

# VHDL Modeling for Synthesis

Finite State Machines and Controllers

# Modeling Finite State Machines (Synchronous Sequential Circuits)

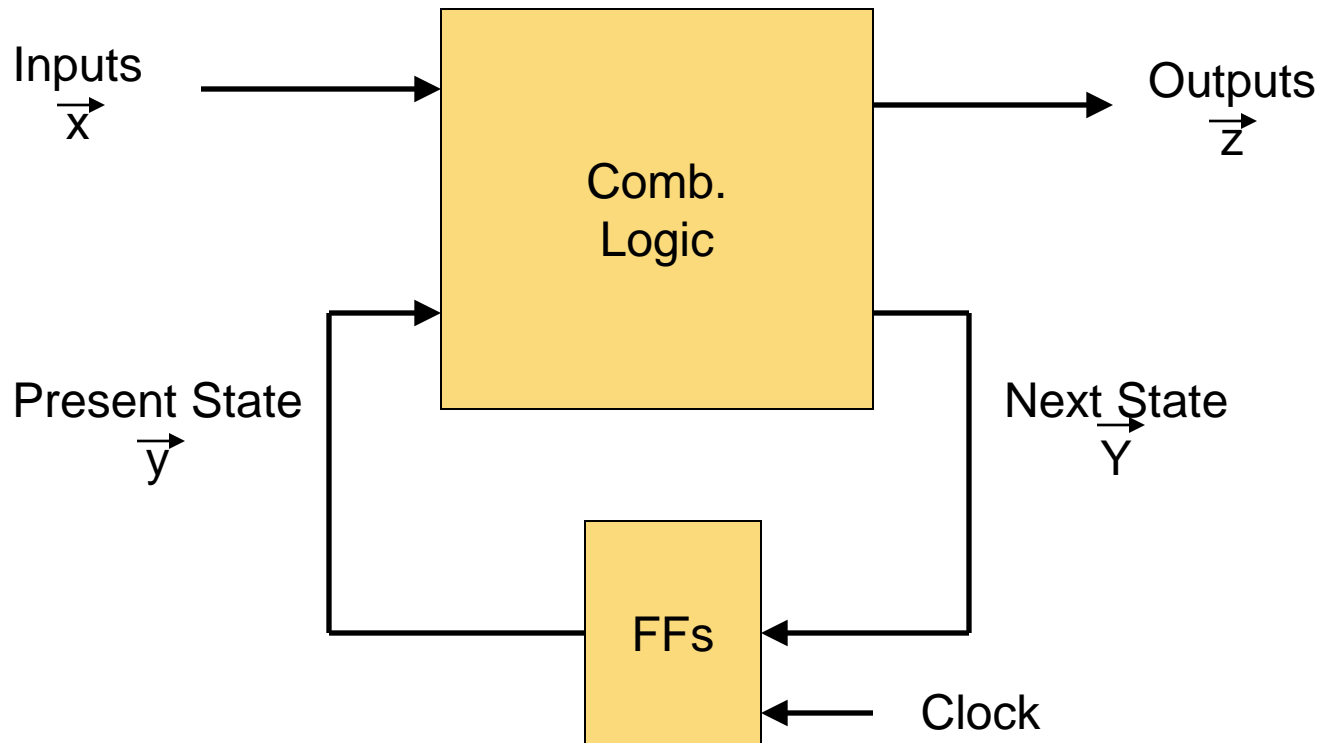
---

- ▶ **FSM design & synthesis process:**
  1. Design state diagram (behavior)
  2. Derive state table
  3. Reduce state table
  4. Choose a state assignment
  5. Derive output equations
  6. Derive flip-flop excitation equations
- ▶ **Synthesis steps 2-6 can be automated, given the state diagram**



# Synchronous Sequential Circuit Model

---



Mealy Outputs  $z = f(x,y)$ , Moore Outputs  $z = f(y)$

Next State  $Y = f(x,y)$

---



# Synthesizing Finite state machines

---

- ▶ Model states as enumerated type
- ▶ Model state and next\_state
- ▶ One process updates state with next\_state
- ▶ One process updates next\_state
- ▶ Allow synthesis tool to encode states  
(binary, one-hot, random, gray code, etc.)
- ▶ Consider how initial state will be forced



# State machine synthesis issues

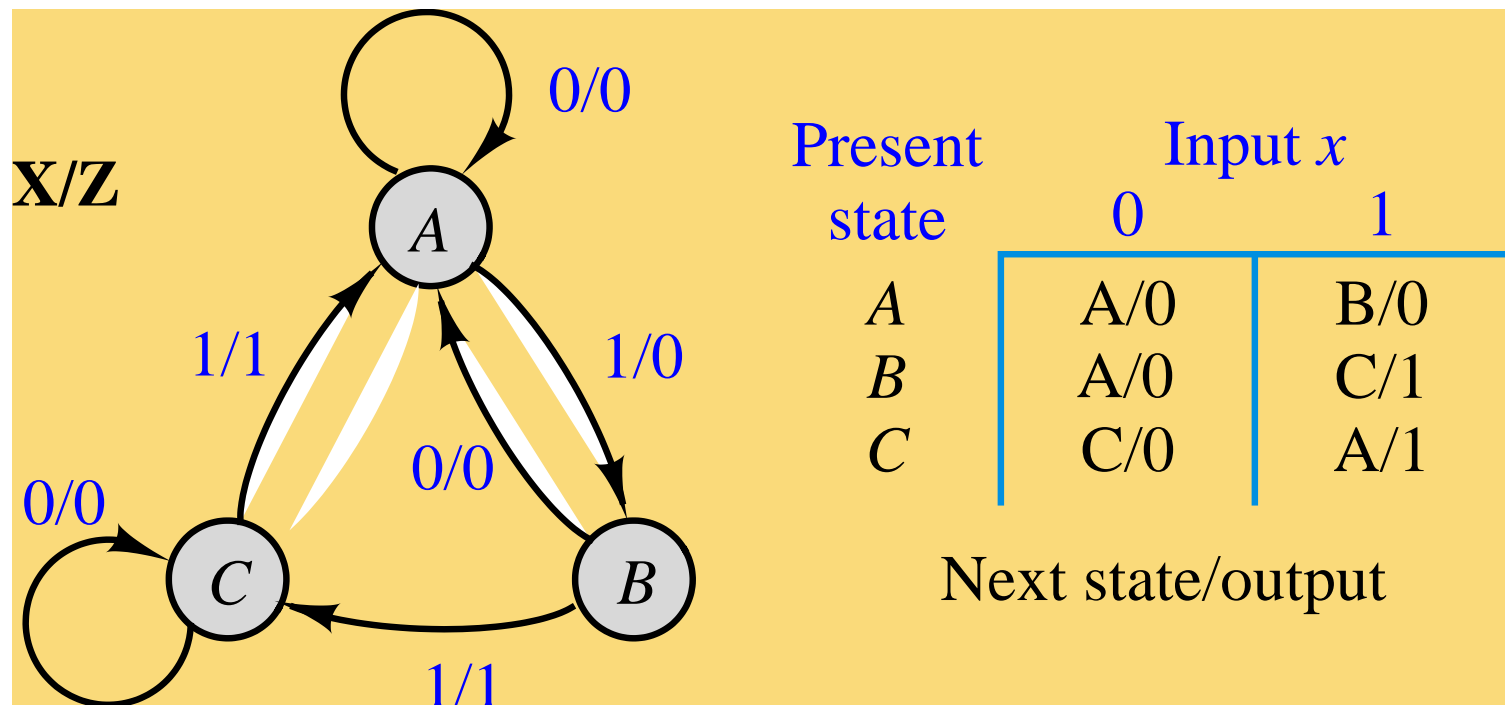
---

- ▶ Mealy model:  $\text{outputs} = f(\text{inputs}, \text{state})$
- ▶ Moore model:  $\text{outputs} = f(\text{state})$
- ▶ “case” more efficient than “if-then-elsif...” to test `present_state` (latter builds a priority encoder)
- ▶ “others” in case statement could also generate extra logic (must determine if not one of the other states)
- ▶ Assign outputs & `next_state` for every condition. Any signal not assigned anything should retain its value. If an output or `next_state` is not assigned something under a certain condition in the case statement, the synthesis tools will have to preserve the value with extra latches/flip-flops.
- ▶ Left-most value of enumeration type is default simulation starting value (use reset to initialize real circuit)



# Synchronous Sequential Circuit (FSM) Example

---



# FSM Example – entity definition

---

entity seqckt is

port (

x: in bit; -- FSM input

z: out bit; -- FSM output

clk: in bit ); -- clock

end seqckt;



# FSM Example - behavioral model

---

architecture behave of seqckt is

```
type states is (A,B,C); -- symbolic state names (enumerate)
```

```
signal curr_state,next_state: states;
```

```
begin
```

```
-- Model the memory elements of the FSM
```

```
process (clk)
```

```
begin
```

```
    if (clk'event and clk='1') then
```

```
        pres_state <= next_state;
```

```
    end if;
```

```
end process;
```

```
(continue on next slide)
```



# FSM Example - continued

---

-- Model next-state and output functions of the FSM

process (x, pres\_state) -- function inputs

begin

    case pres\_state is -- describe each state

        when A => if (x = '0') then

            z <= '0';

            next\_state <= A;

        else -- if (x = '1')

            z <= '0';

            next\_state <= B;

        end if;

(continue next slide for pres\_state = B and C)

---



# FSM Example (continued)

---

```
when B => if (x='0') then
    z <= '0';
    next_state <= A;
else
    z <= '1';
    next_state <= C;
end if;
```

```
when C => if (x='0') then
    z <= '0';
    next_state <= C;
else
    z <= '1';
    next_state <= A;
end if;
```

```
end case;
end process;
```

---



# Alternative Format for Output and Next State Functions

---

## -- Output function

```
z <= '1' when ((curr_state = B) and (x = '1'))  
             or ((curr_state = C) and (x = '1'))  
             else '0';
```

## -- Next state function

```
next_state <= A when ((curr_state = A) and (x = '0'))  
                 or ((curr_state = B) and (x = '0'))  
                 or ((curr_state = C) and (x = '1')) else  
                 B when ((curr_state = I) and (x = '1')) else  
                 C;
```



```

library IEEE; use IEEE.STD_LOGIC_1164.all;
entity SMI is
    port (aln, clk : in Std_logic;  yOut: out Std_logic);
end SMI;
architecture Moore of SMI is
    type state is (s1, s2, s3, s4);
    signal pS, nS : state;
begin
    process (aln, pS) begin -- next state and output functions
        case pS is
            when s1 => yOut <= '0'; nS <= s4; --Moore: yOut = f(pS)
            when s2 => yOut <= '1'; nS <= s3;
            when s3 => yOut <= '1'; nS <= s1;
            when s4 => yOut <= '1'; nS <= s2;
        end case;
    end process;
    process begin
        wait until clk = '1';
        pS <= nS; -- update state variable on next clock
    end process;
end Moore;

```



```

library IEEE; use IEEE.STD_LOGIC_1164.all;
entity SM2 is port (aln, clk : in Std_logic; yOut: out Std_logic); end SM2;
architecture Mealy of SM2 is
    type state is (s1, s2, s3, s4);
    signal pS, nS : state;
begin
    process(aln, pS) begin -- Mealy: yOut & nS are functions of aln and pS
        case pS is
            when s1 => if (aln = '1') then yOut <= '0'; nS <= s4;
                        else yOut <= '1'; nS <= s3;    end if;
            when s2 => yOut <= '1'; nS <= s3;
            when s3 => yOut <= '1'; nS <= s1;
            when s4 => if (aln = '1') then yOut <= '1'; nS <= s2;
                        else yOut <= '0'; nS <= s1;    end if;
        end case; end process;
    process begin
        wait until clk = '1' ;
        pS <= nS;
    end process;
end Mealy;

```



```
when s1 => -- initiate row access
    ras <= '0' ; cas <= '1' ; ready <= '0' ;
    next_state <= s2 ;
when s2 => -- initiate column access
    ras <= '0' ; cas <= '0' ; ready <= '0' ;
    if (cs = '0') then
        next_state <= s0 ; -- end of operation if cs = 0
    else
        next_state <= s2 ; -- wait in s2 for cs = 0
    end if ;
when s3 => -- start cas-before-ras refresh
    ras <= '1' ; cas <= '0' ; ready <= '0' ;
    next_state <= s4 ;
when s4 => -- complete cas-before-ras refresh
    ras <= '0' ; cas <= '0' ; ready <= '0' ;
    next_state <= s0 ;
end case ;
end process ;
end rtl ;
```

---

