ( oe_next = '0' ) then
        oe_buf_reg <= oe_next;
        we_buf_reg <= we_next;
      end if;
    end if;
  end process;
```
7. FSM design examples

- Edge detector circuit
- Arbitrator (read)
- DRAM strobe signal generation
- Manchester encoding/decoding (read)
- FSM base binary counter
Edge detecting circuit (Mealy)

- Edge detecting circuit (direct implementation):
  - edge occurs when previous value is 0 and new value is 1
  - Same as Mealy design with state assignment:
    zero => 0, one => 1

```vhdl
architecture mealy_arch of edge_detector2 is
  type state_type is (zero, one);
  signal state_reg, state_next: state_type;
begin
  -- state register
  process(clk, reset)
  begin
    if (reset='1') then
      state_reg <= zero;
    elsif (clk'event and clk='1') then
      state_reg <= state_next;
    end if;
  end if;
end;
```

- DRAM strobe signal generation
  - E.g., 120ns DRAM (Trc=120ns):
    Tras=85ns, Tcas=20ns, Tpr=35ns
  - 3 intervals has to be at least 65ns, 20 ns, and 35 ns
  - A slow design: use a 65ns clock period
    - 195 ns (3*65ns) read cycle
  - The control signal is level-sensitive

```vhdl
architecture direct_arch of edge_detector2 is
  signal delay_reg: std_logic;
begin
  -- delay register
  process(clk, reset)
  begin
    if (reset='1') then
      delay_reg <= '0';
    elsif (clk'event and clk='1') then
      delay_reg <= strobe;
    end if;
  end if;
  end process;
  -- decoding logic
  p2 <= (not delay_reg and strobe);
end direct_arch;
```
• A faster design: use a 20ns clock period
  – 140 ns (7*20ns) read cycle

• FSM-based binary counter:
  – Free-running mod-16 counter

• RTL Hardware Design
  by P. Chu
  Chapter 10

• 4-bit binary counter with features:
  – Synchronous clear, load, enable

• Should revise the code to obtain glitch-free output

-- next-state logic
process(state_reg, en) begin
  case state_reg is
    when idle =>
      if en = '1' then
        state_next <= p;
      else
        state_next <= idle;
      end if;
      when r =>
        state_next <= id1;
      when p =>
        state_next <= p;
      end case;
next state_reg;
end process;

-- output logic
process(state_reg) begin
  cas_n <= '1';
  rase_n <= '1';
  rase => '1';
  state_reg is
    when idle =>
      rase_n <= '1';
    when r =>
      rase_n <= '1';
    when p =>
      rase_n <= '1';
next state_reg;
end case;
end process;

library ieee;
use ieee.std_logic_1164.all;
entity dram_strobe is
  port(
    clk, reset: in std_logic;
    mem: in std_logic;
    cas_n, rase_n: out std_logic
  )
end dram_strobe;
architecture fast_slv_clk_arch of dram_strobe is
  type fsm.state_type is (idle, r, c);
  signal state_reg, state_next: fsm.state_type;
begin
  -- state register
  process(clk, reset)
  begin
    if (reset = '1') then
      state_reg <= idle;
    elsif (clk'event and clk = '1') then
      state_reg <= state_next;
    end if;
  end process;
end fast_slv_clk_arch;