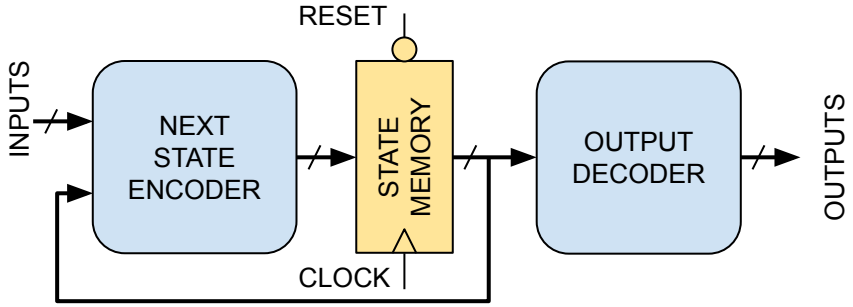
Digital Techniques 2

L6: Sequential Logic 1

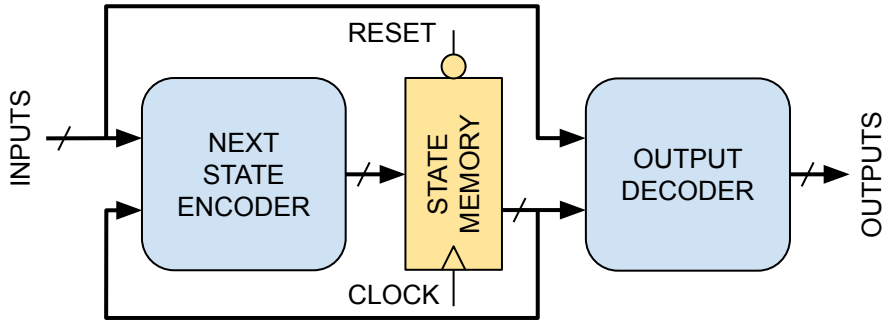
UNIVERSITY OF OULU
CIRCUITS and SYSTEMS

Sequential Circuits

- Outputs of a sequential circuit depend on the current and/or past states of its inputs
- Parts of a sequential circuit:
  - **State memory register** into which an encoded function of the circuit's current inputs and current internal state is stored in on the rising edge of a clock signal
  - Combinational **next-state encoder** that computes the next-state code to be stored in the state memory
  - Optional combinational **output decoder** that computes the output values of the circuit from the contents of the state memory (Moore machine) or from the contents of the state memory and the input states (Mealy machine)



Moore machine



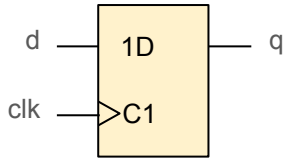
Mealy machine

Modeling Edge-Sensitive Storage Devices for Synthesis

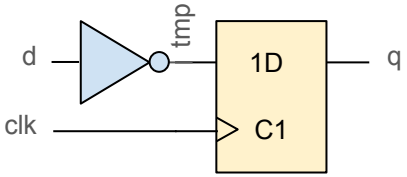
From IEEE Synthesis Standard 1364.1-2002

- An edge-sensitive storage device shall be modeled for a **variable that is assigned a value in an always statement** that has **exactly one edge event in the event list**. The edge **event specified shall represent the clock edge** condition under which the storage device stores the value.
- **Non-blocking procedural assignments** (<=) should be used for variables that model edge-sensitive storage devices.
- Blocking procedural assignments (=) may be used for variables that are temporarily assigned and used within an always statement.

```
logic clk, d, q;

always @ (posedge clk)
  q <= d;
```

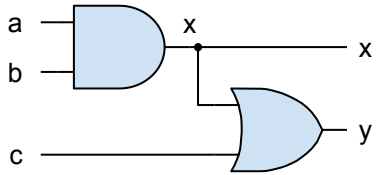



```
logic clk, d, q;

always @ (posedge clk)
  begin
    logic tmp;
    tmp = ~d;
    q <= tmp;
  end
```

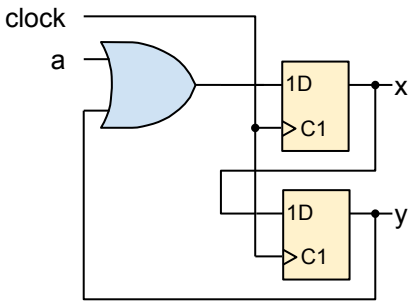


Why Are Two Procedural Assignment Types Needed?

- **Blocking** and **non-blocking assignments** have different scheduling semantics that allow modeling of different kind of behaviour
- An easy way of think of them is that...
  - Blocking assignments (=) take effect immediately (just like logic gates)
  - Non-blocking assignments (<=) take effect all at the same time at the end of current simulation time step (just like flip-flops all change on clock edges)
- …or, inside an always procedure...
  - Variables assigned with = change state immediately
  - Variable assigned with <= change state when the end of the procedure has been reached




```
always @(a, b, c)
  begin
    x = a & b; // x updated
    y = c | x; // new x used
  end
```




```
always @(posedge clock)
begin
  x <= a | y;
  y <= x; // old x used
end       // x,y updated here
```

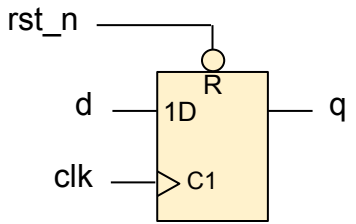
Modeling Edge-Sensitive Storage Devices with Asynchronous Reset for Synthesis

From IEEE Standard 1364.1-2002

- An edge-sensitive storage device with an asynchronous set and/or asynchronous reset is modeled using an always statement whose **event list contains edge events representing the clock and asynchronous control variables**. Level-sensitive events shall not be allowed in the event list of an edge-sensitive storage device model.
- Furthermore, the always statement shall contain an **if statement to model the first asynchronous control** and optional nested else if statements to model additional asynchronous controls. A **final else statement, which specifies the synchronous logic portion** of the always block, shall be controlled by the edge control variable not listed in the if and else if statements

```
logic clk, rst_n, d, q;

always @ (posedge clk or negedge rst_n)
 begin
   if (rst_n == '0)
     q <= '0;
   else
     q <= d;
   end
end
```

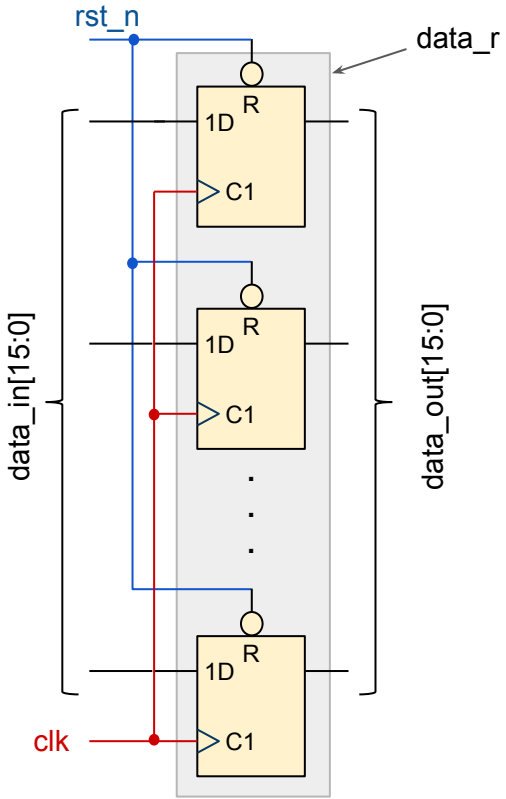
Flip-flop with an R-pin is chosen because the reset state is '0. A FF with an S-pin would be used if the reset state were '1.



SystemVerilog Model of a Register

```
module clocked_register
 (input logic          clk,
  input logic          rst_n,
  input logic  [15:0] data_in,
  output logic [15:0] data_out);

  logic [15:0]        data_r; // Register modelled with a variable

  always @ (posedge clk or negedge rst_n)
    begin
       if (rst_n == '0)
         data_r <= '0;          // All bits (flip-flops) reset at once
       else
         data_r <= data_in;     // Load next state code to register
    end

  assign data_out = data_r;     // Connect register bits to outputs

endmodule
```

It's a good idea to mark variables that represent registers with some suffix so that you remember to handle them as registers in your code...



Register Inference in Logic Synthesis and **always_ff**

- Synthesis programs **infer** registers from code
- Inference is based on matching the code against the standard coding rules for edge-sensitive storage devices
- Synthesis tool allocates the required number and kind (clock edge, reset or set type) of flip-flops for each "clocked" variable
- Connections to flip-flops' data, clock and asynchronous reset and set pins are made according to the code

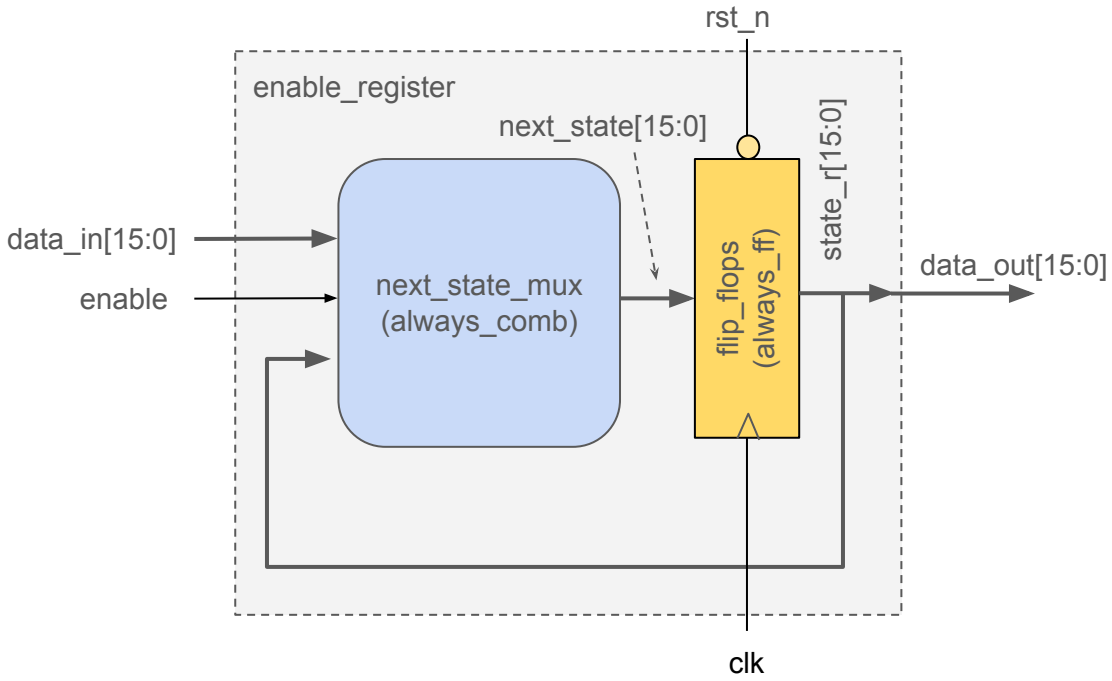
- After synthesis, first check the register inference reports from the log:

```
Inferred memory devices in routine clocked_register line
31 in file register.sv.
=============================================================
| Register Name |   Type    | Width | Bus | MB | AR | AS |
=============================================================
|  data_r_reg   | Flip-flop |  16   |  Y  | N  | Y  | N  |
=============================================================
MB = multi-bit flip-flop used
AR = asynchronous reset
AS = asynchronous set
```

SystemVerilog has an **always_ff** process that tells synthesis tools to check that the process really generates edge-sensitive flip-flops: use it instead of *always*!

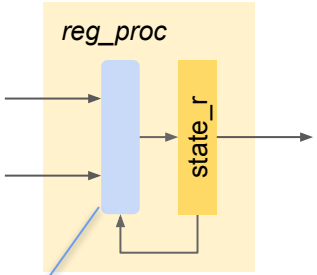
Enabled Register Modeled with Two Processes

```
module enabled_register
  (input logic clk, rst_n, enable,
   input logic  [15:0] data_in,
   output logic [15:0] data_out);

  logic [15:0] state_r, next_state;

  always_comb
    begin : next_state_mux
       if (enable == '1)
         next_state = data_in;
       else
         next_state = state_r;
    end : next_state_mux

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

  assign data_out = state_r;
endmodule
```



Enabled Register with One Process

```
module enabled_register
  (input logic clk, rst_n, enable,
   input logic  [15:0] data_in,
   output logic [15:0] data_out);

  logic [15:0] state_r;

  always_ff @ (posedge clk or negedge rst_n)
    begin: reg_proc
       if (rst_n == '0)
         state_r <= '0;
       else
          begin
            if (enable == '1)
              state_r <= data_in;
          end
    end : reg_proc

  assign data_out = state_r;
endmodule
```

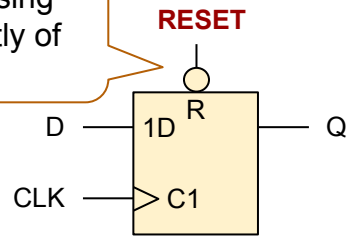


Code written inside this 'clocked region' that determines **the value to be assigned to registered variables** on rising edge of clock, **defines the next state encoding combinational logic** of those registers.
Rest of the code is just 'syntactic fluff' that has to be there!
You can write any code block in this region to define the register's next state.
The registered variable does **not** have to be assigned in all cases (therefore there is no *else* branch…)

Modeling Reset Behaviour

ASYNCHRONOUS RESET

Flip-flop is reset using R pin independently of clock.



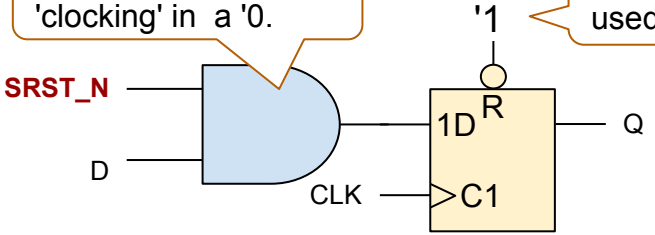

```
always_ff @(posedge CLK or negedge RESET)
  if (RESET  == '0)
    Q <= '0;
  else
    Q <= D;
```

Q <= '1 would connect the S (set) pin of the FF

SYNCHRONOUS RESET

Flip-flops is reset by 'clocking' in a '0.

R pin is not used.



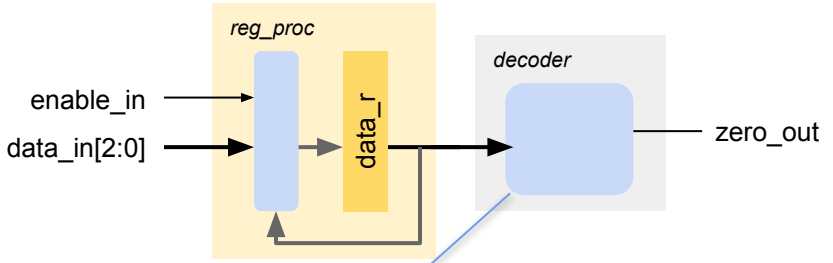

```
always_ff @(posedge CLK)
  if (SRST_N == '0)
    Q <= '0;
  else
    Q <= D;
```

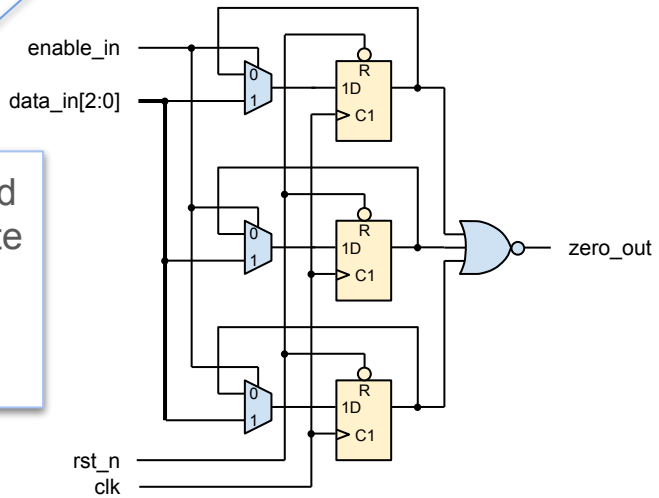
Only CLK starts this process.

Important! Never initialize module variables (logic my_reg = '0), it will **not** generate reset logic but hides missing resets in simulation.

Modeling Output Decoders

```
module output_decode
  (input logic clk, rst_n, enable_in,
   input logic [2:0] data_in,
   output logic      zero_out);

   logic [2:0]        data_r;

   always_ff @ (posedge clk or negedge rst_n)
     begin : reg_proc
       if (rst_n == '0)
         data_r <= '0;
       else
         if (enable_in == '1)
          data_r <= data_in;
     end : reg_proc

   always_comb
     begin : decoder
       if (data_r == 3'b000)
         zero_out = '1;
       else
         zero_out = '0;
     end : decoder

endmodule
```

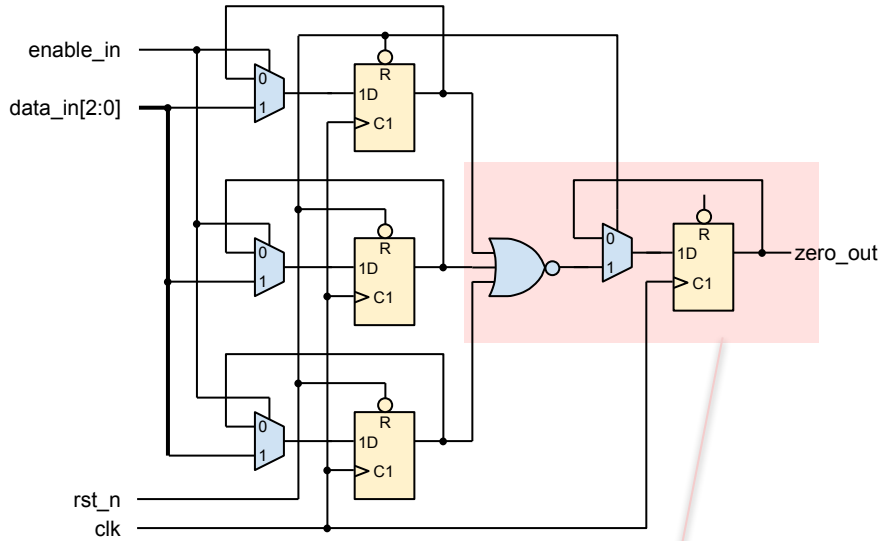




An output decoder should be modeled as a separate combinational process that gets the state variable as an input.

Modeling Output Decoders: Common Error

```
module output_decode1
  (input logic clk, rst_n, enable_in,
   input logic [2:0] data_in,
   output logic      zero_out);
   logic [2:0]        data_r;

   always_ff @ (posedge clk or negedge rst_n)
     begin
      if (rst_n == '0)
        data_r <= '0;
      else
        begin
           if (enable_in == '1)
             data_r <= data_in;
           if (data_r == 3'b000)
             zero_out = '1;
           else
             zero_out = '0;
        end
     end
endmodule
```



If you decode the output value inside a sequential process, an additional flip-flop is synthesized. The output is delayed by one clock cycle. The *extra* flip-flop has no reset.

Tip: Always count your flip-flops!

Summary

- Sequential processes must be modeled using "industry-standard" **code templates** so that all synthesis tools can infer registers in the same way
- Learn the basic templates for different cases (with or without active-high or active-low asynchronous or synchronous reset)
- Start coding by first writing the complete template and then fill in the blanks

```
always_ff @ (posedge clk or negedge rst_n)
  begin
    if (rst_n == '0)
      begin
        // Reset the variables here
      end
    else
      begin
        // Compute next state
        // values here
      end
  end
```

References

1. IEEE Standard for SystemVerilog
   9.2.2.4 Sequential logic always_ff procedure
   10. Assignment statements
2. IEEE Standard for Verilog® Register Transfer Level Synthesis
   5.2 Modeling edge-sensitive sequential logic