Synchronous Sequential Circuits

- **Sequential circuits** – outputs depend on past behavior as well as present inputs.
- **Synchronous circuits** – use a clock signal to sequence behavior.
- **Asynchronous circuits** – no clock signal is used (see Chapter 9).

A Simple Example

- Circuit has one input, \( w \), and one output, \( z \).
- Changes occur on positive clock edge.
- \( z \) is 1 if \( w \) is 1 during last two clock cycles.

```
<table>
<thead>
<tr>
<th>Clock cycle</th>
<th>t_0</th>
<th>t_1</th>
<th>t_2</th>
<th>t_3</th>
<th>t_4</th>
<th>t_5</th>
<th>t_6</th>
<th>t_7</th>
</tr>
</thead>
<tbody>
<tr>
<td>w</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>z</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>1</td>
</tr>
</tbody>
</table>
```

<table>
<thead>
<tr>
<th>Present state</th>
<th>Next state</th>
<th>Output</th>
</tr>
</thead>
<tbody>
<tr>
<td>( w = 0 )</td>
<td>( w = 1 )</td>
<td>( z )</td>
</tr>
<tr>
<td>A</td>
<td>A</td>
<td>B</td>
</tr>
<tr>
<td>B</td>
<td>A</td>
<td>C</td>
</tr>
<tr>
<td>C</td>
<td>A</td>
<td>C</td>
</tr>
</tbody>
</table>

Figure 8.1: The general form of a sequential circuit

Figure 8.2: Sequences of input and output signals

Figure 8.3: State diagram of a simple sequential circuit

Figure 8.4: State table
Figure 8.5 A general sequential circuit

Figure 8.6 A State-assigned table

Figure 8.7 Derivation of logic expressions

Figure 8.8 Sequential circuit

Summary of Design Steps

- Obtain specification of the desired circuit.
- Create a state diagram from specification.
- Create a state table from state diagram.
- Perform state minimization.
- Perform state assignment.
- Derive the next-state logic expressions.
- Implement circuit described by logic.
A digital system with k registers

A shift register control circuit

Signals needed in Example 8.1

State diagram

State table

State-assigned table
Figure 8.14 Derivation of next-state expressions

Figure 8.15 Sequential circuit

Figure 8.16 Improved state assignment

Figure 8.17 Final circuit for the improved state assignment

Figure 8.18 Improved state assignment

Figure 8.19 Derivation of next-state expressions
Mealy State Model

- *Moore machines* – output is determined only by present state.
- *Mealy machines* – output depends on both present state and input values.

<table>
<thead>
<tr>
<th>Present state</th>
<th>Next state</th>
<th>Output z</th>
</tr>
</thead>
<tbody>
<tr>
<td>A</td>
<td>A</td>
<td>0</td>
</tr>
<tr>
<td>B</td>
<td>A</td>
<td>0</td>
</tr>
<tr>
<td></td>
<td>B</td>
<td>1</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Present state</th>
<th>Next state</th>
<th>Output z</th>
</tr>
</thead>
<tbody>
<tr>
<td>A</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>B</td>
<td>0</td>
<td>1</td>
</tr>
</tbody>
</table>

Figure 8.22: Sequences of input and output signals
Design of FSMs using CAD tools

- Could design using manual techniques then use schematic capture or structural VHDL.
- Instead should enter state table via a state diagram editor or behavioral VHDL.
Figure 8.30 Implementation of an FSM in a CPLD

Figure 8.31 An FSM circuit in a small CPLD

Figure 8.32 Simulation results

(a) Timing simulation results

(b) Magnified simulation results, showing timing details

Figure 8.33 Alternative style of code for an FSM

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

Figure 8.34 A user-defined attribute for manual state assignment

ARCHITECTURE Behavior OF simple IS
TYPE State_type IS (A, B, C);
ATTRIBUTE ENUM_ENCODING IS STRING;
BEGIN
...
LIBRARY ieee;
USE ieee.std_logic_1164.all;

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

ARCHITECTURE Behavior OF simple IS
    SIGNAL y_present, y_next : STD_LOGIC_VECTOR(1 DOWNTO 0);
    CONSTANT A : STD_LOGIC_VECTOR(1 DOWNTO 0) := "00";
    CONSTANT B : STD_LOGIC_VECTOR(1 DOWNTO 0) := "01";
    CONSTANT C : STD_LOGIC_VECTOR(1 DOWNTO 0) := "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 =>
                y_next <= A;
        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;

LIBRARY ieee;
USE ieee.std_logic_1164.all;

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

ARCHITECTURE Behavior OF mealy IS
    TYPE State_type IS (A, B);
    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 <= B;
                END IF;
            WHEN OTHERS =>
                y <= A;
            END CASE;
        END PROCESS;

    PROCESS (y, w)
    BEGIN
        CASE y IS
            WHEN A =>
                z <= '0';
            WHEN B =>
                z <= w;
        END CASE;
    END PROCESS;
END Behavior;

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 =>
    y_next <= A;
END CASE;
END PROCESS;

WHEN B =>
    IF w = '0' THEN y <= A;
    ELSE y <= B;
END IF;
END CASE;
END PROCESS;

WHEN B =>
    IF w = '0' THEN y <= A;
    ELSE y <= B;
END IF;
END CASE;
END PROCESS;

WHEN B =>
    IF w = '0' THEN y <= A;
    ELSE y <= B;
END IF;
END CASE;
END PROCESS;

WHEN B =>
    IF w = '0' THEN y <= A;
    ELSE y <= B;
END IF;
END CASE;
END PROCESS;

WHEN B =>
    IF w = '0' THEN y <= A;
    ELSE y <= B;
END IF;
END CASE;
END PROCESS;
Figure 8.39 Block diagram of a serial adder

Figure 8.40 State diagram for the serial adder

Figure 8.41 State table for the serial adder

Figure 8.42 Circuit for the adder FSM

Figure 8.43 State diagram for the Moore-type serial adder FSM

Figure 8.44 State table for the Moore-type serial adder FSM
Figure 8.47 Circuit for the Moore-type serial adder FSM

Figure 8.48a Code for a left-to-right shift register with an enable input

```vhdl
LIBRARY ieee;
USE ieee.std_logic_1164.all;
-- left-to-right shift register with parallel load and enable
ENTITY shiftrne IS
GENERIC (N : INTEGER := 4);
PORT (R: IN STD_LOGIC_VECTOR(N-1 DOWNTO 0);
L, E, w : IN STD_LOGIC;
Clock : IN STD_LOGIC;
Q : BUFFER STD_LOGIC_VECTOR(N-1 DOWNTO 0));
END shiftrne;
ARCHITECTURE Behavior OF shiftrne IS
BEGIN
PROCESS
BEGIN
... con't
WAIT UNTIL Clock'EVENT AND Clock = '1';
IF E = '1' THEN
    IF L = '1' THEN
        Q <= R;
    ELSE
        Genbits: FOR i IN 0 TO N - 2 LOOP
            Q(i) <= Q(i+1);
        END LOOP;
        Q(N-1) <= w;
    END IF;
END IF;
END PROCESS;
END Behavior;
```

Figure 8.48b VHDL code for the serial adder (con't)

```vhdl
WITH y SELECT
    s <= QA(0) XOR QB(0) WHEN G,
        NOT (QA(0) XOR QB(0)) WHEN H;
END CASE;
END PROCESS;
```

WITH y SELECT
    s <= QA(0) XOR QB(0) WHEN G,
        NOT (QA(0) XOR QB(0)) WHEN H;
END CASE;
END PROCESS;
```

Figures 8.48c VHDL code for the serial adder (con't)

Figure 8.49a VHDL code for the serial adder

```vhdl
LIBRARY ieee;
USE ieee.std_logic_1164.all;
ENTITY serial IS
GENERIC (length: INTEGER := 8);
PORT (Clock : IN STD_LOGIC;
Reset : IN STD_LOGIC;
A, B : IN STD_LOGIC_VECTOR(length-1 DOWNTO 0);
Sum : BUFFER STD_LOGIC_VECTOR(length-1 DOWNTO 0));
END serial;
ARCHITECTURE Behavior OF serial IS
COMPONENT shiftrne
GENERIC (N: INTEGER := 4);
PORT (R: IN STD_LOGIC_VECTOR(N-1 DOWNTO 0);
L, E, w: IN STD_LOGIC;
Clock: IN STD_LOGIC;
Q: BUFFER STD_LOGIC_VECTOR(N-1 DOWNTO 0));
END COMPONENT;
SIGNAL QA, QB, Null_in: STD_LOGIC_VECTOR(length-1 DOWNTO 0);
SIGNAL s, Low, High, Run: STD_LOGIC;
SIGNAL Count: INTEGER RANGE 0 TO length;
TYPE State_type IS (G, H);
SIGNAL y: State_type;
BEGIN
Low <= '0'; High <= '1';
Shift.shiftrne(GENERIC MAP(N => length))
PORT MAP (A, Reset, High, Low, Clock, QA);
Shift.shiftrne(GENERIC MAP(N => length))
PORT MAP (B, Reset, High, Low, Clock, QB);
AdderFSM PROCESS (Reset, Clock)
BEGIN
IF Reset = '1' THEN
    y <= G;
ELSIF Clock'EVENT AND Clock = '1' THEN
    CASE y IS
    WHEN G =>
        IF QA(0) = '1' AND QB(0) = '1' THEN
            y <= H;
        ELSE
            y <= G;
        END IF;
    WHEN H =>
        IF QA(0) = '0' AND QB(0) = '0' THEN
            y <= G;
        ELSE
            y <= H;
        END IF;
    END CASE;
END IF;
END PROCESS;
BEGIN
BEGIN
    ... con't
    WITH y SELECT
    s <= QA(0) XOR QB(0) WHEN G,
        NOT (QA(0) XOR QB(0)) WHEN H;
    END CASE;
    END PROCESS;
END Behavior;
```
State Minimization

- It is often difficult for designer to find FSM with minimal number of states.
- Fewer states leads to fewer flip-flops.
- Two states \( S_i \) and \( S_j \) are equivalent iff for every input sequence starting in \( S_i \) or \( S_j \), the same output sequence is produced.

Partitioning Minimization

- If \( w=0 \) in \( S_i \) and result is \( S_u \) then \( S_u \) is 0-successor.
- If \( w=1 \) in \( S_i \) and result is \( S_v \) then \( S_v \) is 1-successor.
- \( S_i \) and \( S_j \) are equivalent if \( k \)-successors equivalent.
- Consider states as set then break set into partitions comprised of subsets which are not equivalent.
- A partition consists of 1 or more blocks, each block comprises a subset of states that may be equivalent, but states in a block are definitely not equivalent to states in another block.
Incompletely Specified FSMs

- Partitioning scheme works well when all entries in state table are specified.
- If one or more entries are not specified, then the FSM is incompletely specified.
- Partitioning scheme not guaranteed to produce minimal solution in this case.

<table>
<thead>
<tr>
<th>Present state</th>
<th>Next state</th>
<th>Output z</th>
</tr>
</thead>
<tbody>
<tr>
<td>w = 0</td>
<td>w = 1</td>
<td></td>
</tr>
<tr>
<td>A</td>
<td>B C</td>
<td>0 0</td>
</tr>
<tr>
<td>B</td>
<td>D -</td>
<td>0 -</td>
</tr>
<tr>
<td>C</td>
<td>F E</td>
<td>0 1</td>
</tr>
<tr>
<td>D</td>
<td>B G</td>
<td>0 0</td>
</tr>
<tr>
<td>E</td>
<td>F C</td>
<td>0 1</td>
</tr>
<tr>
<td>F</td>
<td>E D</td>
<td>0 1</td>
</tr>
<tr>
<td>G</td>
<td>F -</td>
<td>0 -</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Present state</th>
<th>Next state</th>
<th>Output z</th>
</tr>
</thead>
<tbody>
<tr>
<td>w = 0</td>
<td>w = 1</td>
<td></td>
</tr>
<tr>
<td>A</td>
<td>B C</td>
<td>0 0</td>
</tr>
<tr>
<td>B</td>
<td>D -</td>
<td>0 -</td>
</tr>
<tr>
<td>C</td>
<td>F E</td>
<td>0 1</td>
</tr>
<tr>
<td>D</td>
<td>B G</td>
<td>0 0</td>
</tr>
<tr>
<td>E</td>
<td>F C</td>
<td>0 1</td>
</tr>
<tr>
<td>F</td>
<td>E D</td>
<td>0 1</td>
</tr>
<tr>
<td>G</td>
<td>F -</td>
<td>0 -</td>
</tr>
</tbody>
</table>

Figure 8.59: Incompletely specified state table for Example 8.7

Figure 8.60: State diagram for a counter

Figure 8.61: State table for the counter

Figure 8.62: State-assigned table for the counter
Implementation Using JK-FF

- If FF in state 0 to remain 0, J=0 and K=d.
- If FF in state 0 to change to 1, J=1 and K=d.
- If FF in state 1 to remain 1, J=d and K=0.
- If FF in state 1 to change to 0, J=d and K=1.
Figure 8.68  Factored-form implementation of the counter

Figure 8.69  State table for the counter-like example

<table>
<thead>
<tr>
<th>Present state</th>
<th>Next state</th>
<th>Output</th>
</tr>
</thead>
<tbody>
<tr>
<td>000</td>
<td>100</td>
<td>000</td>
</tr>
<tr>
<td>000</td>
<td>010</td>
<td>000</td>
</tr>
<tr>
<td>010</td>
<td>010</td>
<td>010</td>
</tr>
<tr>
<td>010</td>
<td>001</td>
<td>110</td>
</tr>
<tr>
<td>010</td>
<td>011</td>
<td>011</td>
</tr>
<tr>
<td>111</td>
<td>000</td>
<td>111</td>
</tr>
</tbody>
</table>

Figure 8.70  State-assigned table

Figure 8.71  Circuit for the counter-like example

Figure 8.72  State diagram for the arbiter

Figure 8.73  Alternative style of state diagram for the arbiter
LIBRARY ieee;
USE ieee.std_logic_1164.all;

ENTITY arbiter IS
  PORT ( Clock, Resetn : IN STD_LOGIC;
         r : IN STD_LOGIC_VECTOR(1 TO 3);
         g : OUT STD_LOGIC_VECTOR(1 TO 3));
END arbiter;

ARCHITECTURE Behavior OF arbiter IS
  TYPE State_type IS (Idle, gnt1, gnt2, gnt3);
  SIGNAL y : State_type;
BEGIN
  PROCESS (Resetn, Clock)
  BEGIN
    IF Resetn = '0' THEN y <= Idle;
    ELSIF (Clock'EVENT AND Clock = '1') THEN
      CASE y IS
      WHEN Idle =>
        IF r(1) = '1' THEN y <= gnt1;
        ELSIF r(2) = '1' THEN y <= gnt2;
        ELSIF r(3) = '1' THEN y <= gnt3;
        ELSE y <= Idle;
        END IF;
      WHEN gnt1 =>
        IF r(1) = '1' THEN y <= gnt1;
        ELSE y <= Idle;
        END IF;
      WHEN gnt2 =>
        IF r(2) = '1' THEN y <= gnt2;
        ELSE y <= Idle;
        END IF;
      WHEN gnt3 =>
        IF r(3) = '1' THEN y <= gnt3;
        ELSE y <= Idle;
        END IF;
      END CASE;
    END IF;
  g(1) <= '1' WHEN y = gnt1 ELSE '0';
  g(2) <= '1' WHEN y = gnt2 ELSE '0';
  g(3) <= '1' WHEN y = gnt3 ELSE '0';
  END PROCESS;
END Behavior;

Figure 8.75 Incorrect VHDL code for the grant signals

- - -

- - -

PROCESS (y)
BEGIN
  IF y = gnt1 THEN g(1) <= '1';
  ELSIF y = gnt2 THEN g(2) <= '1';
  ELSIF y = gnt3 THEN g(3) <= '1';
  END IF;
END PROCESS;
END Behavior;

Figure 8.76 Correct VHDL code for the grant signals

Figure 8.74 VHDL code for the arbiter (cont’d)

Figure 8.74 VHDL code for the arbiter

Figure 8.76 VHDL code for the grant signals

- - -

- - -

PROCESS (y)
BEGIN
  g(1) <= '0';
  g(2) <= '0';
  g(3) <= '0';
  IF y = gnt1 THEN g(1) <= '1';
  ELSIF y = gnt2 THEN g(2) <= '1';
  ELSIF y = gnt3 THEN g(3) <= '1';
  END IF;
END PROCESS;
END Behavior;

Figure 8.75 Incorrect VHDL code for the grant signals

Figure 8.73 Simulation results for the arbiter circuit

a) Output delays using binary encoding

b) Output delays using one-hot encoding

Figure 8.77 Simulation results for the arbiter circuit

Figure 8.78 Output delays in the arbiter circuit
Figure 8.80. Circuit for Example 8.8

Table 8.81. Tables for the circuit in Example 8.8

(a) State-assigned table

<table>
<thead>
<tr>
<th>Present state (w)</th>
<th>(u)</th>
<th>(v)</th>
<th>(x)</th>
<th>(y)</th>
</tr>
</thead>
<tbody>
<tr>
<td>00</td>
<td>00</td>
<td>00</td>
<td>00</td>
<td>00</td>
</tr>
<tr>
<td>01</td>
<td>00</td>
<td>01</td>
<td>00</td>
<td>01</td>
</tr>
<tr>
<td>10</td>
<td>01</td>
<td>00</td>
<td>10</td>
<td>10</td>
</tr>
<tr>
<td>11</td>
<td>00</td>
<td>00</td>
<td>00</td>
<td>00</td>
</tr>
</tbody>
</table>

(b) State table

<table>
<thead>
<tr>
<th>Present state (w)</th>
<th>(u)</th>
<th>(v)</th>
<th>(x)</th>
<th>(y)</th>
</tr>
</thead>
<tbody>
<tr>
<td>A</td>
<td>A</td>
<td>A</td>
<td>A</td>
<td>A</td>
</tr>
<tr>
<td>B</td>
<td>A</td>
<td>C</td>
<td>B</td>
<td>C</td>
</tr>
<tr>
<td>C</td>
<td>A</td>
<td>D</td>
<td>D</td>
<td>D</td>
</tr>
<tr>
<td>D</td>
<td>A</td>
<td>D</td>
<td>D</td>
<td>D</td>
</tr>
</tbody>
</table>

Figure 8.82. Circuit for Example 8.9

Figure 8.83. Excitation table

<table>
<thead>
<tr>
<th>Present state (w)</th>
<th>(u)</th>
<th>(v)</th>
<th>(x)</th>
<th>(y)</th>
</tr>
</thead>
<tbody>
<tr>
<td>00</td>
<td>01</td>
<td>01</td>
<td>00</td>
<td>00</td>
</tr>
<tr>
<td>01</td>
<td>01</td>
<td>01</td>
<td>10</td>
<td>10</td>
</tr>
<tr>
<td>10</td>
<td>01</td>
<td>01</td>
<td>00</td>
<td>00</td>
</tr>
<tr>
<td>11</td>
<td>01</td>
<td>10</td>
<td>10</td>
<td>10</td>
</tr>
</tbody>
</table>

Figure 8.84. Circuit for Example 8.10

Figure 8.85. Excitation table

<table>
<thead>
<tr>
<th>Present state (w)</th>
<th>(u)</th>
<th>(v)</th>
<th>(x)</th>
<th>(y)</th>
</tr>
</thead>
<tbody>
<tr>
<td>00</td>
<td>00</td>
<td>00</td>
<td>01</td>
<td>00</td>
</tr>
<tr>
<td>01</td>
<td>00</td>
<td>00</td>
<td>10</td>
<td>00</td>
</tr>
<tr>
<td>10</td>
<td>10</td>
<td>01</td>
<td>01</td>
<td>01</td>
</tr>
<tr>
<td>11</td>
<td>10</td>
<td>01</td>
<td>01</td>
<td>11</td>
</tr>
</tbody>
</table>
Formal Model for FSMs

- \( M = (W, Z, S, \varphi, \lambda) \)
  - \( W \) is finite, nonempty set of inputs.
  - \( Z \) is finite, nonempty set of outputs.
  - \( S \) is finite, nonempty set of states.
  - \( \varphi \) is the state transition function:
    - \( S(t+1) = \varphi[S(t), W(t)] \)
  - \( \lambda \) is the output function:
    - \( \lambda(t) = \lambda[S(t)] \) (Moore model)
    - \( \lambda(t) = \lambda[W(t), S(t)] \) (Mealy model)
Summary

• FSM Design (Moore and Mealy)
  – State minimization
  – State assignment
  – Using CAD tools and VHDL