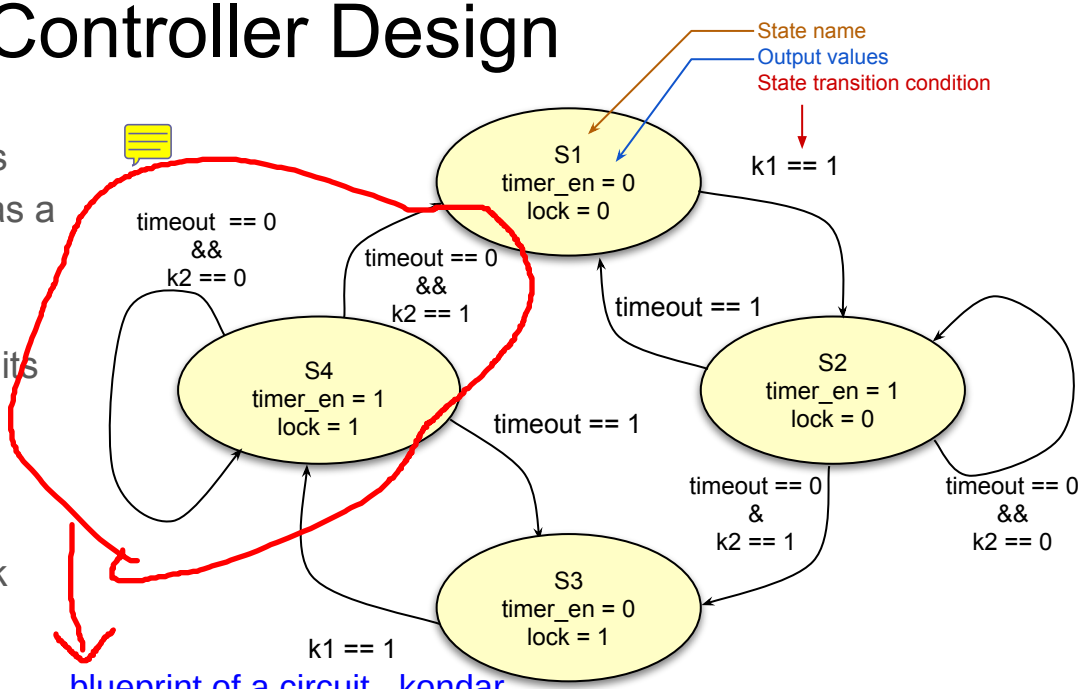
Digital Techniques 2

L8: Sequential Logic 3

UNIVERSITY OF OULU
CIRCUITS and SYSTEMS

Finite-State Machine Controller Design

- FSM controllers are sequential circuits whose function is commonly defined as a **state chart** or table
- A state chart defines a circuit's output values and next state as a function of its inputs and current state
- This *abstraction* is well-suited for specification of control logic
- Example: Classic Nokia keyboard lock
  1) Press key 1
  2) Press key 2 within 1.5 seconds
- Control logic can be modeled as an FSM
- A counter can be used to measure time

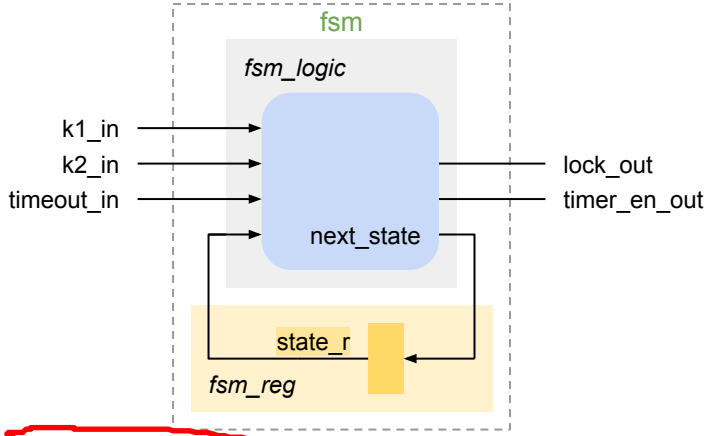


FSM Modeling for Synthesis: Two-Process Template

```
module fsm
  (input logic clk, rst_n, k1_in, k2_in, timeout_in,
   output logic lock_out, timer_en_out);

 enum logic [1:0] {S1 = 2'b00, S2 = 2'b01,
                   S3 = 2'b10, S4 = 2'b11 } state_r, next_state;

always_comb
  begin : fsm_logic
    lock_out = '0; timer_en_out = '0; next_state = S1;
    case (state_r)
      S1:
        begin
          lock_out = '0;
          timer_en_out = '0;
          if (k1_in == '1)
            next_state = S2;
          else
            next_state = S1;
        end
       // Case items for other states...
    endcase
  end : fsm_logic
```



See next slide for an explanation for this line.

```
always_ff @(posedge clk or negedge rst_n)
  begin : fsm_reg
    if (rst_n == '0)
      state_r <= S1;
    else
      state_r <= next_state;
  end : fsm_reg

endmodule
```

If you put the *case* statement here, you still need an *always_comb* with another *case* for the output decoder!

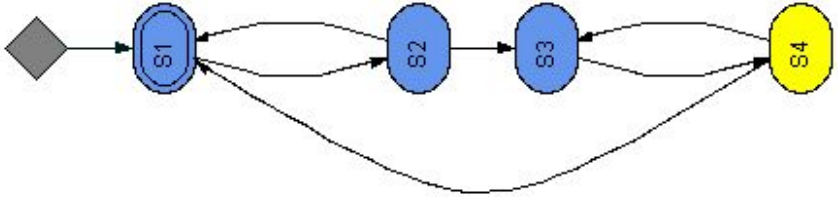
jodi kono current state change hoi taile aida execute hbe

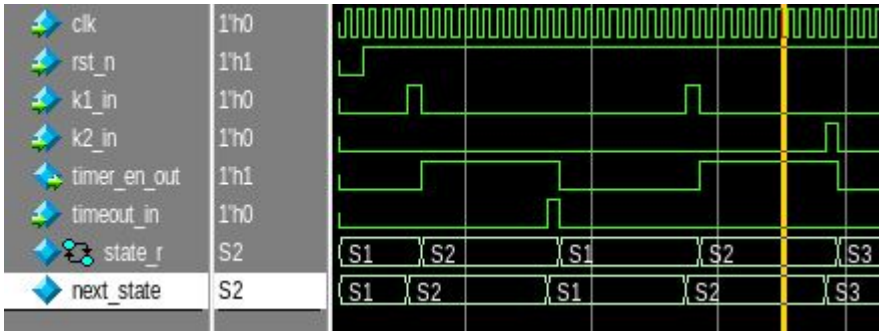
ata current state ar value hold kore

About FSM Coding Styles

- There are advantages of using the two-process template: Many EDA tools recognize this coding style and can visualize the code as a state chart
- Coding style is the same for all FSMs, and the combinational part can be modelled using one **case** statement
- Use of **enumeration type** for state variable makes FSM debugging easier compared to use of binary codes
- **Assign default values** to FSM outputs in the beginning of the *always_comb* is a good way to avoid latches (FSM code can become very complicated)

FSM visualized in QuestaSim

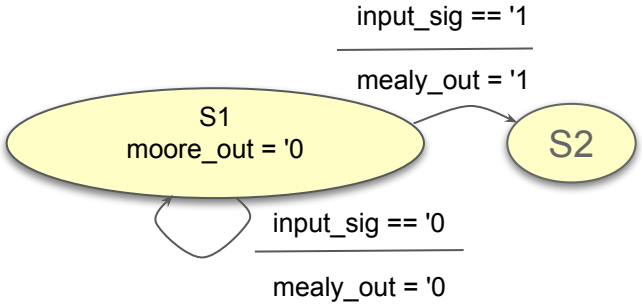




Enumeration data types in simulation.

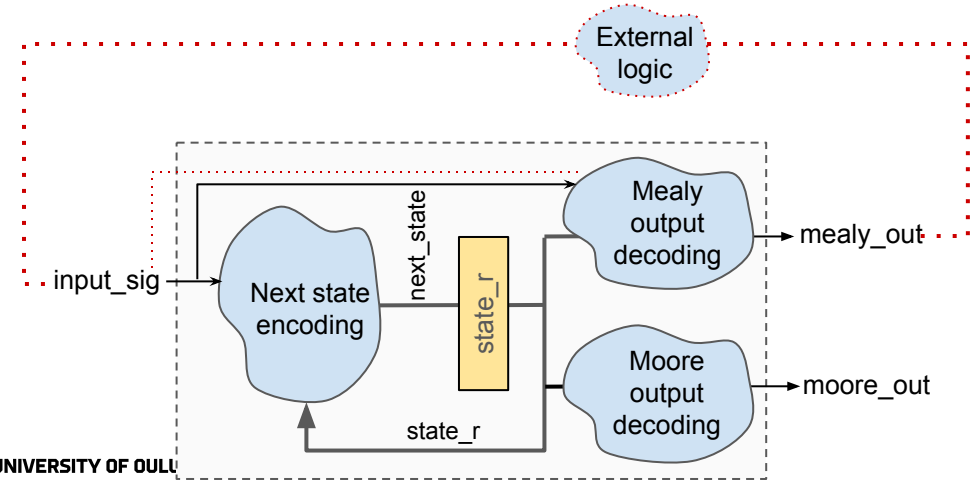
Modeling Moore- and Mealy-Type Outputs

- Mealy outputs depend on current state and inputs, while Moore outputs only depend on current state
- In practice, Mealy- and Moore-type behavior is usually combined in FSMs into one always_comb
- Mealy-type control is fast, but it can be risky



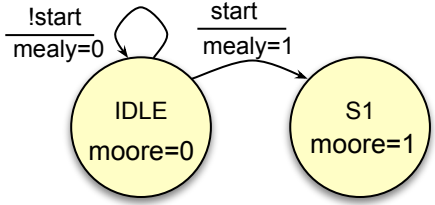

```
case (state_r)
  S1:
    begin
     moore_out= '0;
     if (input_sig == '1)
       begin
         next_state = S2;
         mealy_out = '1;
       end
     else
       begin
         next_state = S1;
         mealy_out = '0;
       end
    end
```

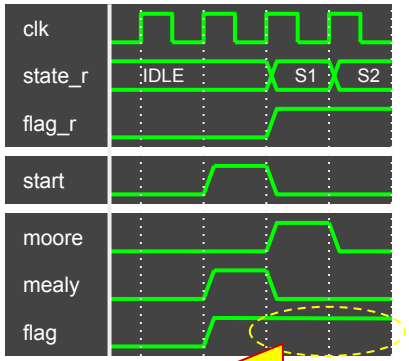
Combinational loop can be accidentally created when a Mealy-controller is connected to a larger design.

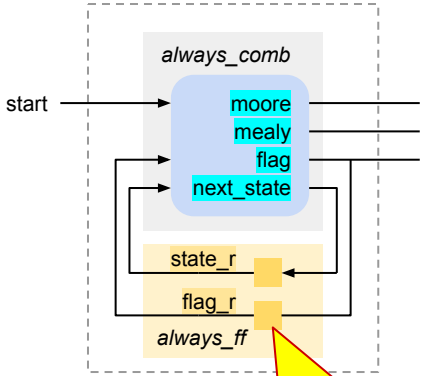


Moore, Mealy and "Flag" Type Control

- In response to some event, Moore outputs have a one-cycle delay while Mealy outputs respond immediately
- Having a **flag** type output that "remembers" its state saves coding effort in controllers (you can define a new value only when it changes). It must be implemented with an additional flip-flop (flag_r)








```
always_comb
begin
 moore = '0;
 mealy = '0;
 flag = flag_r;
 next_state = state_r;
 case (state_r)
   IDLE:
    if (start) begin
     flag = ~flag_r;
     mealy = '1;
     next_state = S1;
    end
   S1:
    begin
     moore = '1;
     next = S2;
    end
   S2:
    begin
     moore = '0;
     next = S3;
    end
    ...
```

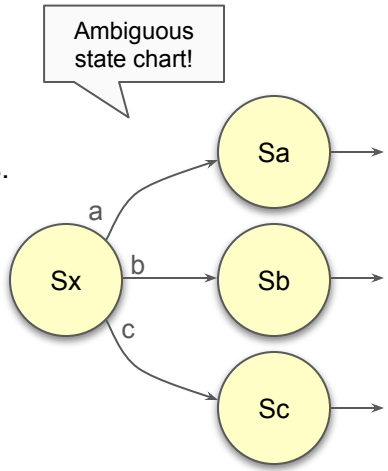
flag *toggled* by **start,** stays on in **S1** and **S2** with no assignments in code.

An additional state bit must be allocated!

The Trouble with State Charts

- State charts are widely used in design, but they are often used in an *ad hoc* manner
- A specification should define the function unambiguously, but drawing a state chart that defines the function completely is tedious
- This leads to implicit assumptions about the function or input data to be made (example)
- In the example, the implicit assumptions could be either:
  A) Input data is one-hot encoded
  (only one of a,b,c is true at a time)
  B) The priority order of inputs is a,b,c
- Mistakes can be made if RTL code is written by another person that does not know these assumptions

Example: 3 inputs a,b,c
→ 8 possible combinations.
A *complete* chart would have 8 state transitions or complete transition condition functions (e.g. a·b'·c')



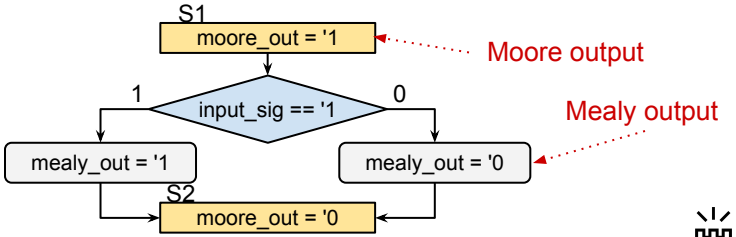
A ? B

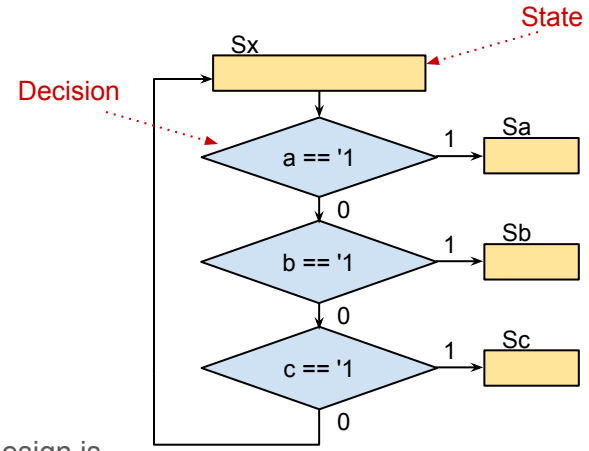
```
case ({a,b,c})
  3'b100:  next_state = Sa;
  3'b010:  next_state = Sb;
  3'b001:  next_state = Sc;
  default: next state = Sx:
endcase
```

```
if (a == '1)
  next_state = Sa;
else if (b == '1)
  next_state = Sb;
else if (c == '1)
  next_state = Sc;
else
  next_state = Sx;
```

Algorithmic State Machine (ASM) Descriptions

- Basic-form ASMs drawing rules:
  - **Each state has only one output transition** that goes to a two-way (yes-no) decision box or to another state
  - Decision boxes can be chained to form complex decision structures that can be coded with an **if - else if - ... - else** statement
- ASM forces the designer to consider all possible cases
- Moore and Mealy output definition:





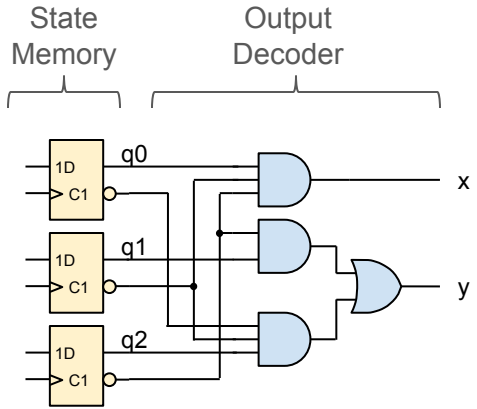
RTL code design is easy: there is only one way the code an ASM chart!

```
case (state_r)
  Sx:
    if (a == '1)
      next_state = Sa;
    else if (b == '1)
      next_state = Sb;
    else if (c == '1)
     next_state = Sc;
    else
      next_state = Sx;
  Sa:
```

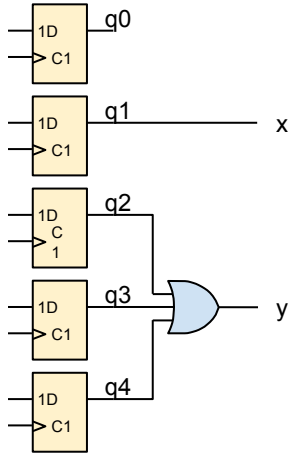
FSM State Encoding

- Assignment of state codes affects circuit complexity, performance and power consumption
- **Maximal encoding** minimizes number of flip-flops (e.g. binary code: 000, 001, 010)
- **One-hot** encoding yields simple combinational parts (e.g. 0001, 0010, 0100, 1000)
- **Gray-encoding** can minimize switching power caused by state changes (example)
- Example on the right =>:
  - FSM with states s0, s1, s2, s3, s4
  - Output x *true* in state s1 and y *true* in state s2, s3, s4
  - Case A: Binary state encoding (000, 001, ...)
  - Case B: One-hot state encoding (00001, 00010, 00100, ...)
- You can define state encoding in code by assigning values to enum constants, or you can use synthesis tool settings

Case A) Binary encoding



Case B) One-hot encoding



One-hot encoding is favoured in FPGAs that often have an abundance of flip-flops.