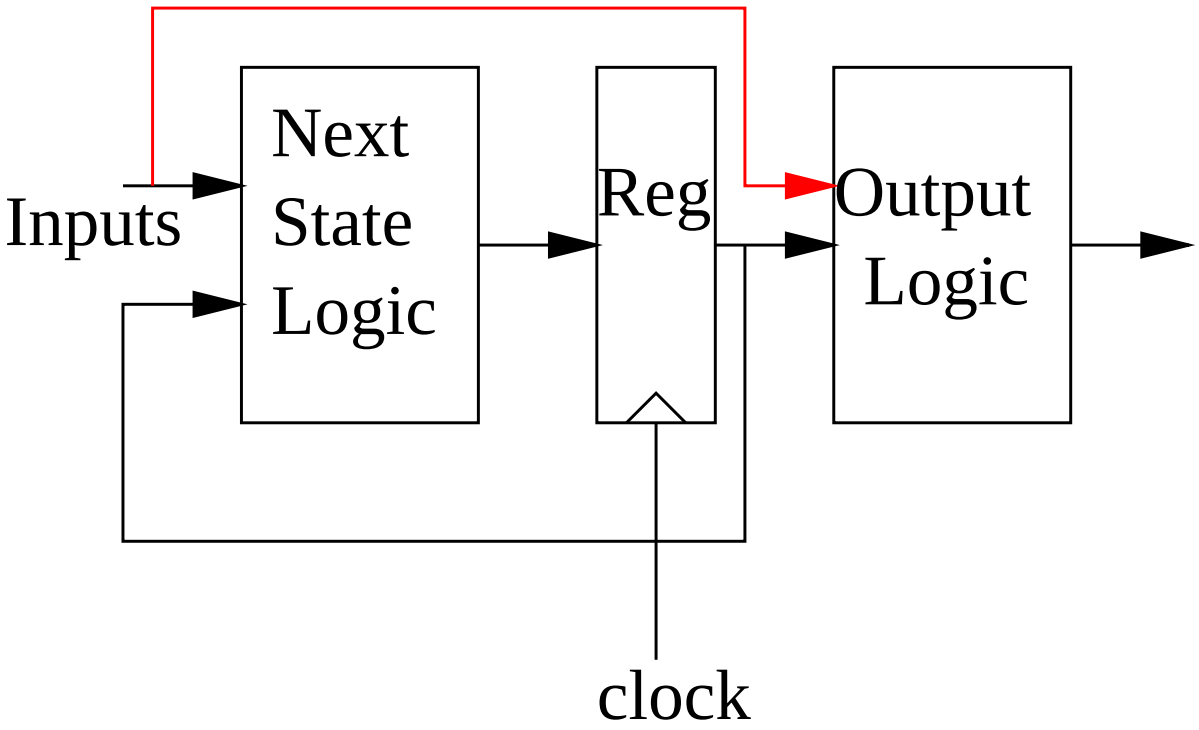
VHDL Descriptions of FSMs

Dr DC Hendry

March 15, 2006

Block Diagram:

The block diagram of a Mealy FSM was:



Block descriptions:

The blocks of this design are:

State Register The state register has as input the clock signal, the next state from the next state logic on the D lines and normally an active low reset line. The output of the state register is the current state of the machine.

Next State Logic This combinational logic has as input the current state and the control inputs, its output is the next state.

Output Logic Another combinational logic block whose inputs are the current state (and only the current state for a Moore machine), and the control inputs (for a Mealy machine). The output is the output of the machine.

VHDL Designs and FSMs

1. Each of the above blocks could be a separate design, but this is usually an overcomplication.

VHDL Designs and FSMs

1. Each of the above blocks could be a separate design, but this is usually an overcomplication.
2. One VHDL design is usually used which includes the state register and all combinational logic, and nothing else.

VHDL Designs and FSMs

1. Each of the above blocks could be a separate design, but this is usually an overcomplication.
2. One VHDL design is usually used which includes the state register and all combinational logic, and nothing else.
3. By including only the FSM in a design the task of FSM analysis required by the synthesis tool for various optimisations is made simpler.

VHDL Type for States

1. User defined types make writing code easier and much more readable.

VHDL Type for States

1. User defined types make writing code easier and much more readable.
2. Such types can also greatly ease debugging of FSM designs as the simulator can show state names rather than state codes for an RTL simulation.

VHDL Type for States

1. User defined types make writing code easier and much more readable.
2. Such types can also greatly ease debugging of FSM designs as the simulator can show state names rather than state codes for an RTL simulation.
3. For the sequence detector of the previous lecture:

```
type statename is (start, seen1, seen10,
                   seen101, success);
signal current_state, next_state : statename;
```

VHDL Type for States

1. User defined types make writing code easier and much more readable.
2. Such types can also greatly ease debugging of FSM designs as the simulator can show state names rather than state codes for an RTL simulation.
3. For the sequence detector of the previous lecture:

```
type statename is (start, seen1, seen10,
                   seen101, success);
signal current_state, next_state : statename;
```

4. The synthesis tool will later assign state codes.

The State Register

- Should be edge triggered and include a reset line.

The State Register

- Should be edge triggered and include a reset line.
- The reset line may be asynchronous or synchronous.

The State Register

- Should be edge triggered and include a reset line.
- The reset line may be asynchronous or synchronous.
- Reset lines are by convention (historical) active low, so names are typically `rst_n`.

The State Register

- Should be edge triggered and include a reset line.
- The reset line may be asynchronous or synchronous.
- Reset lines are by convention (historical) active low, so names are typically `rst_n`.
- On the active edge of the clock, copy `next_state` into `current_state`.

State Register Code

Here is the code for the state register with an asynchronous active low reset line, using a process with a sensitivity list:

```
state_reg : process (clk, rst_n)
begin
  if rst_n = '0' then
    current_state <= start;
  else if clk'event and clk = '1' then
    current_state <= next_state;
  end if;
end process state_reg;
```

Next State Combinational Logic

1. The next state combinational logic for both a Mealy Machine and a Moore Machine has as input the current state and the control inputs.

Next State Combinational Logic

1. The next state combinational logic for both a Mealy Machine and a Moore Machine has as input the current state and the control inputs.
2. As a process therefore, both the current state and the control inputs should be in the sensitivity list of the process.

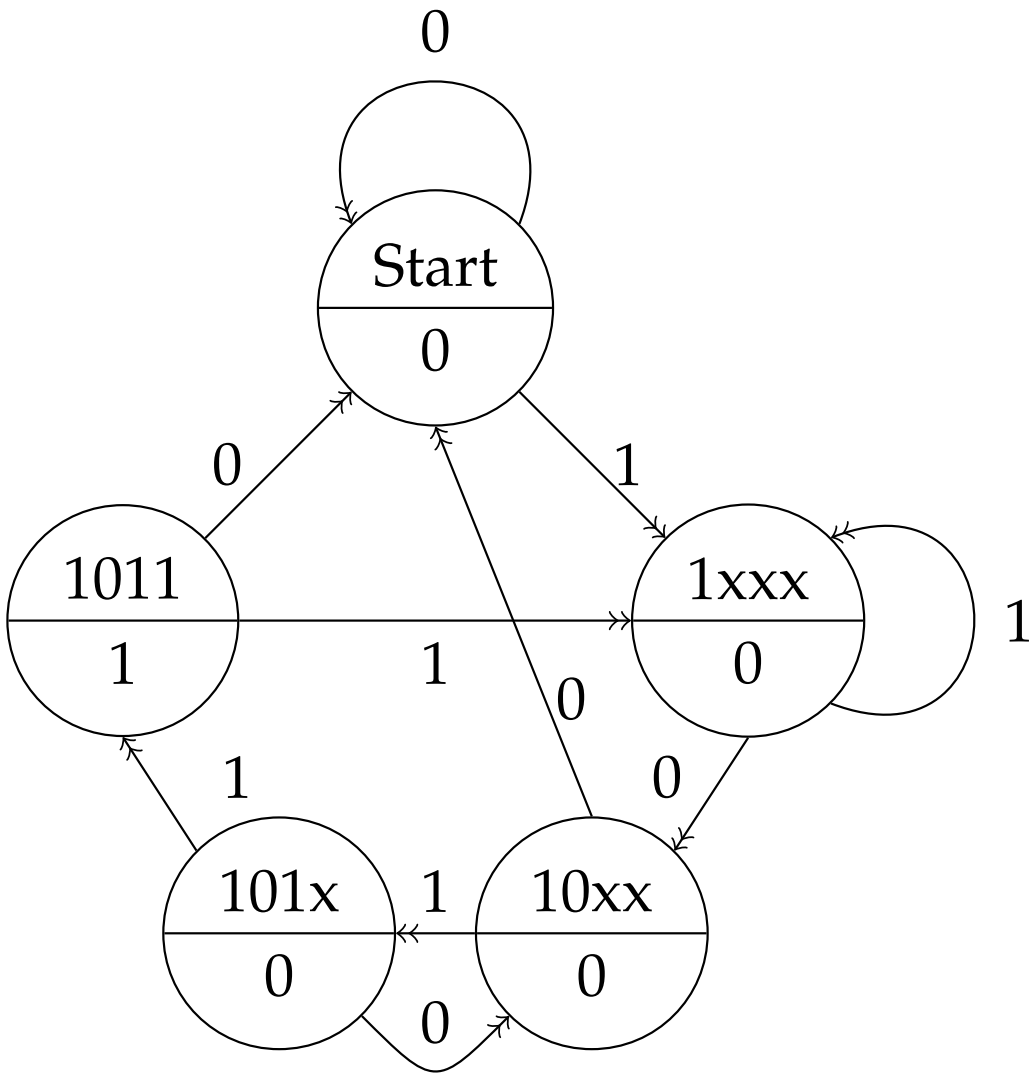
Next State Combinational Logic

1. The next state combinational logic for both a Mealy Machine and a Moore Machine has as input the current state and the control inputs.
2. As a process therefore, both the current state and the control inputs should be in the sensitivity list of the process.
3. It is usual to assign a default next state at the start of the process, this can make code shorter (and so more readable).

Next State Combinational Logic

1. The next state combinational logic for both a Mealy Machine and a Moore Machine has as input the current state and the control inputs.
2. As a process therefore, both the current state and the control inputs should be in the sensitivity list of the process.
3. It is usual to assign a default next state at the start of the process, this can make code shorter (and so more readable).
4. A case statement based on the current state is usually simplest.

The State Diagram ..



Next State CL Process


```
next_state_logic: process (current_state, a)
  begin -- process next_state_logic

    next_state <= start;

    case current_state is
      when start =>
        if a = '0' then
          next_state <= start;
        else
          next_state <= seen1;
        end if;
```

Next State CL Process


```
when seen1 =>
  if a = '0' then
    next_state <= seen10;
  else
    next_state <= seen1;
  end if;

when seen10 =>
  if a = '0' then
    next_state <= start;
  else
    next_state <= seen101;
  end if;
```

```
    when seen101 =>
      if a = '0' then
        next_state <= seen10;
      else
        next_state <= success;
      end if;

    when success =>
      if a = '0' then
        next_state <= start;
      else
        next_state <= seen1;
      end if;

    when others => null;
  end case;
end process next_state_logic;
```

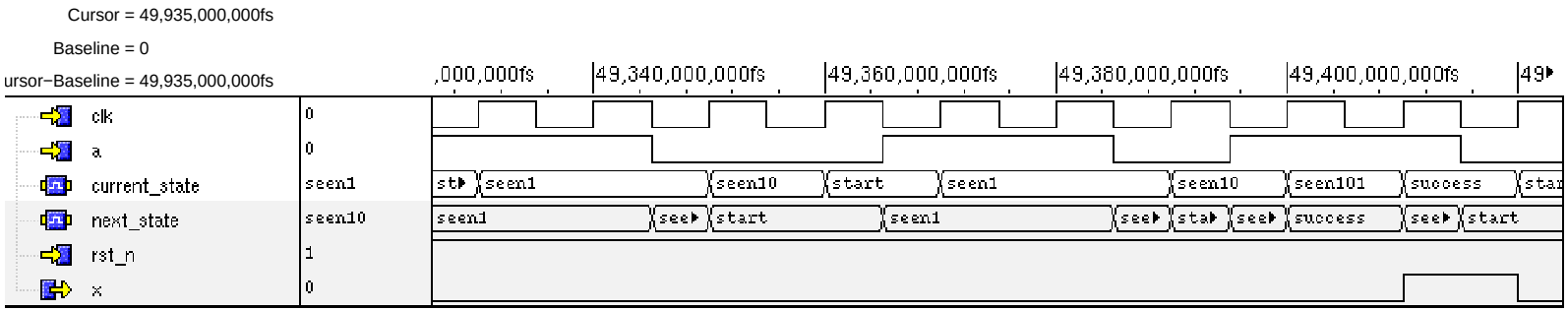
Output Logic


```
output_logic: process (current_state)
begin -- process output_logic
  if current_state = success then
    x <= '1';
  else
    x <= '0';
  end if;
end process output_logic;
```

Figure: Output Logic

Simulation of Sequence Detector

Use of these techniques does also aid in the debugging of state machines, use of an enumerated type for example makes reading a simulation plot much easier.



FSM Optimisations

1. State Reduction. For certain state diagrams some states may be redundant. There are manual methods available to remove such states, these methods are time consuming however.

FSM Optimisations

1. State Reduction. For certain state diagrams some states may be redundant. There are manual methods available to remove such states, these methods are time consuming however.
2. State Assignment. Synthesis tools however do have algorithms available providing an approximate solution to this problem.

FSM Optimisations

1. State Reduction. For certain state diagrams some states may be redundant. There are manual methods available to remove such states, these methods are time consuming however.
2. State Assignment. Synthesis tools however do have algorithms available providing an approximate solution to this problem.
3. Once coded, the algorithms may be re-applied following a modification to the design. The designer is not therefore dissuaded from further development.

FSM Optimisations

1. State Reduction. For certain state diagrams some states may be redundant. There are manual methods available to remove such states, these methods are time consuming however.
2. State Assignment. Synthesis tools however do have algorithms available providing an approximate solution to this problem.
3. Once coded, the algorithms may be re-applied following a modification to the design. The designer is not therefore dissuaded from further development.
4. In PKS, and in most tools, such optimisations are controlled by additional *attributes*.

FSM Optimisations

Here is part of the code of the sequence detector design that I actually used:

```
library ambit;
use ambit.attributes.all;

architecture rtl of detector is

  type state_name is (start, seen1, seen10, seen101, success);
  signal current_state, next_state : state_name;
  attribute STATE_VECTOR of current_state : signal is true ;
  attribute ENCODING of current_state : signal is "one_hot";

begin -- rtl
```

Cadence Approach

1. Cadence supplies a library called `ambit` that supplies the `attributes` package.

Cadence Approach

1. Cadence supplies a library called `ambit` that supplies the `attributes` package.
2. These attributes are therefore vendor dependent.

Cadence Approach

1. Cadence supplies a library called `ambit` that supplies the `attributes` package.
2. These attributes are therefore vendor dependent.
3. To have PKS extract all states in an FSM use the command:
   `do_build_generic -extract_fsm`

Cadence Approach

1. Cadence supplies a library called `ambit` that supplies the `attributes` package.
2. These attributes are therefore vendor dependent.
3. To have PKS extract all states in an FSM use the command:
   `do_build_generic -extract_fsm`
4. and then report the states with:
   `report_fsm -state`

State Coding

The attribute `ENCODING` of the signal `current_state` directs the synthesis tool to use a certain algorithm for coding of the state code. Some of the supported options are:

binary Use a sequential binary coding, 00, 01, 10, 11 etc.

State Coding

The attribute `ENCODING` of the signal `current_state` directs the synthesis tool to use a certain algorithm for coding of the state code. Some of the supported options are:

binary	Use a sequential binary coding, 00, 01, 10, 11 etc.
one_hot	Use a one-hot encoding (discussed later).

State Coding

The attribute `ENCODING` of the signal `current_state` directs the synthesis tool to use a certain algorithm for coding of the state code. Some of the supported options are:

binary	Use a sequential binary coding, 00, 01, 10, 11 etc.
one_hot	Use a one-hot encoding (discussed later).
area	Use an encoding which minimises logic area (heuristic).

State Coding

The attribute `ENCODING` of the signal `current_state` directs the synthesis tool to use a certain algorithm for coding of the state code. Some of the supported options are:

binary	Use a sequential binary coding, 00, 01, 10, 11 etc.
one_hot	Use a one-hot encoding (discussed later).
area	Use an encoding which minimises logic area (heuristic).
timing	Use an encoding for least propagation delay (heuristic).

One Hot Encoding

1. One-hot encoding uses one flip-flop per state of the machine, so more flip-flops are required.

One Hot Encoding

1. One-hot encoding uses one flip-flop per state of the machine, so more flip-flops are required.
2. One one flip-flop has its Q set at a time (the hot flip-flop), all others are at 0.

One Hot Encoding

1. One-hot encoding uses one flip-flop per state of the machine, so more flip-flops are required.
2. One one flip-flop has its Q set at a time (the hot flip-flop), all others are at 0.
3. Resultant state encoding:

State	Code
start	10000
seen1	01000
seen10	00100
seen101	00010
success	00001