VHDL:

Sequential Logic Modeling

Goals

- To understand the use of VHDL in modeling sequential logic.
- To learn the different VHDL models for finite state machines.
- To know how to model the different sequential elements: counters, parallel to serial converters, and registers.
- To learn how to model D flip-flops.
- To get familiar with flip-flop basic delay times.
- To get familiar with datapath concepts.
- To get an introduction to the concept clock skew.
- To learn how to model the different types of memories (ROMs and RAMs).

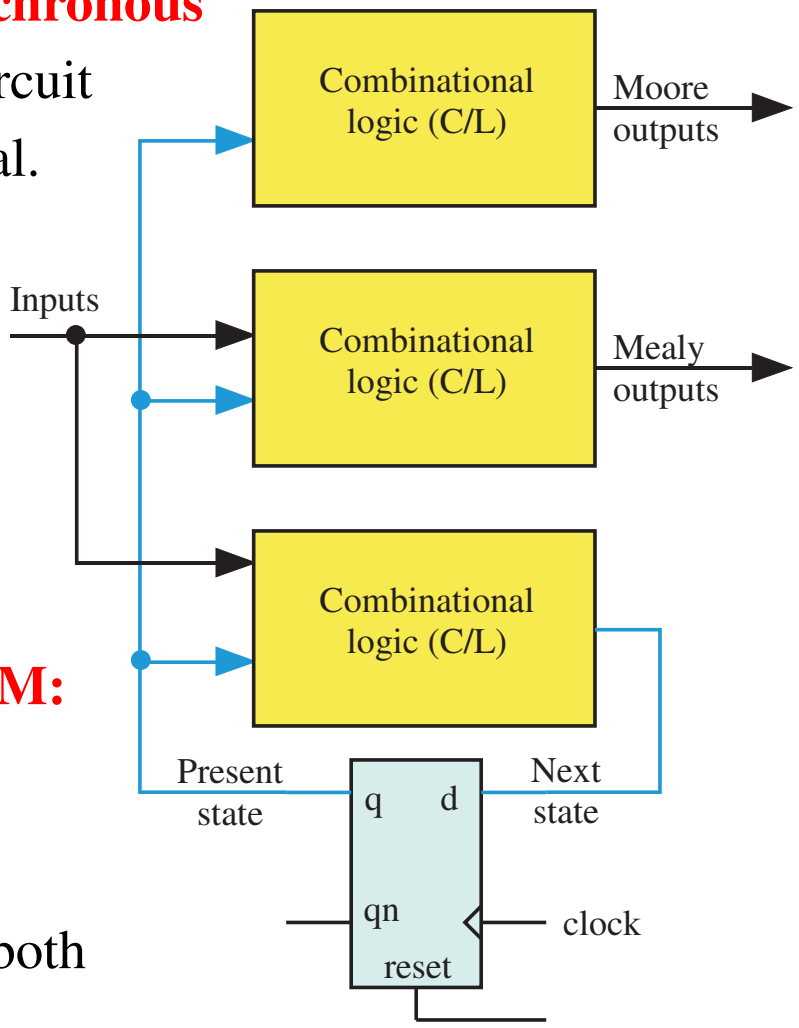
Sequential Circuits

- The output of a **combinational circuit** depends only on the inputs.
- The output of a **sequential circuit** depends on the inputs and the **current state** of the circuit.
- A sequential value thus is able to **store** digital data in special **storage elements**.

- Sequential circuits can be classified by their function into three main types:
  1. Finite State Machines (**FSM**) or (**simple controllers**).
  2. Storage or **memory elements**.
  3. Other types of specialized circuits, such as counters, etc.

Finite State Machines (FSMs)

- We are going to discuss only **synchronous** FSMs where all changes in the circuit occur on the edge of a clock signal.
- There are two types of FSM outputs:
  - **Moore (state-based) FSM:** The output depends only on the present state.
  - **Mealy (transition-based) FSM:** The output depends on the present state and on inputs.
- A single FSM could have one or both of these output types.



Finite State Machines (FSMs) (Cont.)

- An FSM is described by its state diagram or state table.
- A clock input is assumed for synchronous FSM.
- A reset input is assumed for each FSM.
- Transitions between states occur **after** a clock edge when the state register is updated.
- Every state must have an outgoing transition for every input combination. As a convention, if an input combination is ignored at a certain state, it is considered a don't care input.
- Depending on the FSM type, outputs are either associated to states or transitions outgoing from them. Missing outputs should be set to zeros.

Modeling Finite-State Machines (FSMs) (Cont.)

- Four issues must be decided upon when modeling an FSM in VHDL:
  1. Moore or Mealy type outputs
  2. VHDL coding style
  3. Resets and fail safe operation
  4. State encoding
- VHDL description of FSM requires four tasks:
  1. Asynchronous reset (RS)
  2. Update current state register (CS)
  3. Determine next state (NS)
  4. Determine outputs (OP)
- The architecture of a FSM is done using one, two, or three processes:
  - One process is not a common option.
  - It is best to model an FSM using two or three processes.

Modeling Finite-State Machines (FSMs) (Cont.)

No. of processes		One	Two		Three		
Process No.		P 1	P 1	P 2	P 1	P 2	P 3
Tasks	RS	√	√		√		
	CS	√	√		√		
	NS	√		√		√	
	OP	√		√			√
Sensitive to	Mealy	Clock Reset CSs Inputs	Clock Reset	CSs Inputs	Clock Reset	CSs Inputs	CSs Inputs
	Moore	Clock Reset CSs	Clock Reset	CSs Inputs	Clock Reset	CSs Inputs	CSs

Note: Asynchronous reset.

Modeling Finite-State Machines (FSMs) (Cont.)

- FSM must be initialized to a known state before first clock cycle.
  - Use asynchronous reset to ensure that FSM is initialized to a known state before first clock transition.
  - Asynchronous reset is modelled using `IF` statement.
  - Synchronous reset is modelled using `IF` or `WAIT` statements.
- Current state update could use `IF` or `WAIT` to monitor the rising edge of the clock.
- Next state logic is done using `CASE` statement.
- Any CASE statement must handle all possible input combinations.
- Any CASE statement must cover all possible state values to control the behavior of the state machine in unreachable states.

FSM State Definition

- Defining the states as a new `TYPE` that could be declared as follows:
  1. In a PACKAGE,
  2. In an ENTITY, or
  3. In the declaration part of an ARCHITECTURE
- Defining the states as a new `TYPE` could be declared as:
  1. Enumerated data type,
  2. A constant representing each state, or
  3. An integer type
- State encoding can be defined by the programmer or the synthesis tool. Programmers could totally ignore state encoding.
- State encoding methods: sequential, Gray, Johnson, one-hot, etc.

FSM State Definition (Cont.)

No.	Sequential	Gray	Johnson	One-Hot
0	0000	0000	0000_0000	0000_0000_0000_0001
1	0001	0001	0000_0001	0000_0000_0000_0010
2	0010	0011	0000_0011	0000_0000_0000_0100
3	0011	0010	0000_0111	0000_0000_0000_1000
4	0100	0110	0000_1111	0000_0000_0001_0000
5	0101	0111	0001_1111	0000_0000_0010_0000
6	0110	0101	0011_1111	0000_0000_0100_0000
7	0111	0100	0111_1111	0000_0000_1000_0000
8	1000	1100	1111_1111	0000_0001_0000_0000
9	1001	1101	1111_1110	0000_0010_0000_0000
10	1010	1111	1111_1100	0000_0100_0000_0000
11	1011	1110	1111_1000	0000_1000_0000_0000
12	1100	1010	1111_0000	0001_0000_0000_0000
13	1101	1011	1110_0000	0010_0000_0000_0000
14	1110	1001	1100_0000	0100_0000_0000_0000
15	1111	1000	1000_0000	1000_0000_0000_0000

FSM State Definition (Cont.)

Example: 1 (Class) *Show how to define 8 states in a package using an enumerated data type. Use Gray encoding for states.*

```
PACKAGE enumerated_gray_encoding IS

   ATTRIBUTE enum_enconding:  string;

   TYPE state IS
      (ang_0, ang_1, ang_2, ang_3, ang_4, ang_5, ang_6, ang_7);

   ATTRIBUTE enum_enconding OF state:  TYPE IS
      "000 001 011 010 110 111 101 100";

END PACKAGE enumerated_gray_encoding;
```

FSM State Definition (Cont.)

Example: 2 (Class) *Show how to define 8 states in a package using a constant for each state. Use Johnson state encoding for states.*

```
LIBRARY ieee;
USE ieee.std_logic_1164.ALL;
USE ieee.numeric_std.ALL;

PACKAGE const_johnson_encoding IS
   CONSTANT ang_0:  unsigned (3 DOWNTO 0) := "0000";
   CONSTANT ang_1:  unsigned (3 DOWNTO 0) := "0001";
   CONSTANT ang_2:  unsigned (3 DOWNTO 0) := "0011";
   CONSTANT ang_3:  unsigned (3 DOWNTO 0) := "0111";
   CONSTANT ang_4:  unsigned (3 DOWNTO 0) := "1111";
   CONSTANT ang_5:  unsigned (3 DOWNTO 0) := "1110";
   CONSTANT ang_6:  unsigned (3 DOWNTO 0) := "1100";
   CONSTANT ang_7:  unsigned (3 DOWNTO 0) := "1000";
END PACKAGE const_johnson_encoding;
```

Modeling Finite-State Machines (FSMs) (Cont.)

Example: 3 (Class) *Show how to define 8 states in a package using an integer for each state.*

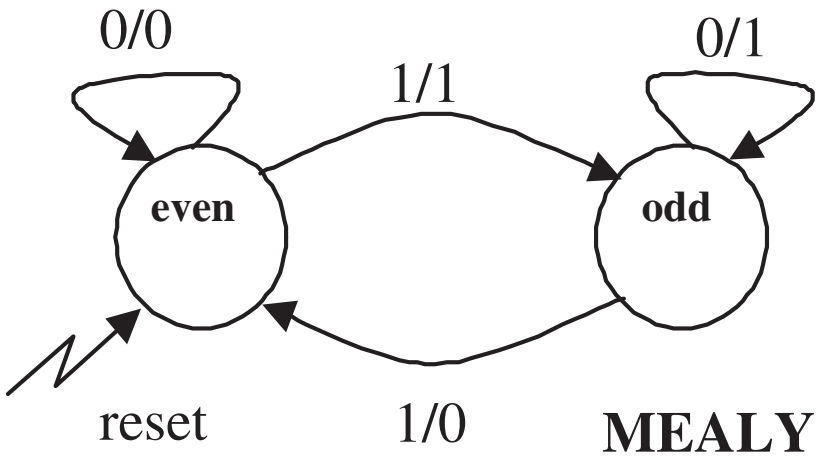
```
PACKAGE integer_encoding IS

   SUBTYPE state IS integer RANGE 0 TO 7;

END PACKAGE integer_encoding;
```

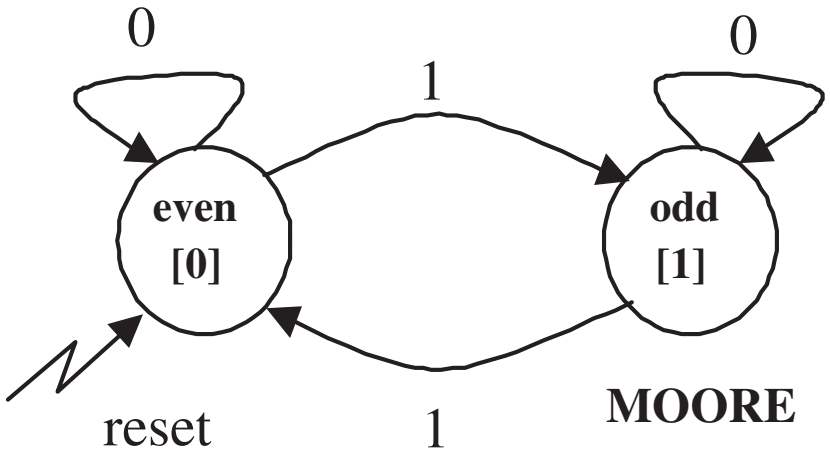
FSM Coding Style

Example: 4 (Class) *We want to model an odd parity checker as an FSM using VHDL.*




```
LIBRARY ieee;
USE ieee.std_logic_1164.ALL;

ENTITY fsm IS
   PORT( clk, reset:  IN std_logic;
         x:  IN std_logic;
         y:  OUT std_logic);
END ENTITY fsm;
```



FSM Coding Style: One Process Mealy

Example 4: (Cont.)

```
ARCHITECTURE mealy_1p OF fsm IS
   TYPE state_type IS (even, odd);
   SIGNAL current_state:  state_type;
BEGIN
   same:  PROCESS (clk, reset, current_state, x) BEGIN
      IF reset = '1' THEN
         current_state <= even;
      ELSIF rising_edge (clk) THEN
         CASE current_state IS
            WHEN even =>
               IF x = '1' THEN
                  current_state <= odd;
               ELSE
                  current_state <= even;
               END IF;
            WHEN odd =>
               IF x = '1' THEN
                  current_state <= even;
               ELSE
         .
         .
         .
```

FSM Coding Style: One Process Mealy (Cont.)

Example 4: (Cont.)

```
          :
          :
                    current_state <= odd;
                END IF;
            END CASE;
        END IF;
        CASE current_state IS
            WHEN even =>
                IF x = '1' THEN
                    y <= '1';
                ELSE
                    y <= '0';
                END IF;
            WHEN odd =>
                IF x = '1' THEN
                    y <= '0';
                ELSE
                    y <= '1';
                END IF;
        END CASE;
    END PROCESS same;
END ARCHITECTURE mealy_1p;
```

FSM Coding Style: Two Processes Mealy

Example 4: (Cont.)

```
ARCHITECTURE mealy_2p OF fsm IS
   TYPE state_type IS (even, odd);
   SIGNAL current_state:  state_type;
   SIGNAL next_state:  state_type;
BEGIN
   cs:  PROCESS (clk, reset)
   BEGIN
      IF reset = '1' THEN
         current_state <= even;
      ELSIF rising_edge (clk) THEN
         current_state <= next_state;
      END IF;
   END PROCESS cs;
         .
         .
         .
```

FSM Coding Style: Two Processes Mealy (Cont.)

Example 4: (Cont.)

```
        :
   ns:  PROCESS (current_state, x) BEGIN
      CASE current_state IS
         WHEN even =>
            IF x = '1' THEN
               y <= '1';      next_state <= odd;
            ELSE
               y <= '0';      next_state <= even;
            END IF;
         WHEN odd =>
            IF x = '1' THEN
               y <= '0';      next_state <= even;
            ELSE
               y <= '1';      next_state <= odd;
            END IF;
      END CASE;
   END PROCESS ns;
END ARCHITECTURE mealy_2p;
```

FSM Coding Style: Three Processes Mealy

Example 4: (Cont.)

```
ARCHITECTURE mealy_3p OF fsm IS
   TYPE state_type IS (even, odd);
   SIGNAL current_state:  state_type;
   SIGNAL next_state:  state_type;
BEGIN
   cs:  PROCESS (clk, reset)
   BEGIN
      IF reset = '1' THEN
         current_state <= even;
      ELSIF rising_edge (clk) THEN
         current_state <= next_state;
      END IF;
   END PROCESS cs;
         .
         .
         .
```

FSM Coding Style: Three Processes Mealy (Cont.)

Example 4: (Cont.)

```
        :
        :
ns:  PROCESS (current_state, x)
BEGIN
   CASE current_state IS
      WHEN even =>
         IF x = '1' THEN
            next_state <= odd;
         ELSE
            next_state <= even;
         END IF;
      WHEN odd =>
         IF x = '1' THEN
            next_state <= even;
         ELSE
            next_state <= odd;
         END IF;
   END CASE;
END PROCESS ns;
        :
        :
```

FSM Coding Style: Three Processes Mealy (Cont.)

Example 4: (Cont.)

```
        .
        .
        .
   op:  PROCESS (current_state, x)
   BEGIN
      CASE current_state IS
         WHEN even =>
            IF x = '1' THEN
               y <= '1';
            ELSE
               y <= '0';
            END IF;
         WHEN odd =>
            IF x = '1' THEN
               y <= '0';
            ELSE
               y <= '1';
            END IF;
      END CASE;
   END PROCESS OP;
END ARCHITECTURE mealy_3p;
```

FSM Coding Style: One Process Moore

Example 4: (Cont.)

```
ARCHITECTURE moore_1p OF fsm IS
   TYPE state_type IS (even, odd);
   SIGNAL current_state:  state_type;
BEGIN
   same:  PROCESS (clk, reset, current_state)
   BEGIN
      IF reset = '1' THEN
         current_state <= even;
      ELSIF rising_edge (clk) THEN
         CASE current_state IS
            WHEN even =>
               IF x = '1' THEN
                  current_state <= odd;
               ELSE
                  current_state <= even;
               END IF;
         :
         :
```

FSM Coding Style: One Process Moore (Cont.)

Example 4: (Cont.)

```
        .
        .
        .
            WHEN odd =>
                IF x = '1' THEN
                    current_state <= even;
                ELSE
                    current_state <= odd;
                END IF;
            END CASE;
        END IF;
        CASE current_state IS
            WHEN even =>
                y <= '0';
            WHEN odd =>
                y <= '1';
        END CASE;
    END PROCESS same;
END ARCHITECTURE moore_1p;
```

FSM Coding Style: Two Processes Moore

Example 4: (Cont.)

```
ARCHITECTURE moore_2p OF fsm IS
   TYPE state_type IS (even, odd);
   SIGNAL current_state:  state_type;
   SIGNAL next_state:  state_type;
BEGIN
   cs:  PROCESS (clk, reset)
   BEGIN
      IF reset = '1' THEN
         current_state <= even;
      ELSIF rising_edge (clk) THEN
         current_state <= next_state;
      END IF;
   END PROCESS cs;
         .
         .
         .
```

FSM Coding Style: Two Processes Moore (Cont.)

Example 4: (Cont.)

```
        :
        :
    ns:  PROCESS (current_state, x)
    BEGIN
       CASE current_state IS
          WHEN even =>
             y <= '0';
             IF x = '1' THEN
                next_state <= odd;
             ELSE
                next_state <= even;
             END IF;
          WHEN odd =>
             y <= '1';
             IF x = '1' THEN
                next_state <= even;
             ELSE
                next_state <= odd;
             END IF;
       END CASE;
    END PROCESS ns;
END ARCHITECTURE moore_2p;
```

FSM Coding Style: Three Processes Moore

Example 4: (Cont.)

```
ARCHITECTURE moore_3p OF fsm IS
   TYPE state_type IS (even, odd);
   SIGNAL current_state:  state_type;
   SIGNAL next_state:  state_type;
BEGIN
   cs:  PROCESS (clk, reset)
   BEGIN
      IF reset = '1' THEN
         current_state <= even;
      ELSIF rising_edge (clk) THEN
         current_state <= next_state;
      END IF;
   END PROCESS cs;
         .
         .
         .
```

FSM Coding Style: Three Processes Moore (Cont.)

Example 4: (Cont.)

```
        :
        :
ns:  PROCESS (current_state, x)
BEGIN
   CASE current_state IS
      WHEN even =>
         IF x = '1' THEN
            next_state <= odd;
         ELSE
            next_state <= even;
         END IF;
      WHEN odd =>
         IF x = '1' THEN
            next_state <= even;
         ELSE
            next_state <= odd;
         END IF;
   END CASE;
END PROCESS ns;
        :
        :
```

FSM Coding Style: Three Processes Moore (Cont.)

Example 4: (Cont.)

```
          :
   op:  PROCESS (current_state)
   BEGIN
      CASE current_state IS
         WHEN even =>
            y <= '0';
         WHEN odd =>
            y <= '1';
      END CASE;
   END PROCESS op;
END ARCHITECTURE moore_3p;
```

FSM Coding Style: Sample Testbench

Example 4: (Cont.)

```
LIBRARY ieee;
USE ieee.std_logic_1164.ALL;
ENTITY fsm_tb IS
END ENTITY fsm_tb;
ARCHITECTURE tb_arch OF fsm_tb IS
   COMPONENT fsm IS
      PORT( clk, reset:  IN std_logic;
            x:  IN std_logic;
            y:  OUT std_logic);
   END COMPONENT fsm;
   SIGNAL clk, reset, x, y:  std_logic;
   FOR fsm_mealy_2p:  fsm USE ENTITY work.fsm (mealy_2p);
BEGIN
   clock:  PROCESS IS
   BEGIN
      clk <= '0', '1' AFTER 10 ns;
      WAIT FOR 20 ns;
   END PROCESS clock;
        .
        .
        .
```

FSM Coding Style: Sample Testbench (Cont.)

Example 4: (Cont.)

```
        :
    sg:  PROCESS IS
    BEGIN
       reset <= '1';
       WAIT FOR 10 ns;
       ASSERT y = '0'
          REPORT "Error:  Reset"
          SEVERITY warning;
       reset <= '0';  x <= '0';
       WAIT FOR 10 ns;
       ASSERT y = '0'
          REPORT "Error:  state even when x = 0"
          SEVERITY warning;
       WAIT FOR 10 ns;
       x <= '1';
       WAIT FOR 10 ns;
       ASSERT y = '1'
          REPORT "Error:  state even when x = 1"
          SEVERITY warning;
          :
          :
```

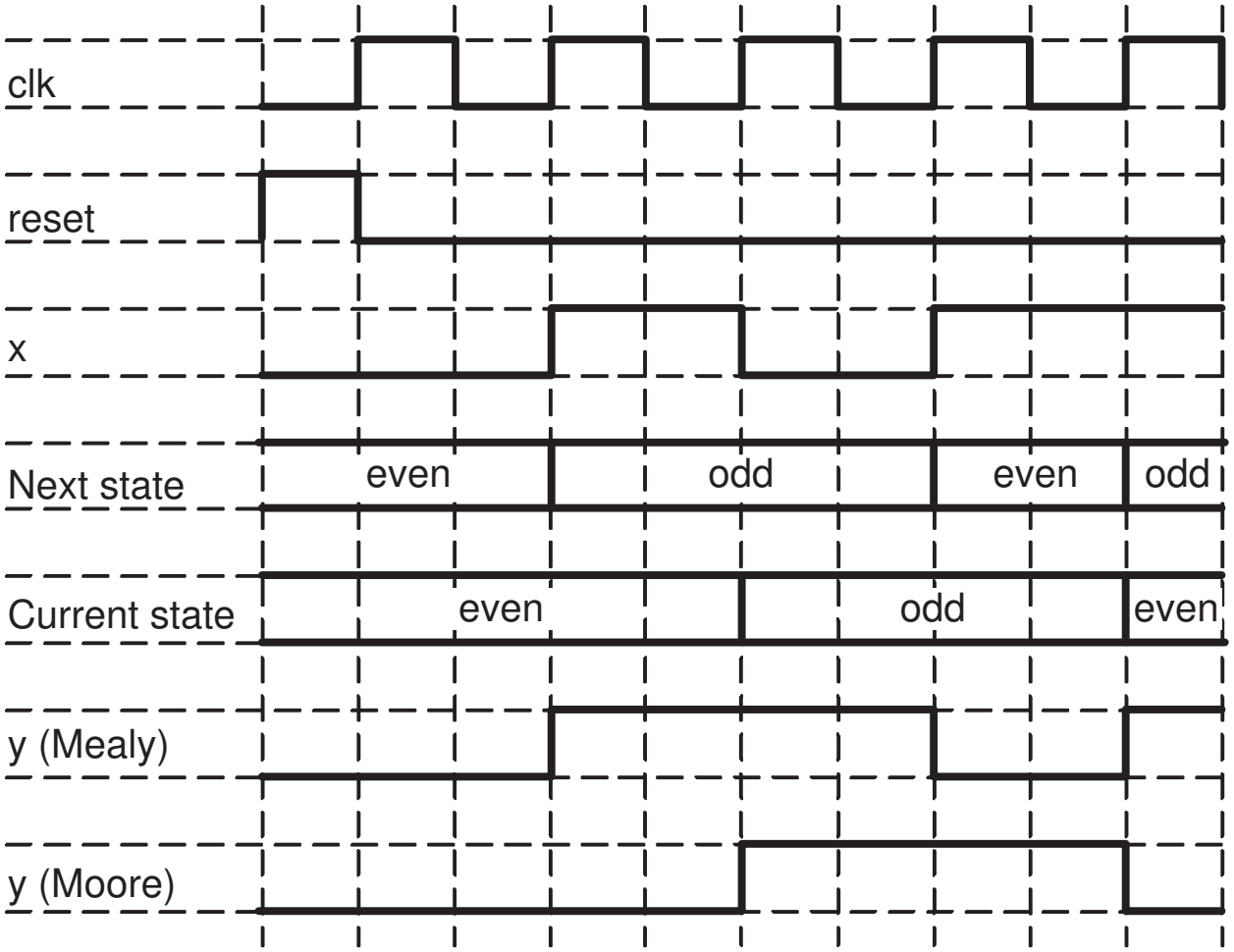
FSM Coding Style: Sample Testbench (Cont.)

Example 4: (Cont.)

```
        ⋮
      WAIT FOR 10 ns;
      x <= '0';
      WAIT FOR 10 ns;
      ASSERT y = '1'
         REPORT "Error:  state odd when x = 0"
         SEVERITY warning;
      WAIT FOR 10 ns;
      x <= '1';
      WAIT FOR 10 ns;
      ASSERT y = '0'
         REPORT "Error:  state odd when x = 1"
         SEVERITY warning;
      WAIT;
   END PROCESS sg;
   fsm_mealy_2p:  fsm PORT MAP (clk, reset, x, y);
END ARCHITECTURE tb_arch;
```

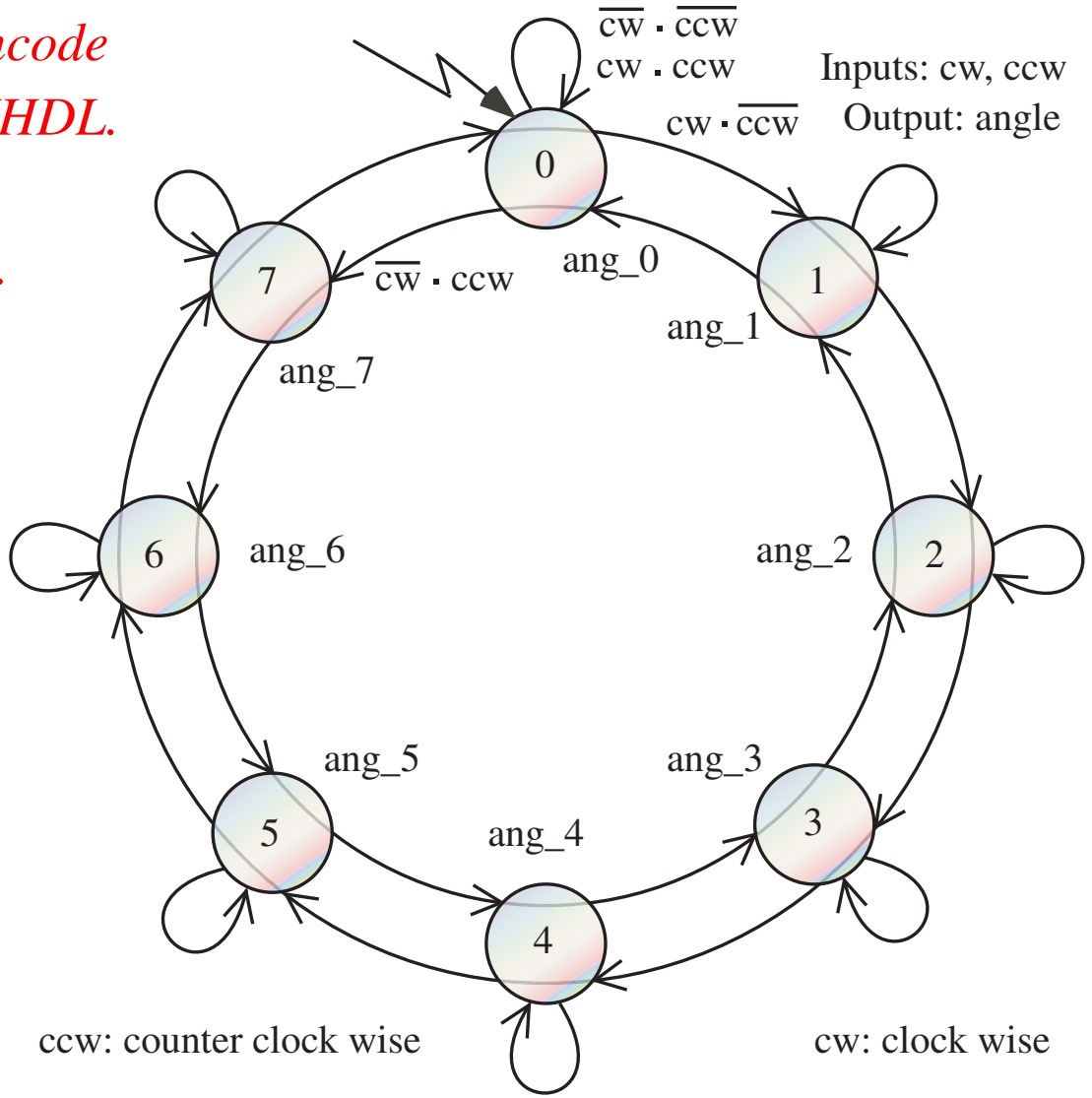
FSM Coding Style: Sample Output

Example 4: (Cont.)



Modeling Finite-State Machines (FSMs) (Cont.)

Example: 5 (Class) *Encode the shown FSM using VHDL. The machine has a synchronous reset input. Use Johnson state encoding for states through the previously defined package. Use a three-process coding style.*



Modeling Finite-State Machines (FSMs) (Cont.)

Example 5: (Cont.)

```
LIBRARY ieee;
USE ieee.std_logic_1164.ALL;
USE ieee.numeric_std.ALL;
USE work.const_johnson_encoding.ALL;
ENTITY angle IS
   PORT( clk, reset, cw, ccw:  IN std_logic;
         angle:  OUT std_logic_vector (2 DOWNTO 0));
END ENTITY angle;
ARCHITECTURE angle OF angle IS
   SIGNAL current_state, next_state:  unsigned (3 DOWNTO 0);
BEGIN -- states have same number of bits as the encoding of ang_0, etc.
   cs:  PROCESS IS BEGIN
      WAIT UNTIL rising_edge (clk);
      IF reset = '1' THEN -- synchronous reset
         current_state <= ang_0;
      ELSE
         current_state <= next_state;
      END IF;
   END PROCESS cs;
            :
            :
```

Modeling Finite-State Machines (FSMs) (Cont.)

Example 5: (Cont.)

```
                :
   ns:  PROCESS (current_state, cw, ccw) IS BEGIN
      CASE current_state IS
         WHEN ang_0 =>
            IF cw = '1' AND ccw = '0' THEN
               next_state <= ang_1;
            ELSIF cw = '0' AND ccw = '1' THEN
               next_state <= ang_7;
            ELSE
               next_state <= ang_0;
            END IF;
         WHEN ang_1 =>
            IF cw = '1' AND ccw = '0' THEN
               next_state <= ang_2;
            ELSIF cw = '0' AND ccw = '1' THEN
               next_state <= ang_0;
            ELSE
               next_state <= ang_1;
            END IF;
               :
               :
```

Modeling Finite-State Machines (FSMs) (Cont.)

Example 5: (Cont.)

```
                :
        WHEN ang_7 =>
            IF cw = '1' AND ccw = '0' THEN
                next_state <= ang_0;
            ELSIF cw = '0' AND ccw = '1' THEN
                next_state <= ang_6;
            ELSE
                next_state <= ang_7;
            END IF;
        WHEN OTHERS =>
            next_state <= ang_0;
    END CASE;
END PROCESS ns;
            :
```

Modeling Finite-State Machines (FSMs) (Cont.)

Example 5: (Cont.)

```
                :
   op:  PROCESS (current_state) IS
   BEGIN
      CASE current_state IS
         WHEN ang_0   => angle <= o"0";
         WHEN ang_1   => angle <= o"1";
         WHEN ang_2   => angle <= o"2";
         WHEN ang_3   => angle <= o"3";
         WHEN ang_4   => angle <= o"4";
         WHEN ang_5   => angle <= o"5";
         WHEN ang_6   => angle <= o"6";
         WHEN ang_7   => angle <= o"7";
         WHEN OTHERS => NULL;
      END CASE;
   END PROCESS op;
END ARCHITECTURE angle;
```

Counters

Example: 6 (Class) *Model an n-bit binary up/down counter with synchronous clear and preset.*

```
LIBRARY ieee;
USE ieee.std_logic_1164.ALL;
USE ieee.numeric_std.ALL;

ENTITY bin_counter IS
   GENERIC( n:  positive := 4);
   PORT( clk, clear, preset, up_down:  IN std_logic;
         d_in:  IN unsigned (n-1 DOWNTO 0);
         d_out:  OUT unsigned (n-1 DOWNTO 0));
END ENTITY bin_counter;
ARCHITECTURE behav OF bin_counter IS
   SIGNAL counter:  unsigned (n-1 DOWNTO 0);
BEGIN
         :
```

Counters (Cont.)

Example 6: (Cont.)

```
          :
   ct:  PROCESS (clk) IS BEGIN
      IF rising_edge (clk) THEN
         IF clear = '1' THEN
            counter <= (OTHERS => '0');
         ELSIF preset = '1' THEN
            counter <= d_in;
         ELSIF up_down = '1' THEN
            counter <= counter + 1;
         ELSE
            counter <= counter - 1;
         END IF;
      END IF;
   END PROCESS ct;
   d_out <= counter;
END ARCHITECTURE behav;
```

Counters (Cont.)

Example: 7 (Class) *Model an n-bit Gray up/down counter with synchronous clear.*

```
LIBRARY ieee;
USE ieee.std_logic_1164.ALL;
USE ieee.numeric_std.ALL;

ENTITY gray_counter IS
   GENERIC( n:  positive := 4);
   PORT( clk, clear, up_down:  IN std_logic;
         d_out:  OUT std_logic_vector (n-1 DOWNTO 0));
END ENTITY gray_counter;
ARCHITECTURE behav OF gray_counter IS
   COMPONENT bin_counter IS
      GENERIC( n:  positive := 4);
      PORT( clk, clear, up_down:  IN std_logic;
            preset:  IN std_logic;
            d_in:  IN unsigned (n-1 DOWNTO 0) := (OTHERS => '0');
            d_out:  OUT unsigned (n-1 DOWNTO 0));
   END COMPONENT bin_counter;
            :
```

Counters (Cont.)

Example 7: (Cont.)

```
          :
   FOR ALL: bin_counter USE ENTITY WORK.bin_counter (behav);

   SIGNAL d_bin:  unsigned (n-1 DOWNTO 0);
BEGIN
   bcount:  bin_counter GENERIC MAP( n => 4)
                        PORT MAP(   clk => clk, clear => clear
                                    up_down => up_down, preset => '0',
                                    d_in => OPEN, d_out => d_bin);
   gray:  PROCESS (d_bin) IS
   BEGIN
      d_out(n-1) <= d_bin(n-1);
      FOR j IN n-2 DOWNTO 0 LOOP
         d_out(j) <= d_bin (j+1) XOR d_bin (j);
      END LOOP;
   END PROCESS gray;
END ARCHITECTURE behav;
```

Notice that if an input port is to be left open, a default value must be specified for it.

Parallel to Serial Converter

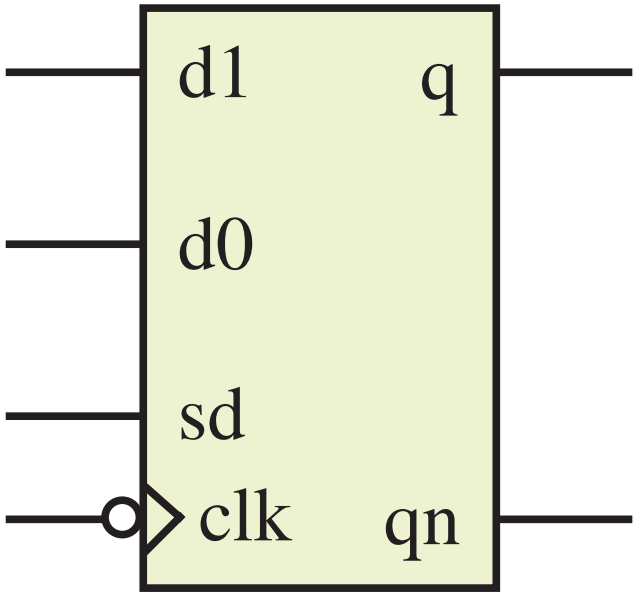
Example: 8 (Class) *A parallel to serial converter that scans out the MSB first.*

```
ENTITY p2s IS
   GENERIC( n:  positive := 4);
   PORT( d_in:  IN bit_vector (0 TO n-1);
         clk, load, shift:  IN bit;
         d_out:  OUT bit);
END ENTITY p2s;
ARCHITECTURE shift OF p2s IS
   SIGNAL save:  bit_vector (0 TO n-1);
BEGIN
   ps:  PROCESS (clk) IS BEGIN
      IF clk = '1' AND clk'event THEN
         IF load = '1' THEN
            save <= d_in;
         ELSIF shift = '1' THEN
            save <= save (1 TO n-1) & '0';
         END IF;
      END IF;
   END PROCESS ps;
   d_out <= save(0);
END ARCHITECTURE shift;
```

Flip-Flops

Example: 9 (Class) *Model a negative edge-triggered D flip-flop with select input.*

```
LIBRARY ieee;
USE ieee.std_logic_1164.ALL;
ENTITY d_ff_sel IS
   PORT( d1, d0, sd, clk:  IN std_logic;
         q, qn:  OUT std_logic);
END ENTITY d_ff_sel;
ARCHITECTURE sel OF d_ff_sel IS
BEGIN
   sl:  PROCESS IS
   BEGIN
      WAIT UNTIL falling_edge (clk);
      IF sd = '1' THEN
         q <= d1;       qn < = NOT d1;
      ELSE
         q <= d0;       qn < = NOT d0;
      END IF;
   END PROCESS sl;
END ARCHITECTURE sel;
```



Flip-Flops (Cont.)

Example: 10 (Class) *Model a positive edge-triggered D flip-flop with asynchronous clear and set.*

```
LIBRARY ieee;
USE ieee.std_logic_1164.ALL;
ENTITY d_ff_async IS
   PORT( clk, clear, set:  IN std_logic;
         d_in:  IN std_logic;
         d_out:  OUT std_logic);
END ENTITY d_ff_async;
ARCHITECTURE async OF d_ff_async IS BEGIN
   asr:  PROCESS (clk, clear, set) IS BEGIN
      IF clear = '1' THEN
         d_out <= '0';
      ELSIF set = '1' THEN
         d_out <= '1';
      ELSIF rising_edge (clk) THEN
         d_out <= d_in;
      END IF;
   END PROCESS asr;
END ARCHITECTURE async;
```

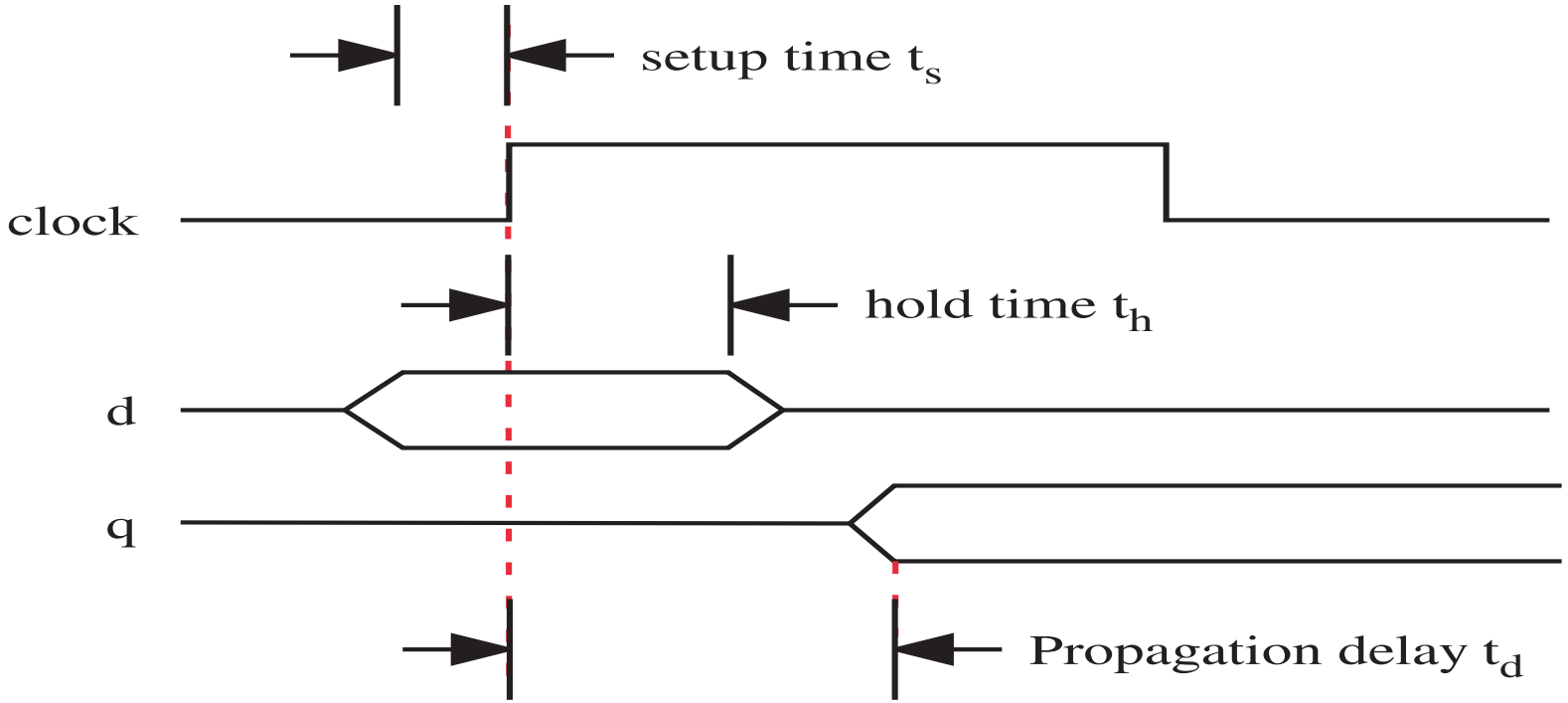
Flip-Flops (Cont.)

Example: 11 (Class) *Repeat Example 10 with synchronous clear and set.*

```
LIBRARY ieee;
USE ieee.std_logic_1164.ALL;
ENTITY d_ff_sync IS
   PORT( clk, clear, set:  IN std_logic;
         d_in:  IN std_logic;
         d_out:  OUT std_logic);
END ENTITY d_ff_sync;
ARCHITECTURE sync OF d_ff_sync IS BEGIN
   ssr:  PROCESS IS BEGIN
      WAIT UNTIL rising_edge (clk);
      IF clear = '1' THEN
         d_out <= '0';
      ELSIF set = '1' THEN
         d_out <= '1';
      ELSE
         d_out <= d_in;
      END IF;
   END PROCESS ssr;
END ARCHITECTURE sync;
```

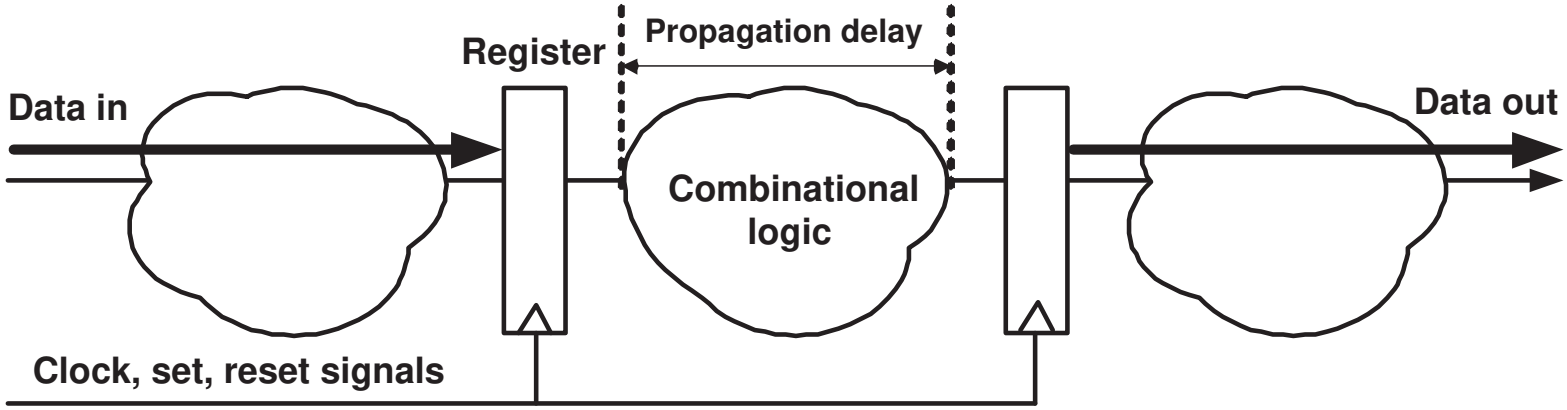
Flip-Flops (Cont.)

- **Setup time t_s:** Minimum time during which data must be stable prior to a clock edge.
- **Hold time t_h:** Minimum time during which input data must be stable after a clock edge. You cannot sample output date during this time. Hold times are usually either 0 or very small.



Flip-Flops (Cont.)

- **Propagation delay t_d:** Time to produce correct result of the circuit measured from the clock edge.
- **Clock skew Tcs:** Arrival of a clock signal at the clock inputs of different flip-flops at different times as a result of propagation delays.
- Given two adjacent flip-flops, i and j, the clock skew between them ($Tcs_{i,j}$) is defined as $Tcs_{i,j} = Tcs_i - Tcs_j$, where Tcs_i and Tcs_j are the clock delays from the clock source to i and j, respectively.



Clock period >= t_d + combinational logic propagation delay + t_s + $2 \times Tcs$

Flip-Flops (Cont.)

Example: 12 (Class) *Model a D flip-flop with setup, hold, and delay times.*

```
ENTITY d_ff IS
   GENERIC( setup, hold:  time := 0.1 ns; delay:  time := 2 ns);
   PORT( d, clk:  IN bit;
         q:  OUT bit);
END ENTITY d_ff;
ARCHITECTURE d_ff OF d_ff IS BEGIN
   q <= d AFTER delay WHEN clk = '1' AND clk'event;
   check_setup:  PROCESS IS BEGIN
      WAIT UNTIL clk = '1' AND clk'event;
      ASSERT d'last_event >= setup     -- Same as d'stable (setup)
         REPORT "setup violation"
         SEVERITY warning;
   END PROCESS check_setup;
   check_hold:  PROCESS IS BEGIN
      WAIT UNTIL clk'delayed (hold) = '1' AND clk'delayed (hold)'EVENT;
      ASSERT d'stable (hold)           -- Same as d'last_event >= hold
         REPORT "hold violation"
         SEVERITY warning;
   END PROCESS check_hold;
END ARCHITECTURE d_ff;
```

Flip-Flops (Cont.)

Example 12: (Cont.)

```
ENTITY dff_test IS
END ENTITY dff_test;
ARCHITECTURE test OF dff_test IS
   COMPONENT d_ff IS
      GENERIC(  delay:  time := 4 ns;
                setup:  time := 3 ns;
                hold:  time := 1 ns);
      PORT(  d, clk:  IN bit;
             q:  OUT bit);
   END COMPONENT d_ff;
   FOR gate:  d_ff USE ENTITY WORK.d_ff (basic);
   SIGNAL d_in, clock, d_out:  bit;
BEGIN
   gate:  d_ff  GENERIC MAP(  delay => 4 ns,
                              setup => 3 ns,
                              hold => 1 ns)
                PORT MAP(  d => d_in,
                           clk => clock,
                           q => d_out);
END ARCHITECTURE test;
```

Registers

Example: 13 (Class) *Model a shift register to perform the operations in the table below. A zero asynchronous clr signal resets all output bits.*

Function	Control inputs (s1, s0)	q[3]	q[2]	q[1]	q[0]
Hold value	0 0	q[3]	q[2]	q[1]	q[0]
Shift left	0 1	q[2]	q[1]	q[0]	r_in
Shift right	1 0	l_in	q[3]	q[2]	q[1]
Load	1 1	d[3]	d[2]	d[1]	d[0]

```
LIBRARY ieee;
USE ieee.std_logic_1164.ALL;
ENTITY sr IS
   GENERIC( width:  positive := 4);
   PORT( clk, clr, l_in, r_in, s0, s1:  IN std_logic;
         d:  IN std_logic_vector (width-1 DOWNTO 0);
         q:  INOUT std_logic_vector (width-1 DOWNTO 0));
END ENTITY sr;
```

Registers (Cont.)

Example 13: (Cont.)

```
          :
ARCHITECTURE sr OF sr IS
BEGIN
   shift:  PROCESS (clk, clr) IS
      VARIABLE sel:  std_logic_vector (1 DOWNTO 0);
   BEGIN
      IF clr = '0' THEN
         q <= (OTHERS => '0');
      ELSIF rising_edge (clk) THEN
         sel := s1 & s0;
         CASE sel IS
            WHEN "00" => NULL;
            WHEN "01" => q <= q(width-2 DOWNTO 0) & r_in;
            WHEN "10" => q <= l_in & q(width-1 DOWNTO 1);
            WHEN "11" => q <= d;
            WHEN OTHERS => NULL;
         END CASE;
      END IF;
   END PROCESS shift;
END ARCHITECTURE sr;
```

ROMs

Example: 14 (Class) *Show how to use a ROM component that was designed using standard cells.*

```
        :
   COMPONENT ROM IS
      PORT(  en:  IN std_logic;
             address:  IN unsigned (m-1 DOWNTO 0);
             q:  OUT std_logic_vector (n-1 DOWNTO 0));
   END COMPONENT ROM;
   SIGNAL enable:  std_logic;
   SIGNAL address_bus:  unsigned (m-1 DOWNTO 0);
   SIGNAL data_bus:  std_logic_vector (n-1 DOWNTO 0);
BEGIN
   r1:  ROM( en => enable, address => address_bus, q => data_bus);
        :
```

ROMs (Cont.)

Example: 15 (Class) *Use a ROM to build a squaring circuit. Use a flag to ensure that the ROM is initialize only one time.*

```
LIBRARY ieee;
USE ieee.std_logic_1164.ALL;
USE ieee.numeric_std.ALL;

ENTITY rom IS
   GENERIC(  n:  integer := 3;
             m:  integer := 6);
   PORT( enable:  IN std_logic;
         address:  IN unsigned (n-1 DOWNTO 0);
         data:  OUT unsigned (m-1 DOWNTO 0));
END ENTITY rom;
          .
          .
          .
```

ROMs (Cont.)

Example 15: (Cont.)

```
          :
          :
ARCHITECTURE flag_arch OF rom IS
   TYPE rm IS ARRAY (0 TO 2**n-1) OF unsigned (m-1 DOWNTO 0);
   SIGNAL word:  rm;
   SIGNAL initialized:  boolean := false;
BEGIN
   memory:  PROCESS (enable, address) IS BEGIN
      IF NOT initialized THEN
         FOR index IN word'range LOOP
             word(index) <= to_unsigned(index*index, m);
         END LOOP;
         ASSERT FALSE REPORT "ROM init in flag_arch" SEVERITY note;
         initialized <= true;
      END IF;
      IF enable = '1' THEN
         data <= word(to_integer(address));
      ELSE
         data <= (OTHERS => 'Z');
      END IF;
   END PROCESS memory;
END ARCHITECTURE flag_arch;
```

ROMs (Cont.)

Example: 16 (Class) *Repeat Example 15 using a LOOP statement.*

```
ARCHITECTURE loop_arch OF rom IS
   TYPE rm IS ARRAY (0 TO 2**n-1) OF unsigned (m-1 DOWNTO 0);
   SIGNAL word:  rm;
BEGIN
   memory:  PROCESS IS
BEGIN
      FOR index IN word'range LOOP
         word(index) <= to_unsigned(index*index, m);
      END LOOP;
      ASSERT FALSE REPORT "ROM init in loop_arch" SEVERITY note;
      LOOP
         WAIT ON enable, address;
         IF enable = '1' THEN
             data <= word(to_integer(address));
         ELSE
             data <= (OTHERS => 'Z');
         END IF;
      END LOOP;
   END PROCESS memory;
END ARCHITECTURE loop_arch;
```

ROMs (Cont.)

Example: 17 (Class) *Rewrite the architecture in Example 15 using a separate process without a sensitivity list to initialize the ROM.*

```
ARCHITECTURE init_process_arch OF rom IS
   TYPE rm IS ARRAY (0 TO 2**n-1) OF unsigned (m-1 DOWNTO 0);
   SIGNAL word:  rm;
   SIGNAL initialized:  boolean := false;
BEGIN
   init_wait_proc:  PROCESS IS
   BEGIN
      FOR index IN word'range LOOP
         word(index) <= to_unsigned(index*index, m);
      END LOOP;
      ASSERT FALSE REPORT "ROM init in init_wait_proc" SEVERITY note;
      initialized <= true;
      WAIT;
   END PROCESS init_wait_proc;
         :
         :
```

ROMs (Cont.)

Example 17: (Cont.)

```
        :
        :
   memory:  PROCESS (initialized, enable, address) IS
   BEGIN
      IF initialized THEN
         IF enable = '1' THEN
             data <= word(to_integer(address));
         ELSE
             data <= (OTHERS => 'Z');
         END IF;
      END IF;
   END PROCESS memory;
END ARCHITECTURE init_process_arch;
```

ROMs (Cont.)

Example: 18 (Class) *Rewrite the initializing process in Example 17 using a sensitivity list.*

```
        :
init_sens_proc:  PROCESS (initialized) IS
BEGIN
   IF NOT initialized THEN
      FOR index IN word'range LOOP
          word(index) <= to_unsigned(index*index, m);
      END LOOP;
      ASSERT FALSE REPORT "ROM init in init_sens_proc" SEVERITY note;

      initialized <= true;
   END IF;
END PROCESS init_sens_proc;
        :
```

Note: This process will actually run twice. But, the ROM itself will be initialized only once.

ROMs (Cont.)

Example: 19 (Class) *Repeat Example 15, initializing the ROM with a function.*

```
ARCHITECTURE funct_arch OF rom IS
   TYPE rm IS ARRAY (0 TO 2**n-1) OF unsigned (m-1 DOWNTO 0);
   FUNCTION rom_fill RETURN rm IS
      VARIABLE memory:  rm;
   BEGIN
      FOR index IN memory'range LOOP
         memory(index) := to_unsigned(index*index, m);
      END LOOP;
      ASSERT FALSE REPORT "ROM init in funct_arch" SEVERITY note;
      RETURN memory;
   END FUNCTION rom_fill;
   constant word:  rm := rom_fill;
BEGIN
   memory:  PROCESS (enable, address) IS BEGIN
      IF enable = '1' THEN
         data <= word(to_integer(address));
      ELSE
         data <= (OTHERS => 'Z');
      END IF;
   END PROCESS memory;
END ARCHITECTURE funct_arch;
```

ROMs (Cont.)

Example: 20 (Class) *Rewrite the function in Example 19, initializing the ROM from a text file.*

```
LIBRARY ieee;
USE std.textio.ALL;
USE ieee.std_logic_textio.ALL;
          :
          :
   IMPURE FUNCTION rom_fill RETURN rm IS
      VARIABLE memory:  rm;
      FILE f:  text OPEN READ_MODE IS "rom.txt";
      VARIABLE l:  line;
      VARIABLE read_word:  std_logic_vector (m-1 DOWNTO 0);
   BEGIN
      FOR index IN memory'range LOOP
         readline (f, l);
         read (l, read_word);  -- cannot read unsigned values from a file
         memory(index) := unsigned(read_word);
      END LOOP;
      ASSERT FALSE REPORT "ROM init in funct_arch & file" SEVERITY note;
      RETURN memory;
   END FUNCTION rom_fill;
```

ROMs (Cont.)

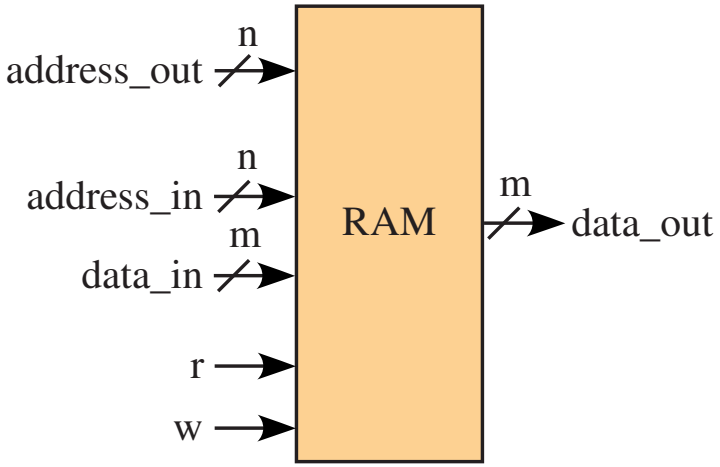
Example: 21 (Class) *Repeat Example 15, initializing the ROM using a generate statement.*

```
ARCHITECTURE gen_arch OF rom IS
   TYPE rm IS ARRAY (0 TO 2**n-1) OF unsigned (m-1 DOWNTO 0);
   SIGNAL word:  rm;
BEGIN
   contents:  FOR index IN word'range GENERATE
      word(index) <= to_unsigned(index*index, m);
   END GENERATE contents;

   data <=   word(to_integer(address)) WHEN enable = '1' ELSE
             (OTHERS => 'Z');
END ARCHITECTURE gen_arch;
```

RAMs

Example: 22 (Class) *Show how to model an n-bit address, m-bit word size, dual-port RAM.*




```
LIBRARY ieee;
USE ieee.std_logic_1164.ALL;
USE ieee.numeric_std.ALL;
ENTITY dual_port_ram IS
   GENERIC(  n:  positive := 4;
             m:  positive := 4);
   PORT( r, w:  IN std_logic;
         address_in:  IN unsigned (n-1 DOWNTO 0);
         address_out:  IN unsigned (n-1 DOWNTO 0);
         data_in:  IN std_logic_vector (m-1 DOWNTO 0);
         data_out:  OUT std_logic_vector (m-1 DOWNTO 0));
END ENTITY dual_port_ram;
         :
```

RAMs (Cont.)

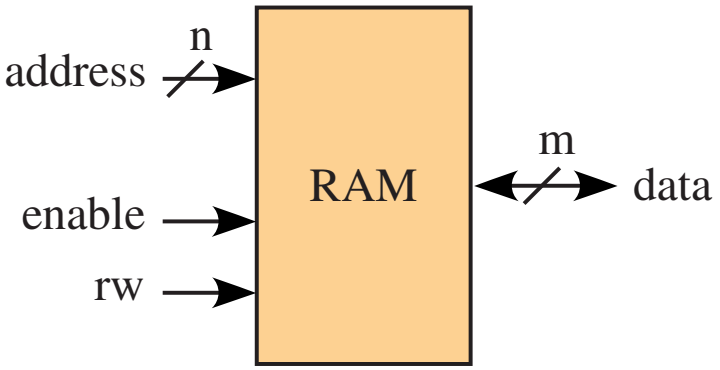
Example 22: (Cont.)

```
          ⋮
ARCHITECTURE behav OF dual_port_ram IS
BEGIN
   memory:  PROCESS (r, w, address_in, address_out, data_in) IS
      TYPE rm IS ARRAY (0 TO 2**n-1) OF std_logic_vector (m-1 DOWNTO 0);
      VARIABLE word:  rm;
   BEGIN
      IF w = '1' THEN           -- write location in memory
         word(to_integer(address_in)) := data_in;
      END IF;

      IF r = '1' THEN           -- read location in memory
         data_out <= word(to_integer(address_out));
      END IF;
   END PROCESS memory;
END ARCHITECTURE behav;
```

RAMs (Cont.)

Example: 23 (Class) *Show how to model an n-bit address, m-bit word size, single-port RAM.*




```
LIBRARY ieee;
USE ieee.std_logic_1164.ALL;
USE ieee.numeric_std.ALL;

ENTITY single_port_ram IS
   GENERIC(  n:  positive := 4;
             m:  positive := 4);
   PORT( rw, enable:  IN std_logic;
         address:  IN unsigned (n-1 DOWNTO 0);
         data:  INOUT std_logic_vector (m-1 DOWNTO 0));
END ENTITY single_port_ram;
         :
         :
```

RAMs (Cont.)

Example 23: (Cont.)

```
          :
          :
ARCHITECTURE behav OF single_port_ram IS BEGIN
   memory:  PROCESS (rw, enable, address, data) IS
      TYPE rm IS ARRAY (0 TO 2**n-1) OF std_logic_vector (m-1 DOWNTO 0);
      VARIABLE word:  rm;
   BEGIN
      data <= (OTHERS => 'Z');
      IF enable = '1' THEN
         IF rw = '1' THEN          -- write location in memory
             word(to_integer(address)) := data;
         ELSE                      -- read location in memory
             data <= word(to_integer(address));
         END IF;
      END IF;
   END PROCESS memory;
END ARCHITECTURE behav;
```

Example: 24 (Home) *Explain why data is all driven to Z at the beginning of the process in Example 23.*