EASTERN MEDITERRANEAN UNIVERSITY
COMPUTER ENGINEERING DEPARTMENT
CMPE224 **DIGITAL LOGIC SYSTEMS**
VHDL EXPERIMENT IV

TITLE: VHDL IMPLEMENTATION OF SEQUENTIAL CIRCUITS: BEHAVIORAL STATE TRANSITION DIAGRAM (TABLE) APPROACH.

OBJECTIVES: Implementation of sequential circuits using their functional (or behavioral) state transition table or state transition diagram descriptions will be studied. In this respect, VHDL implementation of state transition table descriptions will be introduced with examples. The students are expected to learn the mapping of behavioral description of digital systems to VHDL implementations.

Motivation:

1. ***Read the explanations in "Additional VHDL Topics Used in This Experiment".***
2. ***Examine the source codes given below.***

ADDITIONAL TOPICS TO BE COVERED WITHIN THIS EXPERIMENT

In practical implementation of digital systems, it is generally tedious to manually synthesize a circuit from a state transition diagram or state transition table. VHDL allows us to write a code that represents the state transition diagram, as described below.

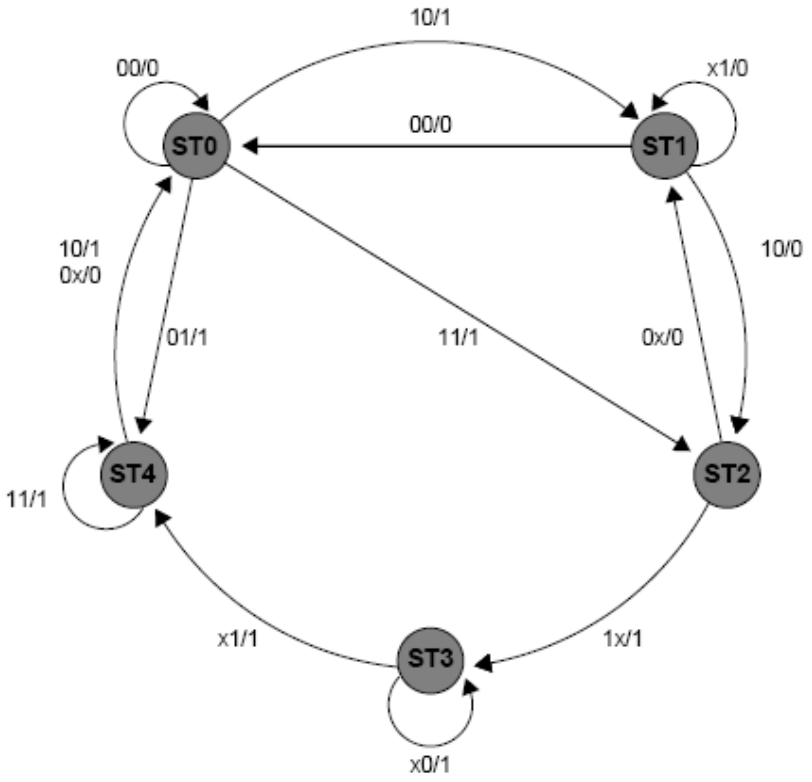
A generic coding template for VHDL implementation of sequential circuits from state transition descriptions is given below.

```
library declarations
entity model_name is
        port
        (
        list of inputs and outputs
        );
end model_name;
architecture behavior of model_name is
internal signal declarations
begin
        — the state process defines the storage elements
        state: process ( sensitivity list — clock, reset, next_state, inputs)
        begin
                vhdl statements for state elements
        end process state;
        — the comb process defines the combinational logic
        comb: process ( sensitivity list — usually includes all inputs)
        begin
                vhdl statements which specify combinational logic
```

```
        end process comb;
end behavior;
```

You will find detailed examples on instances of this template below. Examine these VHDL codes and see how the above template is used.

Example 1: Consider the following Mealy-type state transition diagram. It conatins five states and describes a sequential circuit with two inputs, "data_in(1), data_in(0)", and one output "data_out". The VHDL code implementing this state transition diagram is given below.




```
library ieee;
use ieee.std_logic_1164.all;
entity First_Example is          -- Top-level design entity, we assumed that state ST0
                                 -- is the reset state and an asynchronous  reset signal
                                 -- puts the system into this state.
        port (clock, reset: in std_logic;
                data_out: out std_logic;
                data_in: in std_logic_vector (1 downto 0));
end First_Example;

architecture state_behavior of First_Example is     -- architectural description
                                 -- using the above given state transition diagram.
```

```
type state_values is (st0, st1, st2, st3, st4);    -- allows us to create a user-defined data
                                                   -- type. The new signal type is
                                                   -- state_values and it can have five
                                                   -- possible values: st0, st1, st2, st3, st4,
                                                   -- and st5.
signal pres_state, next_state: state_values;       -- defines signals that are of the
                                                   -- state_value type.
begin
state: process (clock, reset)    -- process that implements sequential behavior
begin
        if (reset = '0') then
                pres_state <= st0;
        elsif (clock'event and clock ='1') then
                case pres_state is
                        when st0 =>
                                case data_in is
                                        when "00" => next_state <= st0;
                                        when "01" => next_state <= st4;
                                        when "10" => next_state <= st1;
                                        when "11" => next_state <= st2;
                                end case;
                        when st1 =>
                                case data_in is
                                        when "00" => next_state <= st0;
                                        when "10" => next_state <= st2;
                                        when others => next_state <= st1;
                                end case;
                        when st2 =>
                                case data_in is
                                        when "00" => next_state <= st1;
                                        when "01" => next_state <= st1;
                                        when "10" => next_state <= st3;
                                        when "11" => next_state <= st3;
                                end case;
                        when st3 =>
                                case data_in is
                                        when "01" => next_state <= st4;
                                        when "11" => next_state <= st4;
                                        when others => next_state <= st3;
                                end case;
                        when st4 =>
                                case data_in is
                                        when "11" => next_state <= st4;
                                        when others => next_state <= st0;
                                end case;
                        when others => next_state <= st0;
                end case;
        end if;
end process state; -- Mealy output definition using pres_state and  data_in
output: process (pres_state, data_in)
```

```
begin
        case pres_state is
                when st0 =>
                        case data_in is
                                when "00" => data_out <= '0';
                                when others => data_out <= '1';
                        end case;
                when st1 => data_out <= '0';
                when st2 =>
                        case data_in is
                                when "00" => data_out <= '0';
                                when "01" => data_out <= '0';
                                when others => data_out <= '1';
                        end case;
                when st3 => data_out <= '1';
                when st4 =>
                        case data_in is
                                when "10" => data_out <= '1';
                                when "11" => data_out <= '1';
                                when others => data_out <= '0';
                        end case;
                when others => data_out <= '0';
        end case;
end process output;
end state_behavior;
```

Example 2

A sequential circuit has two inputs **w1** and **w2**, and one output **z**. Its function is to compare the input sequences on the two inputs. If **w1=w2** during any *four consecutive clock cycles*, the circuit produces **z=1**; otherwise **z=0**. For example,

w1: 0**1101**1**1000**1**1**0
w2: 1**1101**0**1000**1**1**1
z : 0000**1**0000**111**0

A Mealy-type state transition table for this circuit is given below, where the equality of two 1-bit signals is detected using an XOR gate. That is, to compare individual bits, let **k=w1⊕w2**, and then, a suitable state transition table is

Present State	Next State		Output z	
	k=0	k=1	k=0	k=1
A	B	A	0	0
B	C	A	0	0
C	D	A	0	0
D	D	A	1	0

Consequently, the state-assigned table (using the rules given in lecture notes) is

Present	Next State	Output

State	k=0	k=1	k=0	k=1
y_2y_1	Y_2Y_1	Y_2Y_1	z	z
00	01	00	0	0
01	10	00	0	0
10	11	00	0	0
11	11	00	1	0

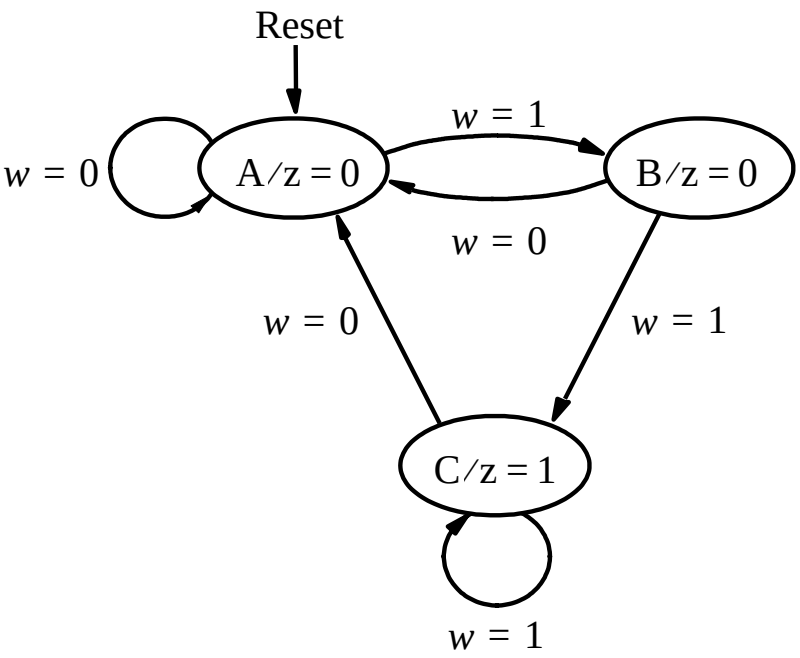
The VHDL code implementing the above given functional description is:

```
LIBRARY ieee ;
USE ieee.std_logic_1164.all ;

ENTITY Example_2 IS
        PORT ( Clock : IN STD_LOGIC ;
                Resetn : IN STD_LOGIC ;
                w1, w2 : IN STD_LOGIC ;
                z : OUT STD_LOGIC ) ;
END Example_2 ;

ARCHITECTURE State_Behavior OF Example_2 IS
TYPE State_type IS (A, B, C, D);
SIGNAL y : State_type ;
SIGNAL k : STD_LOGIC;

BEGIN
        k <= w1 XOR w2 ;
        PROCESS ( Resetn, Clock )
        BEGIN
                IF Resetn = '0' THEN
                        y <= A ;
                ELSIF (Clock'EVENT AND Clock = '1') THEN
                        CASE y IS
                                WHEN A =>
                                        IF k = '0' THEN
                                                y <= B ;
                                        ELSE
                                                y <= A ;
                                        END IF ;
                                WHEN B =>
                                        IF k = '0' THEN
                                                y <= C ;
                                        ELSE
                                                y <= A ;
                                        END IF ;
                                WHEN C =>
                                        IF k = '0' THEN
                                                y <= D ;
                                        ELSE
                                                y <= A ;
                                        END IF ;
```

```
                        WHEN D =>
                                IF k = '0' THEN
                                        y <= D ;
                                ELSE
                                        y <= A ;
                                END IF ;
                END CASE ;
        END IF ;
END PROCESS;
z <= '1' WHEN y = D AND k = '0' ELSE '0' ;
END State_Behavior ;
```

Example 3

Consider the following Moore-type state transition diagram.



Its corresponding VHDL implementation is**:**

```
LIBRARY ieee;
USE ieee.std_logic_1164.all ;

ENTITY Moore_Type IS
        PORT (Clock, Resetn, w : IN STD_LOGIC ;
                z                       : OUT STD_LOGIC ) ;
END Moore_Type ;


ARCHITECTURE Behavior OF Moore_Type IS
        TYPE State_type IS (A, B, C) ;
```

```
        SIGNAL y : State_type ;
BEGIN
        PROCESS ( Resetn, Clock )
        BEGIN
                IF Resetn = '0' THEN
                        y <= A ;
                ELSIF (Clock'EVENT AND Clock = '1') THEN
                        CASE y IS
                                WHEN A =>
                                        IF w = '0' THEN
                                                y <= A ;
                                        ELSE
                                                y <= B ;
                                        END IF ;
                                WHEN B =>
                                        IF w = '0' THEN
                                                y <= A ;
                                        ELSE
                                                y <= C ;
                                        END IF ;
                                WHEN C =>
                                        IF w = '0' THEN
                                                y <= A ;
                                        ELSE
                                                y <= C ;
                                        END IF ;
                        END CASE ;
                END IF ;
        END PROCESS ;
        z <= '1' WHEN y = C ELSE '0' ;
END Behavior ;
```

Another alternative coding style that can also be used for the other examples is given below: The only change is that sensitivity to the asynchronous “reset” signal is defied within another “process” statement.

```
ARCHITECTURE Behavior OF Moore_Type IS
         TYPE State_type IS (A, B, C) ;
         SIGNAL y_present, y_next : State_type ;
BEGIN
         PROCESS ( w, y_present )
         BEGIN
                CASE y_present IS
                        WHEN A =>
                                IF w = '0' THEN
                                        y_next <= A ;
                                ELSE
                                        y_next <= B ;
                                END IF ;
```

```
                WHEN B =>
                        IF w = '0' THEN
                                y_next <= A ;
                        ELSE
                                y_next <= C ;
                        END IF ;

                WHEN C =>
                        IF w = '0' THEN
                                y_next <= A ;
                        ELSE
                                y_next <= C ;
                        END IF ;
        END CASE ;
    END PROCESS ;

    PROCESS (Clock, Resetn)
    BEGIN
        IF (Resetn = '0') THEN
                y_present <= A ;
        ELSIF (Clock'EVENT AND Clock = '1') THEN
                y_present <= y_next ;
        END IF ;
    END PROCESS ;

    z <= '1' WHEN y_present = C ELSE '0' ;
END Behavior ;
```

Example 4

Note that, in the above examples there was no state assignment. Instead, symbolic labels of states are used. This means that we have not specified the number of flip-flops that should be used, the VHDL compiler automatically chooses an appropriate number of FFs when synthesizing the circuit.

However, we have seen that state assignment may have an impact on the complexity of a designed circuit; it is possible to reduce the number of gates used in FF input circuits and save from the implementation cost. In some cases, the structure of programmable logic devices (PLDs) also forces to make the state assignment manually.

In VHDL, it is possible to make the state assignment manually, but there is no standardized way of doing it. On way of manual state assignment is to define symbolic state names as constants, with the value of each constant corresponding to the desired encoding. When the code is translated, VHDL replaces symbolic names with their assigned constant values. This is illustrated in the following VHDL code which is implementing the system of Example 3.

```
LIBRARY ieee ;
USE ieee.std_logic_1164.all ;

ENTITY Example4 IS
        PORT ( Clock, Reset, w : IN STD_LOGIC ;
                        z               : OUT     STD_LOGIC ) ;
END Example4;

ARCHITECTURE Behavior OF Example4 IS
        SIGNAL y_present, y_next : STD_LOGIC_VECTOR(1 DOWNTO 0);
                -- We are using two-bit codes for state names, hence present-state
                -- and next-state variables  are defined as 2-bit vectors.
        CONSTANT A        : STD_LOGIC_VECTOR(1 DOWNTO 0) := "00" ;
                -- state name A is encoded wit “00”
        CONSTANT B        : STD_LOGIC_VECTOR(1 DOWNTO 0) := "01" ;
                -- state name B is encoded with “01”
        CONSTANT C        : STD_LOGIC_VECTOR(1 DOWNTO 0) := "11" ;
                -- state name C is encoded with “11”
BEGIN
        PROCESS ( w, y_present )
        BEGIN
                CASE y_present IS
                        WHEN A =>
                                IF w = '0' THEN y_next <= A ;
                                ELSE y_next <= B ;
                                END IF ;

                        WHEN B =>
                                IF w = '0' THEN y_next <= A ;
                                ELSE y_next <= C ;
                                END IF ;
                        WHEN C =>
                                IF w = '0' THEN y_next <= A ;
                                ELSE y_next <= C ;
                                END IF ;
                        WHEN OTHERS =>            -- since y_present is a
                                --  STD_LOGIC_VECTOR signal (not a state_type
                                -- signal), we MUST provide when other clause to
                                -- prevent the machine entering the unused state.
                                y_next <= A ;
                END CASE ;
        END PROCESS ;

        PROCESS ( Clock, Reset)
        BEGIN
                IF Reset = '0' THEN
                        y_present <= A ;
                ELSIF (Clock'EVENT AND Clock = '1') THEN
                        y_present <= y_next ;
```

```
        END IF ;
    END PROCESS ;
    z <= '1' WHEN y_present = C ELSE '0' ;
END Behavior ;
```

PRELIMINARY WORK:

Consider the following design problem: we want to design a 3-bit parity generator. For every three bits that are observed on the input ***w*** during three consecutive clock cycles (this means three consecutive bits on w), the circuit generates the parity bit p=1 if and only if the number of 1's in the 3-bit sequence is odd. We assume that the 3-bit codes **do not** overlap.

A 5-state minimal state-transition table for this circuit is given below.

Present State	Next State		Output p	
	w=0	w=1	w=0	w=1
A	B	C	0	0
B	D	E	0	0
C	E	D	0	0
D	A	A	0	1
E	A	A	1	0

I. Implement this state-transition table description in VHDL using automatic state-assignment approach.

II. Implement this state transition table description in VHDL using manual state-assignment approach. Show your state assignments and specify reasoning about them.

III. **Prepare** steps I and II **as homework**. Perform all experimental steps including **VHDL coding, waveform preparation, and simulations**; make them ready before you come to the laboratory.

EXPERIMENTAL WORK:

Demonstrate your preliminary work to the laboratory assistants using Quartus II environment. Be ready for detailed questions on your work.

Good Luck.

Dr. Adnan Acan
Dr. Evgueni Doukhnitch
Dr. Muhammed Salamah.