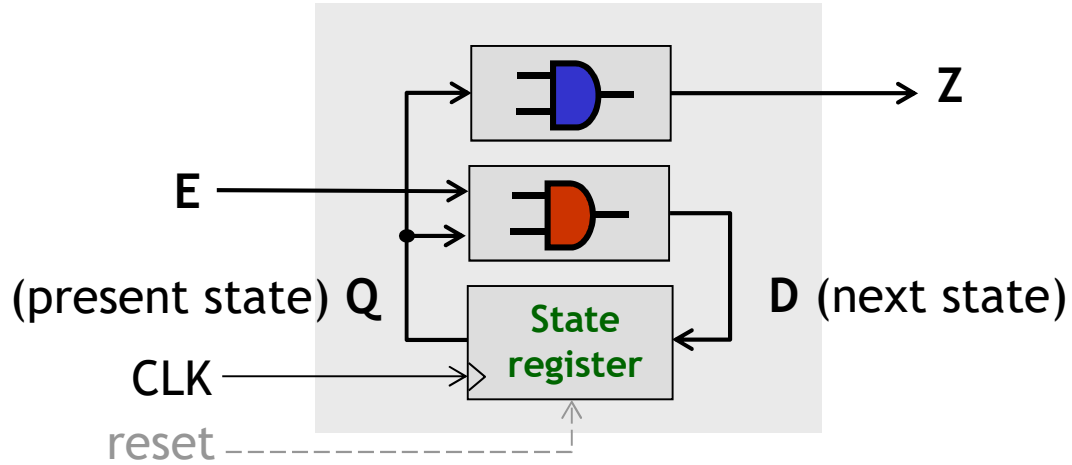
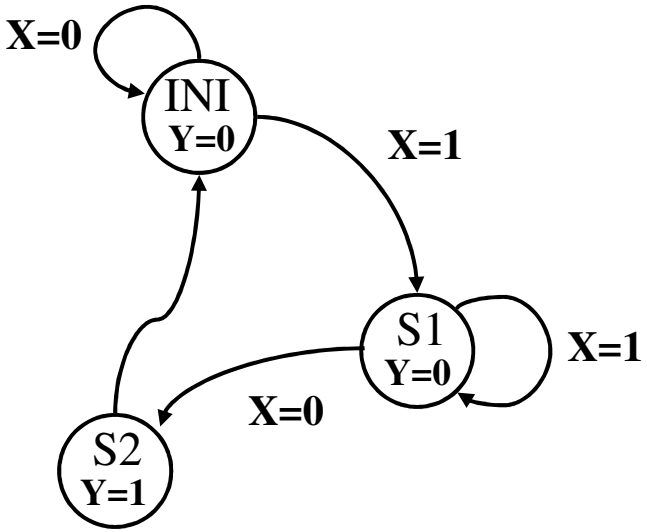
Finite state machines
(Moore model)

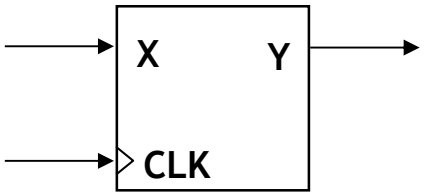


- **State register**
  - Loads the next state (D input) when the clock triggers (single register)
- **Next state combinational logic**
  - Generates the next state **D** from the present state **Q** and the inputs **E**
- **Output logic**
  - Generates the outputs **Z** from the current state **Q** (Moore model)

```
module fsm( input CLK, input RST, input E, output Z);
reg  [1:0] STATE;     // state register
reg  [1:0] NEXTSTATE; // next state (this will not be a register)

always @(posedge CLK or posedge RST)
begin
  if ( RST )
    STATE <= 2'b00;
  else
    STATE <= NEXTSTATE;
end

always @* // next state combinational logic
  case ( STATE )
    // defines NEXTSTATE using state and E
  endcase

always @*  // output logic
  case( STATE)
    // defines the outputs Z with STATE
  endcase
endmodule
```

Example





- 3 states (INI is the initial state)
  - The minimum number of bits to encode all states is 2
- input X
- output Y depending only on the current state (Moore model)

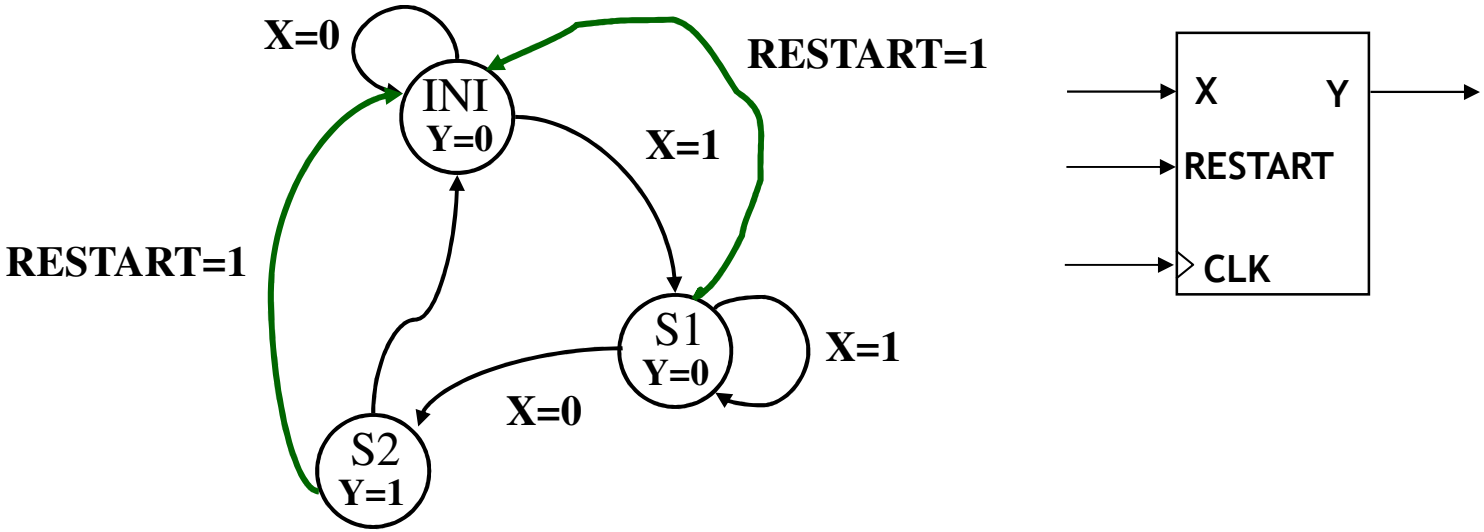
Something is wrong here!

```
module fsm( input CLK, input RST, input X, output reg Y);
reg  [1:0] STATE;      // current state
reg  [1:0] NEXTSTATE;  // next state
parameter INI = 2'b00, // symbolic state names
          S1  = 2'b01, //
          S2  = 2'b11;
always @(posedge CLK or posedge RST) // state register
begin
  if ( RST )
    STATE <= INI; else STATE <= NEXTSTATE;
end
always @*  // NEXTSTATE logic
  case ( STATE )
    INI: if ( X == 1'b0 ) NEXTSTATE = INI; else NEXTSTATE = S1;
    S1 : if ( X ) NEXTSTATE = S1; else NEXTSTATE = S2;
    S2 : NEXTSTATE = INI;
  endcase
always @*  // output logic
  case( STATE)
    INI: Y = 0;
    S1 : Y = 0;
    S2 : Y = 1;
  endcase
endmodule
```

A different way to define the output Y

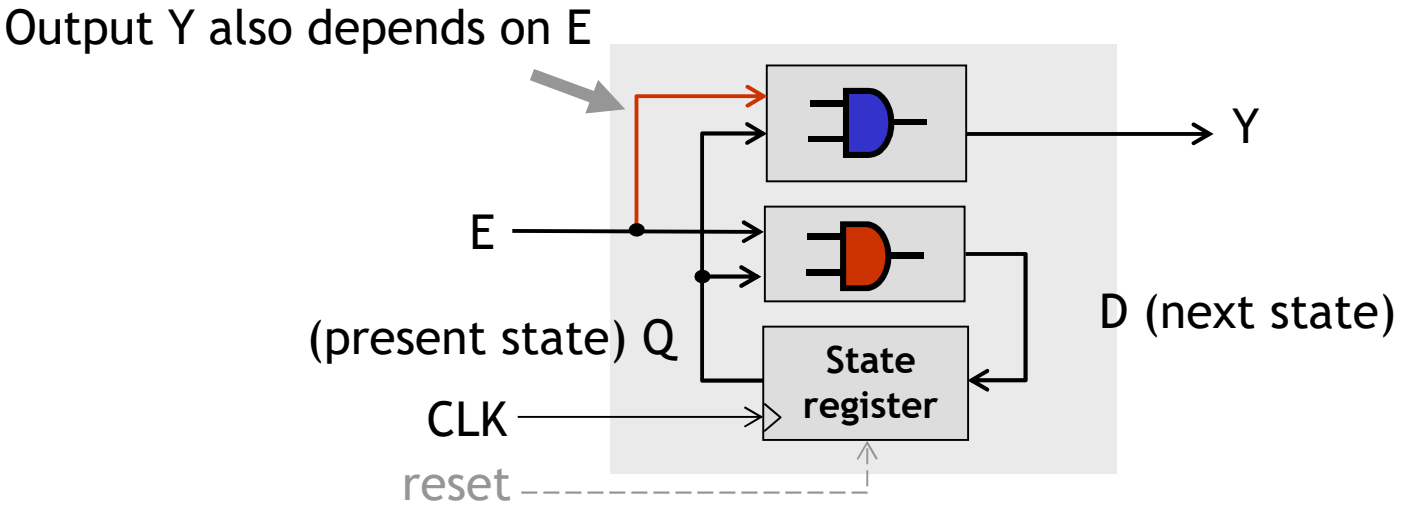
```
module fsm( .  .  . );
reg  [1:0] STATE;
reg  [1:0] NEXTSTATE;
parameter INI = 2'b00,
          S1  = 2'b01,
          S2  = 2'b11;

...

assign Y = ( STATE == S2 ); // Y is 1 when STATE is equal to S2

endmodule
```

How to add a ‘restart’ input (synchronous *reset*)



```
module fsm( .   .   . );

. . .

always @(posedge CLK or posedge RST) // state register
begin
  if ( RST )
    STATE <= INI; else STATE <= NEXTSTATE;
end
always @* // NEXTSTATE logic
  if ( RESTART ) NEXTSTATE = INI;
  else
    case ( STATE )
     INI: if ( X == 1'b0 ) NEXTSTATE = INI; else NEXTSTATE = S1;
     S1 : if ( X ) NEXTSTATE = S1; else NEXTSTATE = S2;
     S2 : NEXTSTATE = INI;
    endcase

always @*  // output logic
  ...
endmodule
```

Mealey finite state machines



```
always @* // next state logic
  case ( STATE )
    INI: if ( X ) NEXTSTATE = S1; else NEXTSTATE = INI;
    S1 : if ( X ) NEXTSTATE = S1; else NEXTSTATE = S2;
    S2 : if ( X ) NEXTSTATE = INI; else NEXTSTATE = S2;
  endcase
```

```
always @*  // output logic
  case( STATE )
    INI: Y = 0;
    S1 : if ( X ) Y = 0; else Y = 1;
    S2 : if ( X ) Y = 1; else Y = 0;
  endcase
```

```
endmodule
```

Mealy or Moore ?

```
always @(posedge clock)
if ( reset )
  state <= INI;
else
  case ( state )
    INI: if (~X )
           state <= INI;
         else
           state <= S1;
    S1:  if ( X )
           state <= S1;
         else
           state <= S2;
    S2:  state <= INI;
  endcase


assign Y = ( state == S2 & X );
```

```
always @(posedge clock)
if ( reset )
  begin
    state <= INI;
    Y <= 1'b0;
  end
else
  case ( state )
    INI: if (~X )
         begin
           state <= INI;
           Y <= 1'b0;
         end
         else
         begin
           state <= S1;
           y <= 1'b1;
         end
    ...
  endcase
```

Verilog operators (similar to C operators)

	operador	#opr	notes
arithmetic	+ - * / %	2	sign extension if signed, zero extension if unsig.
logic	! && ||	1 2 2	logic NOT / AND / OR: zero means false, other is true returns 1 (true) or 0 (false)
relational	> < >= <=	2	different behaviour for signed and unsigned
equality	== != === !==	2 2	0-1 comparision (compare with 'x' is alws. false) also compare 'z' and 'x'
bitwise	~ & | ^	1, 2	signed/unsigned extension
shift	>> <<	2	logical shift (shift right inserts zeros at the left)
arithm. shift	>>> <<<	2	arithmetic shift (shift right keeps the sign bit)
concatenation	{ }	N	`{3'b101,1'b0,3'b111}=7'b1010111`
replication	{{ }}	N	`{N{A}}` replicates `N` times the bit pattern `A`
conditional	<cond> ? True : False	3	as in C

Numeric constants

- Default is integer, 32 bit, twos-complement

```
reg [3:0] a;
...
initial
begin
  a = 28; // 28=11100 -> a is loaded with 1100=12₁₀
...
end
```

- Defining the number of bits and numeric base (recommended):

```
5'd10;             //  5 bits, decimal
10'b1010_0011_11; // 10 bits, binary
16'h1E_C6;         // 16 bits, hexadecimal
```

value ("_" can be used as separator for readability)

numeric base

number of bits

Signed and unsigned

- Wires and regs are handled by default as unsigned numbers
  - **`wire [31:0] x, y, z; reg [14:0] r1, r2;`**
  - Sign extension is not performed when increasing the number of bits
  - Comparisions of magnitude are done as unsigned numbers
- Any wire, register in or out can be declared as **`signed`**
  - **`wire signed [31:0] a, b;  reg signed [3:0] ra, rb;`**
  - Arithmetic expressions should contain only signed or unsigned
  - The function **`$signed( ... )`** converts unsigned to signed

```
ACCUM0:
   begin  // Accumulate the LEFT slices
     accumLL <= accumLL + gainLL * $signed( dataoutL1_L );
     accumRL <= accumRL + gainRL * $signed( dataoutR1_L );
     state <= ACCUM1;
   end
```

Verilog *tasks*

- Main purpose: “encapsulate” simulation tasks
  - May contain delays and time synchronization
  - Much like a C function, without returning a value

```
task writecommand;
input [15:0] data;     // data to write
input [1:0] address; // address of output port
begin
    din = {4'b0010, 2'b00, address}; // compose and write command
    load = 1;  #10 load = 0;  #10  // ld for 1 clock
    din = data[15:8]; // send MS byte
    load = 1;  #10 load = 0; #20  // ld for 2 clocks
    din = data[7:0]; // send LS byte
    load = 1;  #10 load = 0;  #30  load = 0;
  end
endtask
```

```
initial
begin
  …
  writecommand(16'h39A6, 2'b00 );
```

Two good references for Verilog HDL

- www.asic-world.com
- www.chipverify.com/verilog/verilog-tutorial