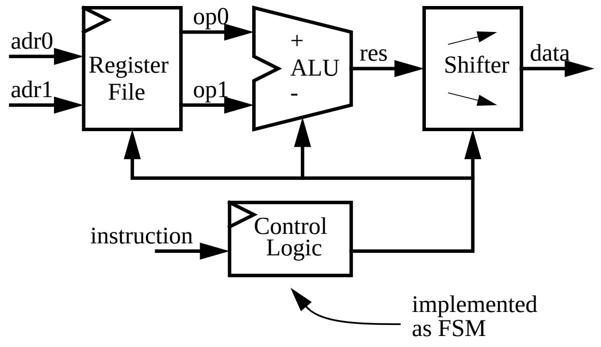
Chapter III: Finite State Machines

Machine Models

The most difficult part in designing hardware is to correctly implement the **control logic** of a model. Several data-transforming or data-storing units (adder, multiplier, shifter, register file, ...) have to be supplied with control signals at the right time, so the whole implementation behaves as expected:



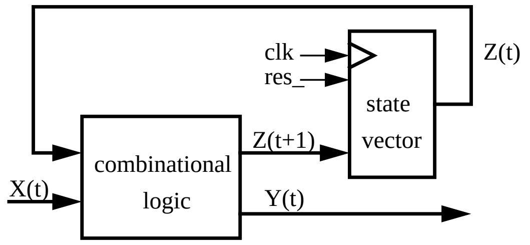
Finite State Machines (**FSM**) provide an easy-to-use model for defining, testing and implementing complex control circuits. They have been derived from the theory of finite automata. An FSM consists of a sequential part storing the current state of the machine in registers, and a combinational part for evaluation of the output signals and the next state from the input signals and the current state. The machine can change its state on every active clock edge.

Chapter III: Finite State Machines

Machine Models

Let X(t) be the vector of input signals at step t, Y(t) the output signals and Z(t) the state. Three slightly different models are known:

Mealy-FSM:



The next state and the outputs are both a function of the current state and the inputs:

$$Z(t+1) = f(Z(t), X(t))$$

$$Y(t) = g(Z(t), X(t))$$

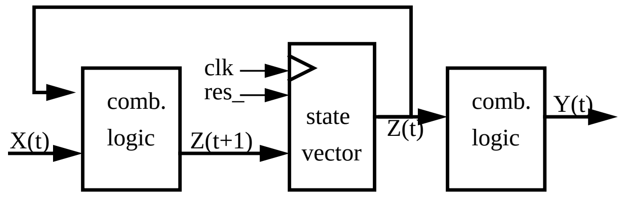
Though this model offers the greatest flexibility, the **asynchronous behaviour** of the output signals is a great disadvantage. A transition of an input signal might cause a delayed transition of some output signals resulting in a highly unpredictable behaviour.

There are a few special cases where the Mealy-FSM must be used in order to react very quick (without one clock delay) on changed input signals, e.g. gating of clock signals for synchronous modules.

Chapter III: Finite State Machines

Machine Models

Moore-FSM:

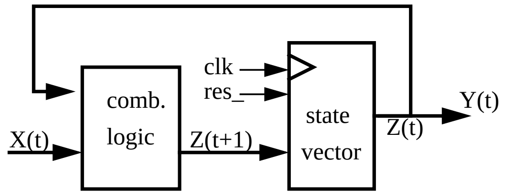


Now the output is only a function of the current state:

Y(t) = g(Z(t))

Output signals are no longer affected by transitions of inputs. Only a small delay relative to the active clock edge is added from the combinational logic for generating the output signals from the current state vector.

Simple Moore-FSM:



When choosing the output signals as state vector, a **full synchronous** design can be implemented. This model offers the **fastest** implementation, but the designer has to do the state encoding explicitely by himself. In some cases it is necessary to add dummy bits to the state vector to ensure a unique coding for states with equal outputs.

Chapter III: Finite State Machines

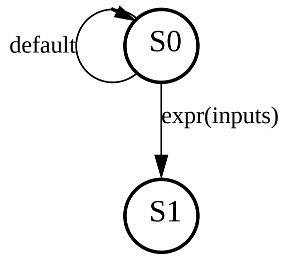
Graphical Representation

State Encoding:

Different states are represented by a **unique bit vector**. Assigning these bit vectors to states is known as state encoding. Two different approaches are possible:

- **One-Hot**: every bit of the state vector identifies a unique state. So there are as many bits as states have to be coded. This method enables a **quick** decoding of the current state in the combinational logic evaluating the next state and the outputs, but is only suitable for machines with few states. One-Hot coding is mostly incompatible with the Simple Moore-FSM model.
- **Multi-Bit**: a bit vector can be assigned to every state by simple binary enumeration of all states. Sometimes it can be useful to choose an arbitrary coding with a simple decoding scheme in mind (or gray code, etc.). For a Simple Moore-FSM the state bits correspond directly to the outputs. If different states have the same output values, dummy bits have to be added to the state vector to ensure a unique coding.

Analogous to the specification of finite automata, FSMs can be described graphically. States are represented by named circles, and transitions between them by directed arcs with conditional expressions enabling the transition on the active edge of the clock signal.



Some conditions have to be observed:

- the expressions of different arcs have to be disjoint
- all possible combinations of inputs have to be covered by the arc expressions of a given state. If this is not the case, a special default arc has to be added to ensure a complete specification of the function evaluating the next state

Chapter III: Finite State Machines

Textual Representation

A Finite State Machine can also be described textually by defining the state names and their corresponding bit vectors in a so called **state table**. The transitions are listed in a separate **flow table** (- means *don't care*).

state table

name	coding
idle	000
start	001
⋮	⋮

flow table

current state	inputs	next state
idle	01	start
idle	1-	idle
⋮	⋮	⋮

The combinational function for every state bit can be simply derived from these tables by collecting all rows with the bit set in the 'next state' column. All bits from the 'current state' and 'inputs' columns combined by a logical AND form one term of the function. A logical OR of all terms results in the complete function for a given state bit.

Example: A 2bit-counter with enable (active-high) and reset_ (active-low) signals could be implemented in the following way:

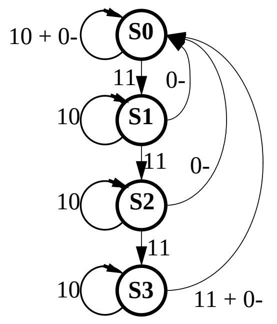
flow table

current	(reset_, enable)	next	terms bit [1]	[0]
S0	11	**S1**		+
S0	10	**S0**		
S0	0-	**S0**		
S1	11	**S2**	+	
S1	10	**S1**		+
S1	0-	**S0**		
S2	11	**S3**	+	+
S2	10	**S2**	+	
S2	0-	**S0**		
S3	11	**S0**		
S3	10	**S3**	+	+
S3	0-	**S0**		

minterms for bit [x]

state table

name	coding [1:0]
S0	00
S1	01
S2	10
S3	11



Chapter III: Finite State Machines

Verilog Coding of FSMs

The Verilog code for Finite State Machines complies with the block diagram of the machine model:

- one *always*-block is responsible for the **combinational logic** evaluating the next state. The trigger is formed from the input and state signals. A *case*-statement lists all possible states and the corresponding transitions. Two nested *case*-statements can also be used, if the number of transitions is too large:

```
always @(enable or state)
      case (state)
          S0: if (enable)
                  next_state = S1;
              else
                  next_state = S0;
          S1: ...
      endcase
```

- one *always*-block latches the new state into the registers. The active edge of the clock signal triggers the block. Also the reset is considered here:

```
  always @(posedge clk or negedge reset_)
      if (reset_ == 1'b0)
          state <= S0;
      else
          state <= next_state;
```

- an additional block implements the output function for Mealy or Moore machines

Some **guidelines** must be observed for generating efficient, synthesizable code:

- don't use inout signals in FSMs
- try to partition complex FSMs into smaller independent machines
- use a reset signal to initialize the machine correctly
- fully specify a *case*-statement; if not all cases are covered, make use of the *default*-case

Chapter III: Finite State Machines

Verilog FSM Example

The following code is an implementation of the 2bit counter:

```
module count2bit(cnt, clk, res_, en);
output [1:0] cnt;  // the counter value
reg [1:0] cnt;
input clk, res_, en;
reg [1:0] next_cnt;  // the next state

parameter S0=2’b00,  // the states
          S1=2’b01,
          S2=2’b10,
          S3=2’b11;

always @(posedge clk or negedge res_)
   if (res_ == 1’b0)
      cnt <= 2’b00;
   else
      cnt <= next_cnt;

always @(en or cnt)
   case (cnt)
   S0: if (en == 1’b1)  // full if-then-else implem.
         next_cnt = S1;
      else
         next_cnt = S0;
   S1: if (en == 1’b1)  // the else path is not needed
         next_cnt = S2;
   S2: next_cnt = (en == 1’b1) ? S3 : S2;  // shorter
   S3: next_cnt = (en) ? S0 : S3;  // even shorter
   endcase

endmodule
```