DAT093
Introduction to Electronic System Design

FSMs

Finite State Machines

Sven Knutsson
svenk@chalmers.se
Dept. Of Computer Science and Engineering
Chalmers University of Technology
Gothenburg
Sweden

Introduction

FSMs, Finite State Machines, are important structures in digital design

In a FSM we move between a number of defined states for the design. The pace of the move is controlled by a clock and external signals.

The path we take is determined by the inputs to the design.

In a simple structure there might be no inputs so we always move in the same sequence

The states are defined by a number of state variables, in its simpliest form these are just binary numbers

Outputs

In each state the design will generate some kind of output

We have two kinds of FSM.

In a Moore type FSM the outputs in a state are always the same.

In a Moore type FSM we can define the state variable to be the same as the outputs for that state.

In a Mealy type of FSM the outputs in a state can change depending on the inputs to the FSM

In many cases both types can be used but the Moore type will need more states while the coding might be simplier than in the Mealy case

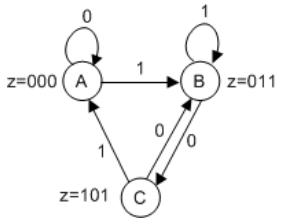
Example Moore

We have the state table

Present state	*Next state*		*Output*
	w=0	*w=1*	*z*
A	A	B	000
B	C	B	011
C	B	A	101

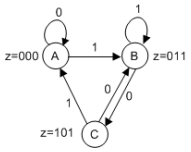
We can let the design tool assign state variables or just let the outputs represent the state

And the state diagram



The output in any given state is constant

Example Moore cont.



Let´s write VHDL code

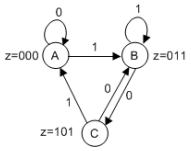
First the entity

The clock sets the pace

We have added an active low reset signal

```
ENTITY Moore IS
   PORT(clk:IN STD_LOGIC;
        Resetn:IN STD_LOGIC;
        w:IN STD_LOGIC;
        z:OUT STD_LOGIC_VECTOR(2 DOWNTO 0));
END Moore;
```

Example Moore cont.



Then the start of the architecture

A new enumbered type, the design tool will turn this into binary variables

Two signals of the new type

```
ARCHITECTURE arch_Moore OF Moore IS
   TYPE State_type IS (A,B,C);
   SIGNAL state:State_type;
   SIGNAL next_state:State_type;
```

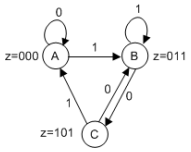
The architecture contains three processes

The state assignment – to determine what state to go to next

The state flow – the transition from one state to the next

The output assignment – assigns the output signals

Example Moore cont.



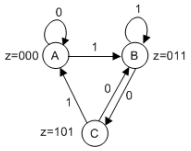
Process for state assignment

```
state_assignment_proc:
PROCESS(w,state)
BEGIN
   CASE state IS
      WHEN A =>
         IF (w = '1') THEN
               next_state <= B;
            ELSE
               next_state <= A;
            END IF;
      WHEN B =>
         IF (w = '1') THEN
            next_state <= B;
         ELSE
            next_state <= C;
         END IF;
      WHEN C =>
         IF (w = '1') THEN
            next_state <= A;
         ELSE
            next_state <= B;
         END IF;
      WHEN OTHERS =>
         next_state <= state;
   END CASE;
END PROCESS state_assignment_proc;
```

What will happen next?

Not needed when we have covered all values of `state`

Example Moore cont.



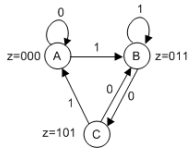
Process for state flow

```
stateflow_proc:
PROCESS(resetn,clk)
BEGIN
   IF (Resetn = '0') THEN
      state <= A;
   ELSIF rising_edge(clk) THEN
      state <= next_state;
   END IF;
END PROCESS stateflow_proc;
```

State A is the reset state

We go to the next state on the rising edge of the clock

Example Moore cont.

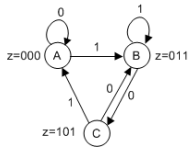


Finally the process for output assignment

```
output_process:
PROCESS(state)
BEGIN
   CASE state IS
      WHEN A =>
         z <= "000";
      WHEN B =>
         z <= "011";
      WHEN C =>
         z <= "101";
      WHEN OTHERS =>
         z <= "000";
   END CASE;
END PROCESS output_process;
```

The output is updated when we reach a new state

Example Moore using constants



We can use the state variables to directly give the outputs

The entity will be the same

Instead of creating a new signal type we define the states as constants

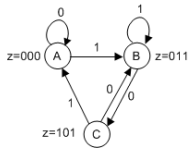
```
ARCHITECTURE arch_Moore_constants OF Moore_constants IS
   CONSTANT A:STD_LOGIC_VECTOR(2 DOWNTO 0):="000";
   CONSTANT B:STD_LOGIC_VECTOR(2 DOWNTO 0):="011";
   CONSTANT C:STD_LOGIC_VECTOR(2 DOWNTO 0):="101";
   SIGNAL state:STD_LOGIC_VECTOR(2 DOWNTO 0);
   SIGNAL next_state:STD_LOGIC_VECTOR(2 DOWNTO 0);
```

Not a enumbered signal but a std_logic_vector

This way the output we want in each state is the same as the state variable

The state assignment and the state flow will be the same with one important difference

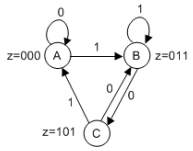
Example Moore using constants cont.




```
state_assignment_proc:
PROCESS(w,state)
BEGIN
   CASE state IS
      WHEN A =>
         IF (w = '1') THEN
               next_state <= B;
            ELSE
               next_state <= A;
            END IF;
      WHEN B =>
         IF (w = '1') THEN
            next_state <= B;
         ELSE
            next_state <= C;
         END IF;
      WHEN C =>
         IF (w = '1') THEN
            next_state <= A;
         ELSE
            next_state <= B;
         END IF;
      WHEN OTHERS =>
         next_state <= state;
   END CASE;
END PROCESS state_assignment_proc;
```

The `others` clause must be there since we are not covering all the 9^3 possible values of the three bit std_logic vector

Example Moore using constants cont.



In this case we don´t need any process to assign output values. We can just assign `state` to the output

```
z <= state;
```

Example Mealy

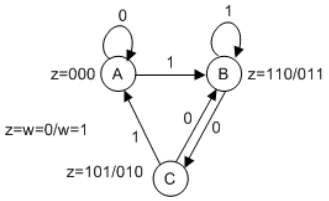
Let´s look at a simular Mealy machine

We have the state table

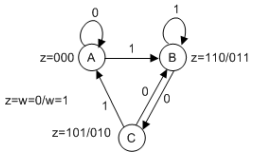
Present state	*Next state*		*Output*	
	w=0	*w=1*	*w_1=0*	*w_1=1*
A	A	B	000	000
B	C	B	110	011
C	B	A	010	101

The output depends on the input in each state

And the state diagram



Example Mealy cont.



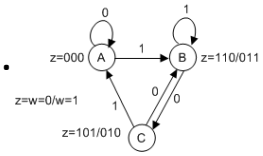
The entity will be the same

```
ENTITY Mealy IS
   PORT(clk:IN STD_LOGIC;
        Resetn:I STD_LOGIC;
        w:IN STD_LOGIC;
        z:OUT STD_LOGIC_VECTOR(2 DOWNTO 0));
END Mealy;
```

We use the same enumbered type as in the Moore machine

```
ARCHITECTURE arch_Mealy OF Mealy IS
   TYPE State_type IS (A,B,C);
   SIGNAL state:State_type;
   SIGNAL next_state:State_type;
```

Example Mealy cont.



What changes is the process for output assignment

```
output_process:
PROCESS(state,w)
BEGIN
   CASE state IS
      WHEN A =>
         z <= "000";
      WHEN B =>
         IF (w = '1') THEN
            z <= "011";
         ELSE
            z <= "110";
         END IF;
      WHEN C =>
         IF (w = '1') THEN
            z <= "101";
         ELSE
            z <= "010";
         END IF;
      WHEN OTHERS =>
         z <= "000";
   END CASE;
END PROCESS output_process;
```

The process must be sensitive to both the state and the input signal w

We have more than one output option in each state

Not needed when we have covered all values of `state`

State coding

The states must be remembered so we can move from one state to the next.

The states have to be coded as binary patterns and stored in flop-flops.

The normal way is to just use binary words .

In our examples with three states we might use

00
01
10

Sometimes it simplifies the logic to use other codes. In this case you could use 11 instead of one of the used codes.

Which code that is used for which state can also influence the complexity of the logic.

In most cases we leave the coding to the syntesis tool.

State coding cont.

In some cases we use one-hot coding.

Here we have one flip-flop for each state.

In our examples the coding would be

001
010
100

This often makes the coding simple.

In this case we need more flip-flops though.
For N states we nead N flip-flops instead of the $^{2}\log(N)$ flip-flops we need in normal coding.

We can configure the synthesis tool to use this coding.

Some times a variant of one-hot coding is used where the reset state is assigned all zeros (000).

The two process method

What we have described has been formalized by Aeroflex Gaisler into the two process method.

The differences from my description are that they have the state flow and the output assignment in the same process and they are collecting all inputs, all outputs and all signals into three clusters.

You really need a substantial number of signals and ports to motivate the use of records so it´s more for larger designs.