Logic Design with Verilog III: Sequential Logic and FSM

Jae W. Lee (jaewlee@snu.ac.kr)

Department of Computer Science and Engineering

Seoul National University

Slide credits: Prof. Ming-Bo Lin (Digital System Designs and Practices Using Verilog HDL and FPGAs)

Objectives

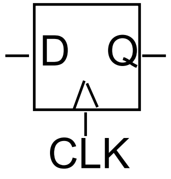
After completing this lecture, you will be able to:

- Describe how to model asynchronous and synchronous *D*-type flip-flops
- Describe how to model registers (data register, register file, and synchronous RAM)
- Describe how to model shift registers
- Describe how to model counters (ripple/synchronous counters and modulo *r* counters)
- Describe how to model sequence generators

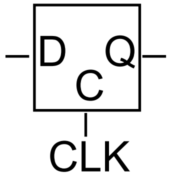
Outline

- Flip-Flops
- Memory elements
- Shift registers
- Counters
- Finite-state machine (FSM)

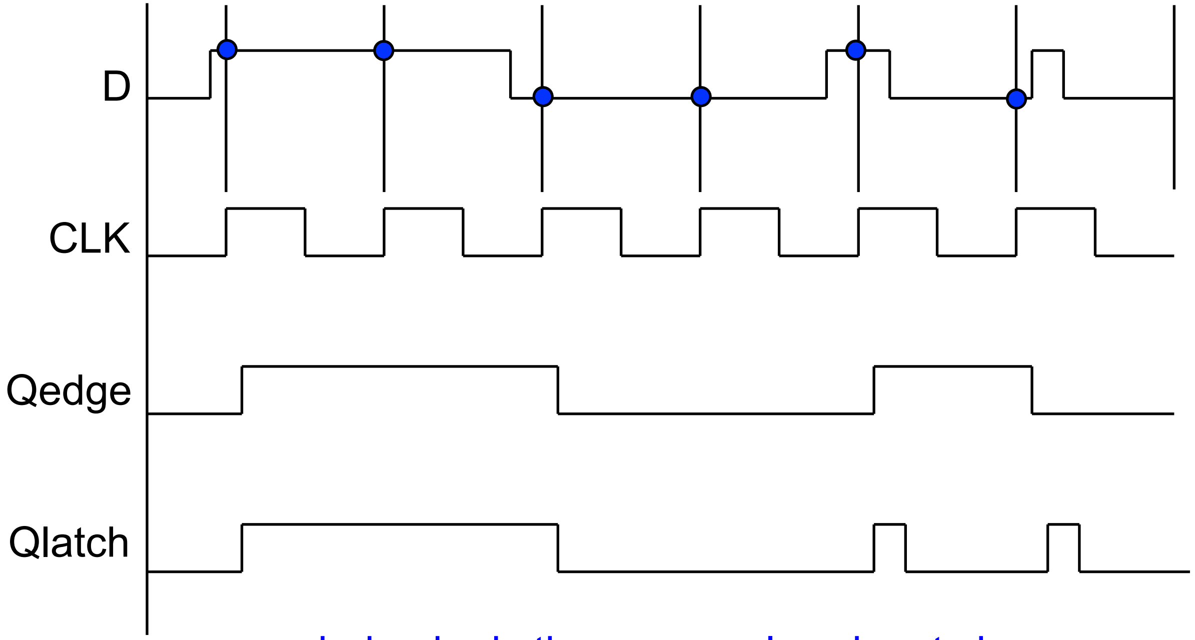
Comparison of Latches and Flip-Flops (FFs)



positive
edge-triggered
flip-flop



transparent
(level-sensitive)
latch

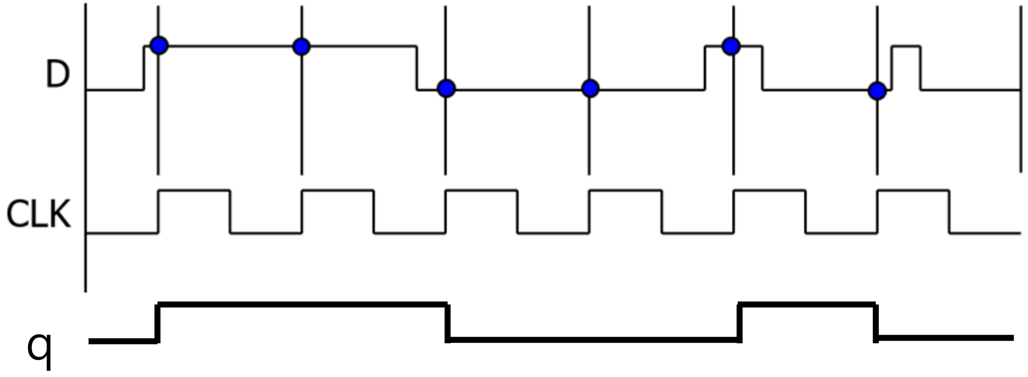


behavior is the same unless input changes
while the clock is high

Again, in FFs, the data value only at the rising edge (or falling edge) is critical (see blue dots). Meanwhile, most latches are sensitive to D value changes as long as the clock is high. Typically, the clock input of a FF is depicted by a triangle.

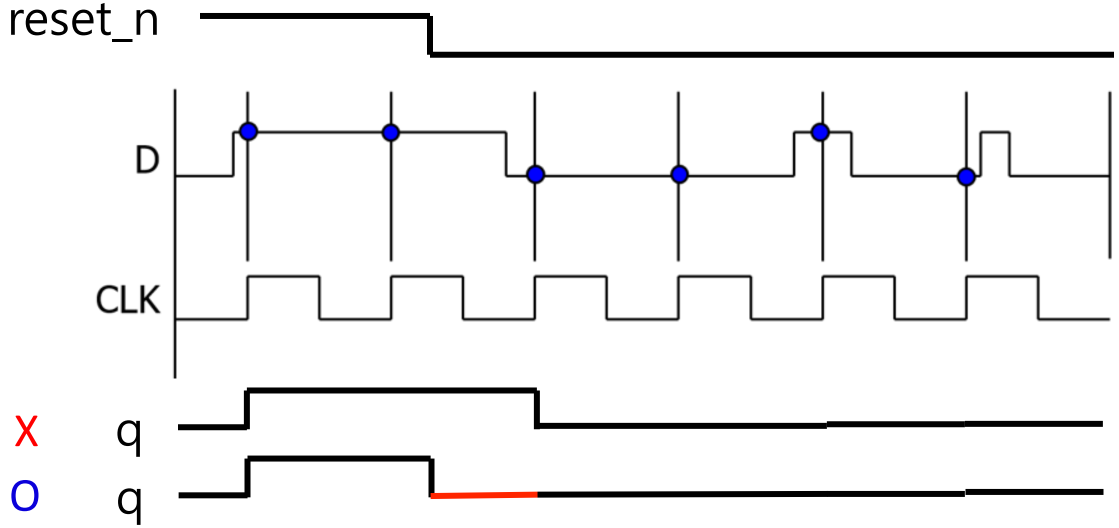
D-Type Flip-Flops

```
// D-type flip-flop
module DFF (clk, d, q);
input        clk, d;
output reg   q;

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



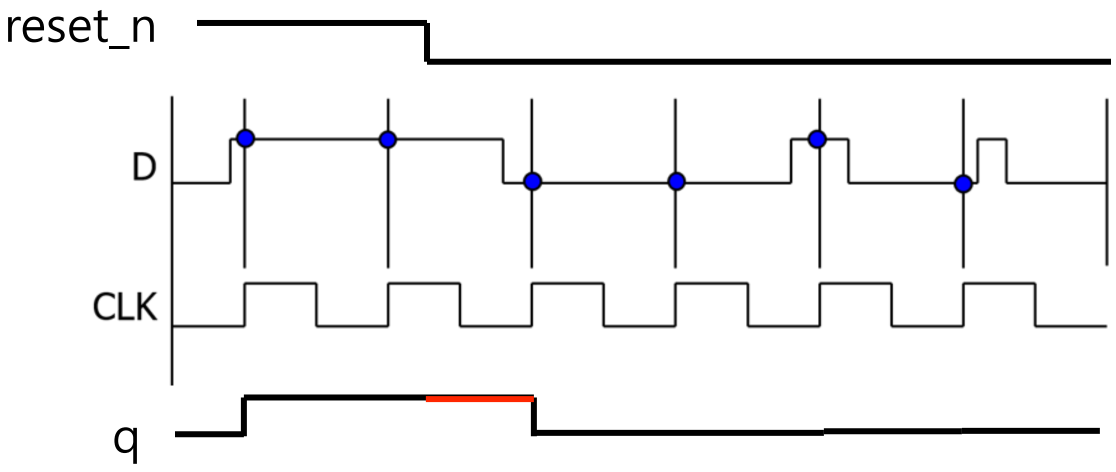
Asynchronous Reset D-Type Flip-Flops

```
// asynchronous reset D-type flip-flop
module DFF_async_reset (clk, reset_n, d, q);
input           clk, reset_n, d;
output reg      q;

always @(posedge clk or negedge reset_n)
    if (!reset_n)   q <= 0;
    else            q <= d;
endmodule
```



Synchronous Reset D-Type Flip-Flops

```
// synchronous reset D-type flip-flop
module DFF_sync_reset (clk, reset_n, d, q);
input           clk, reset_n, d;
output reg      q;

always @(posedge clk)
    if (!reset_n)   q <= 0;
    else            q <= d;
endmodule
```



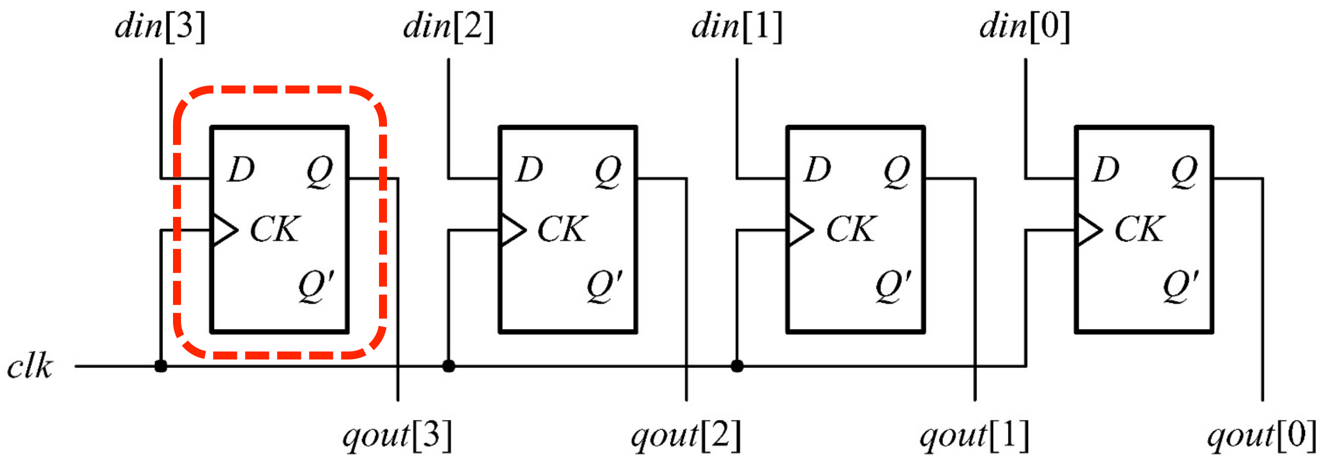
Outline

- Flip-Flops
- Memory elements
- Shift registers
- Counters
- Finite-state machine (FSM)

Memory Elements: Types

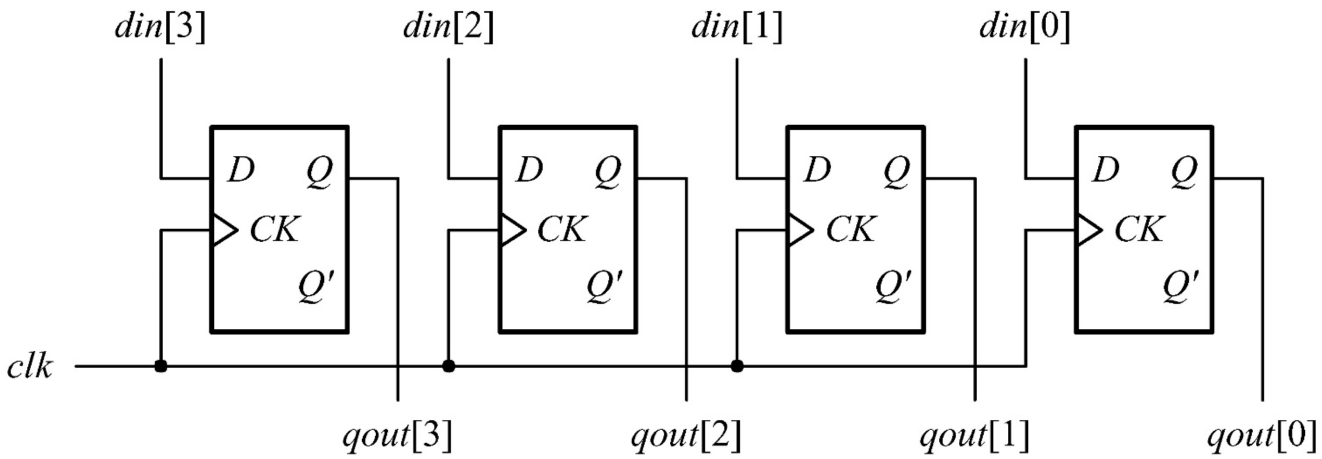
- Data registers
- Register files
- Synchronous RAMs

Memory Elements: Data Registers



```
// D-type flip-flop
module DFF (clk, din, qout);
input        clk, din;
output reg   qout;

       always @(posedge clk)     qout <= din;
endmodule
```

Memory Elements: Data Registers



```
// an n-bit data register
module register(clk, din, qout);
parameter             N = 4;   // number of bits
…
input      [N-1:0]    din;
output reg [N-1:0]    qout;
     always @(posedge clk) qout <= din;
endmodule
```

Memory Elements: Data Registers

```
// an N-bit data register with asynchronous reset
module register_reset (clk, reset_n, din, qout);
parameter                 N = 4;   // number of bits
…
input        [N-1:0]      din;
output reg   [N-1:0]      qout;
always @(posedge clk or negedge reset_n)
   if (!reset_n)    qout <= {N{1'b0}};
   else             qout <= din;
endmodule
```

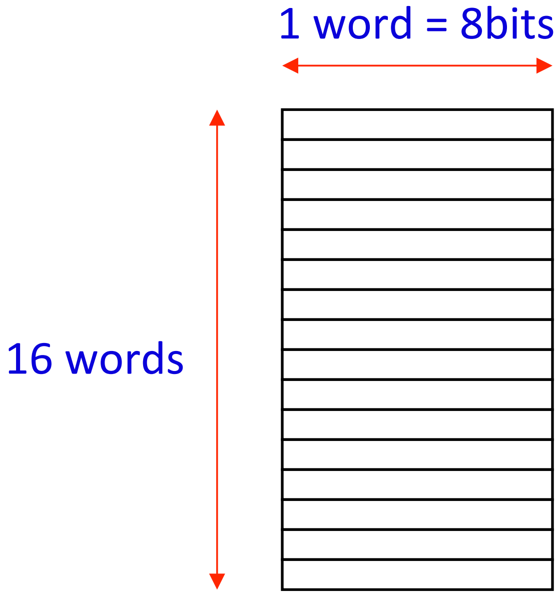
Memory Elements: Data Registers

```
// an N-bit data register with synchronous load and asynchronous reset
module register_load_reset (clk, load, reset_n, din, qout);
parameter                 N = 4;   // number of bits
input                     clk, load, reset_n;
input        [N-1:0]      din;
output reg   [N-1:0]      qout;

always @(posedge clk or negedge reset_n)
   if (!reset_n)    qout <= {N{1'b0}};
   else if (load)   qout <= din;
endmodule
```

Memory Elements: Register File

- Example: A register file having 16 8-bit words

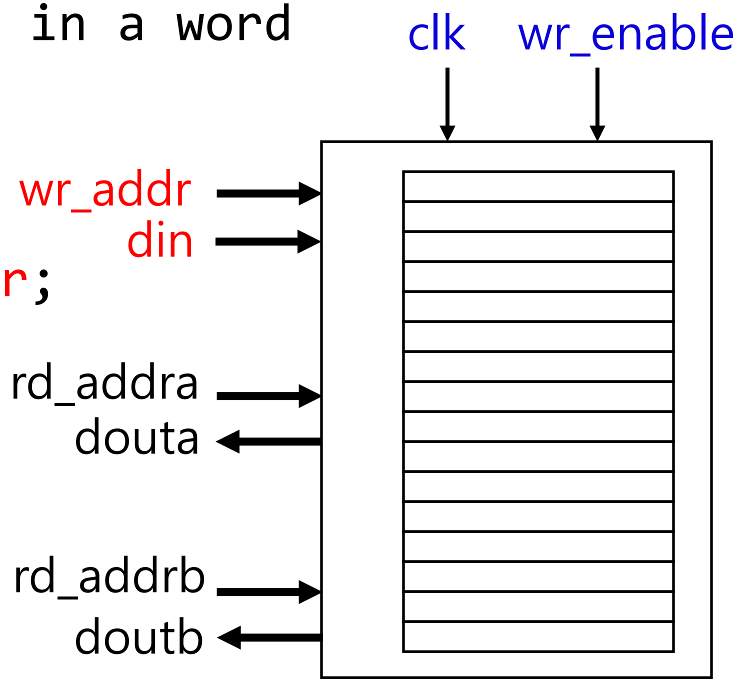
```
// an N-word register file with one-write and two-read ports
parameter                 M = 4;   // number of address bits
parameter                 N = 16;  // number of words, N = 2^M
parameter                 W = 8;   // number of bits in a word
input                     clk, wr_enable;
input        [W-1:0]      din;
output       [W-1:0]      douta, doutb;
input        [M-1:0]      rd_addra, rd_addrb, wr_addr;
reg          [W-1:0]      reg_file [N-1:0];
…
assign douta = reg_file[rd_addra];
assign doutb = reg_file[rd_addrb];
always @(posedge clk)
   if (wr_enable) reg_file[wr_addr] <= din;
```



Memory Elements: Register File

- Example: A register file having 16 8-bit words

```
// an N-word register file with one-write and two-read ports
parameter                 M = 4;   // number of address bits
parameter                 N = 16;  // number of words, N = 2^M
parameter                 W = 8;   // number of bits in a word
input                     clk, wr_enable;
input        [W-1:0]      din;
output       [W-1:0]      douta, doutb;
input        [M-1:0]      rd_addra, rd_addrb, wr_addr;
reg          [W-1:0]      reg_file [N-1:0];
…
assign douta = reg_file[rd_addra];
assign doutb = reg_file[rd_addrb];
always @(posedge clk)
   if (wr_enable) reg_file[wr_addr] <= din;
```



Memory Elements: Register File

- Example: A register file having 16 8-bit words

```
// an N-word register file with one-write and two-read ports with read_en
parameter                 M = 4;   // number of address bits
parameter                 N = 16;  // number of words, N = 2^M
parameter                 W = 8;   // number of bits in a word
input                     clk, wr_enable, read_en_a, read_en_b;
input        [W-1:0]      din;
output       [W-1:0]      douta, doutb;
input        [M-1:0]      rd_addra, rd_addrb, wr_addr;
reg          [W-1:0]      reg_file [N-1:0];
…
assign douta = read_en_a ? _________________________;
assign doutb = read_en_b ? _________________________;
always @(posedge clk)
   if (wr_enable) reg_file[wr_addr] <= din;
```

Memory Elements: Synchronous RAM

- RAM = Random Access Memory

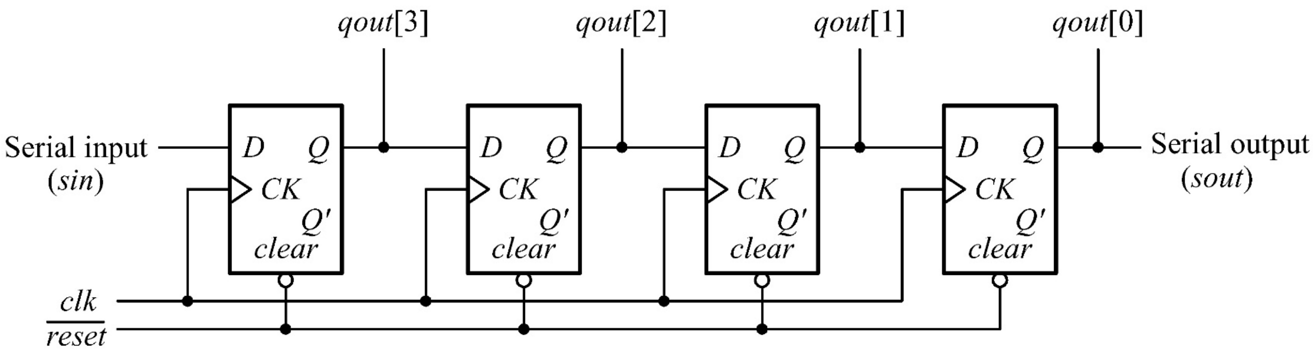
```
// a synchronous RAM module example
parameter                 N = 16;      // number of words
parameter                 A = 4;       // number of address bits
parameter                 W = 8;       // number of wordsize in bits
input        [A-1:0]      addr;
input        [W-1:0]      din;
input        cs, wr, clk;   //chip select, read-write control, and clock signals
output reg   [W-1:0]      dout;
reg          [W-1:0]      ram[N-1:0];  // declare an N * W memory array

always @(posedge clk)
     if (cs) begin
             if (wr)      ram[addr] <= din;
             else         dout <= ram[addr];
     end
```

Outline

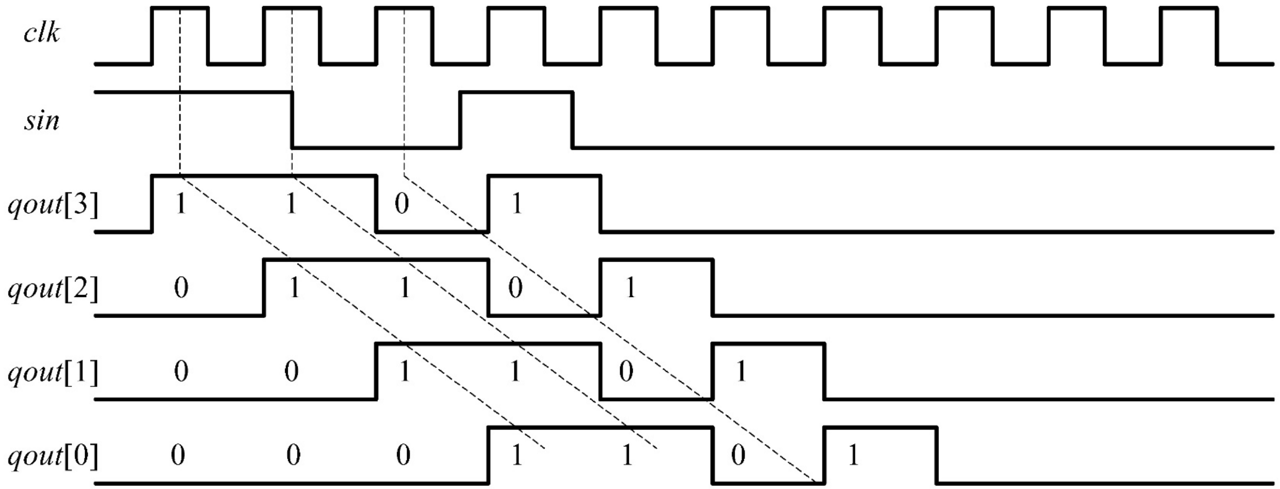
- Flip-Flops
- Memory elements
- Shift registers
- Counters
- Finite-state machine (FSM)

Shift Registers

- Parallel/serial format conversion
  - SISO (serial in serial out)
  - SIPO (serial in parallel out)
  - PISO (parallel in serial out)
  - PIPO (parallel in parallel out)
- Example: SISO shift register



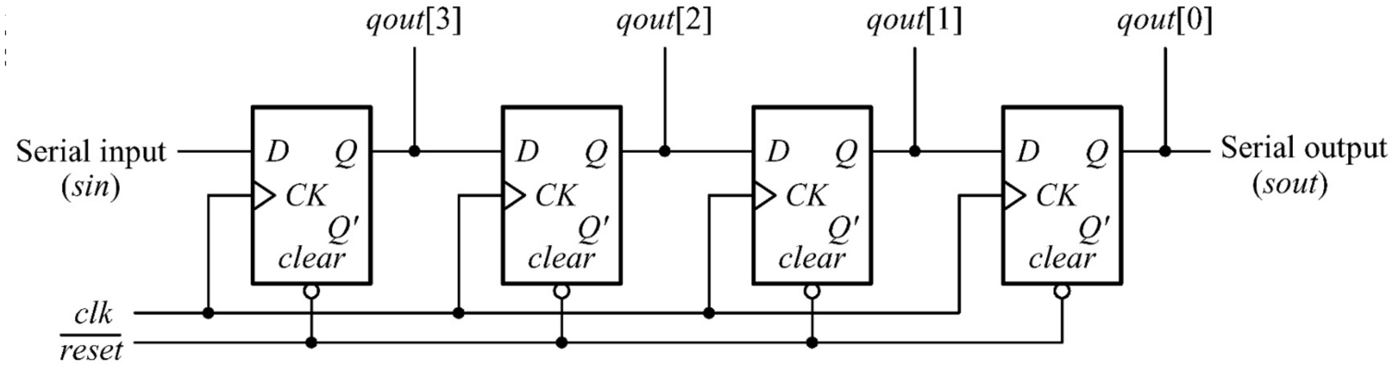
(a) Logic circuit



(b) Timing

Shift Registers: Verilog Code

```
// a shift register module example
module shift_register(clk, reset_n, sin, qout);
parameter                 N = 4;   // number of bits
input                     clk, reset_n, sin;
output reg   [N-1:0]      qout;

always @(posedge clk or negedge reset_n)
   if (!reset_n)    qout <= {N{1'b0}};
   else             qout <= {sin, qout[N-1:1]};
endmodule
```



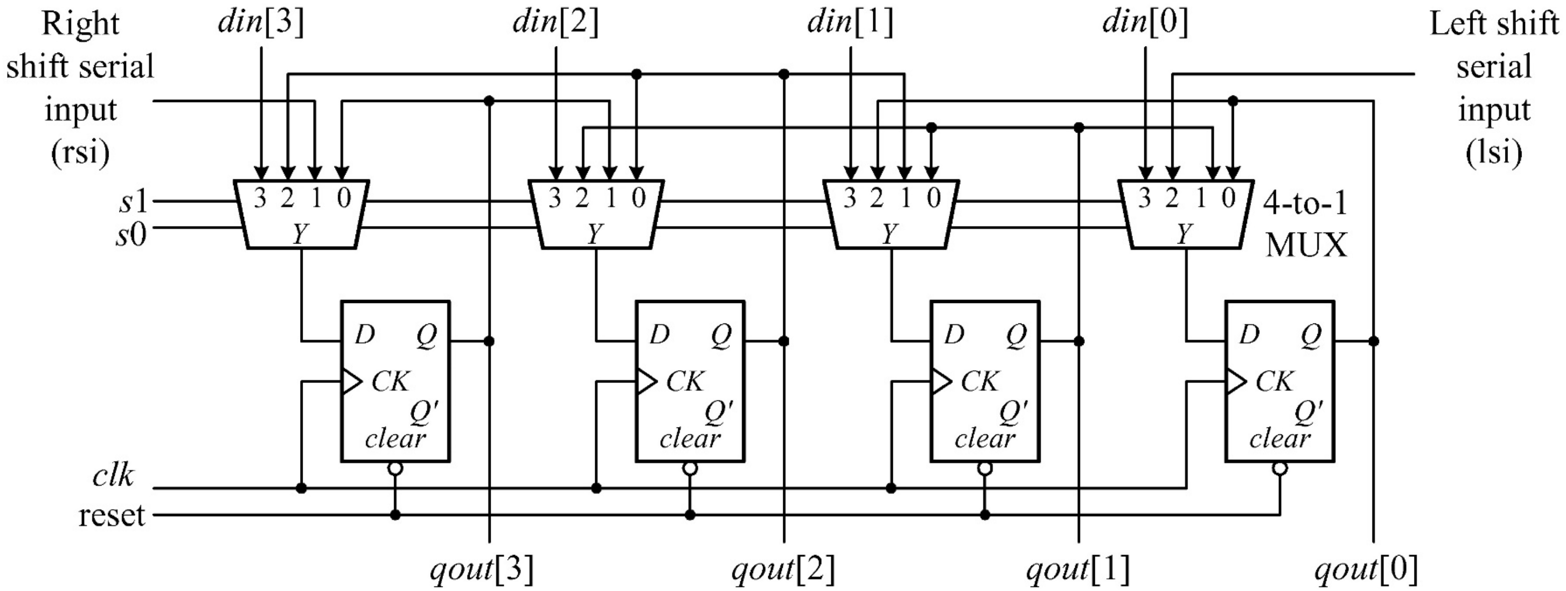
Shift Registers: With Parallell Load

```
// a shift register with parallel load module example
module shift_register_parallel_load (clk, load, reset_n, sin, din, qout);
parameter                 N = 8;  // number of bits
input                     clk, load, reset_n, sin;
input        [N-1:0]      din;
output reg   [N-1:0]      qout;

always @(posedge clk or negedge reset_n)
   if (!reset_n)    qout <= {N{1'b0}};
   else if (load)   qout <= din;
   else             qout <= {sin, qout[N-1:1]};
```

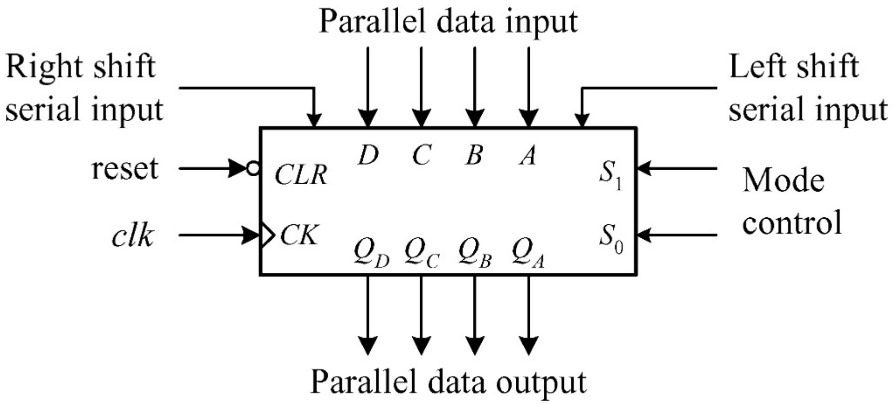
Universal Shift Registers

- A universal shift register can carry out
  - SISO
  - SIPO
  - PISO
  - PIPO
- The register must have the following capabilities
  - Parallel load
  - Serial in and serial out
  - Shift left and shift right

Universal Shift Registers



(a) Logic diagram



(b) Logic symbol

*s*1	*s*0	Function
0	0	No change
0	1	Right shift
1	0	Left shift
1	1	Load data

(c) Function table

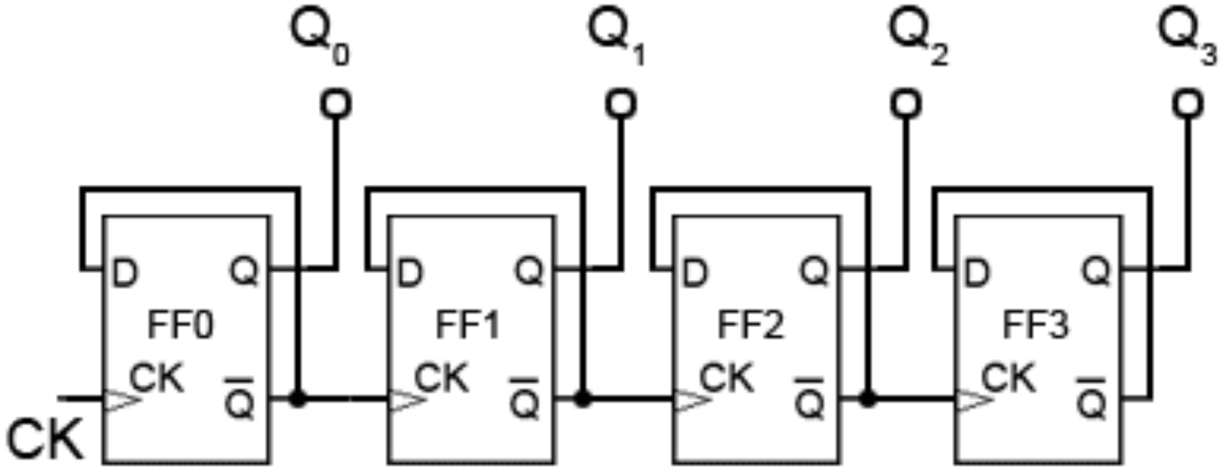
Universal Shift Registers: Verilog Code

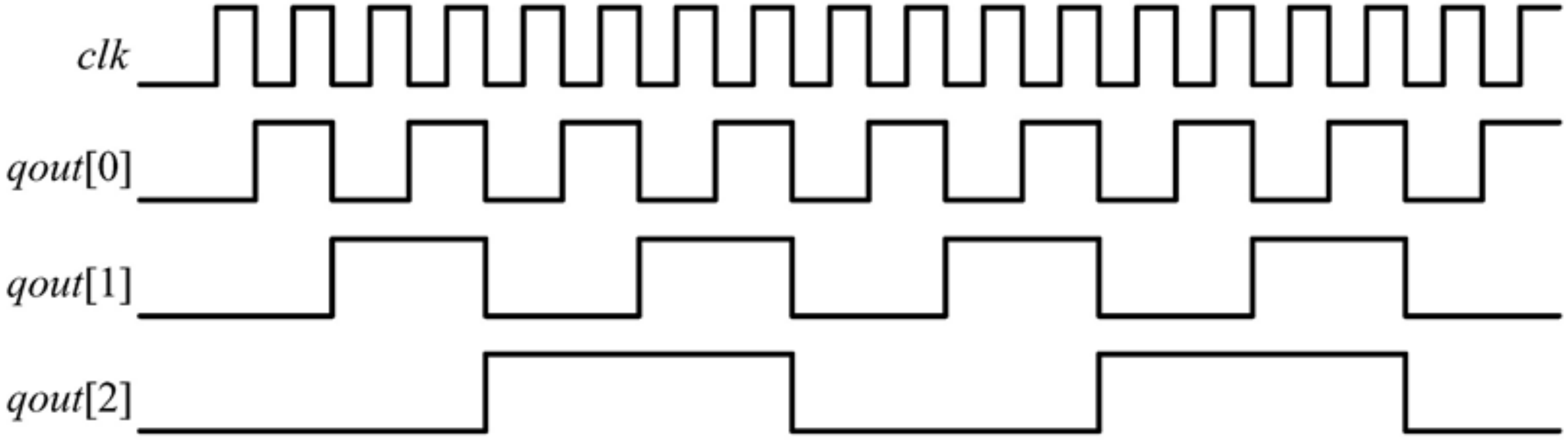
```
// a universal shift register module
module universal_shift_register (clk, reset_n, s1, s0, lsi, rsi, din, qout);
parameter                 N = 4;  // define the default size
input                     clk, reset_n, s1, s0, lsi, rsi;
input        [N-1:0]      din;
output reg   [N-1:0]      qout;
always @(posedge clk or negedge reset_n)
   if (!reset_n)    qout <= {N{1'b0}};
   else case ({s1,s0})
         2'b00: ; // qout <= qout;                   // No change
         2'b01: qout <= {lsi, qout[N-1:1]};          // Shift right
         2'b10: qout <= {qout[N-2:0], rsi};          // Shift left
         2'b11: qout <= din;                         // Parallel load
   endcase
endmodule
```

Outline

- Flip-Flops
- Memory elements
- Shift registers
- Counters
- Finite-state machine (FSM)

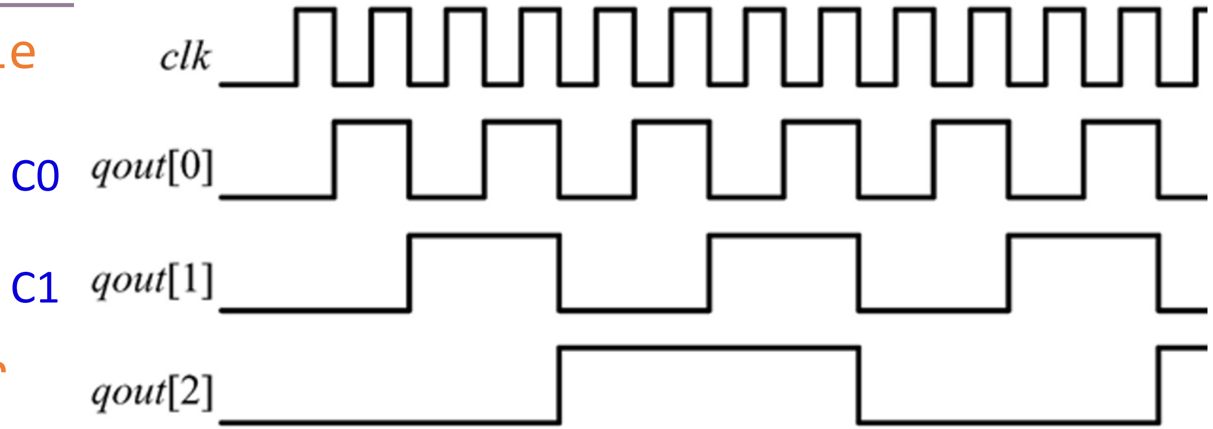
Counters: Binary Ripple Counters

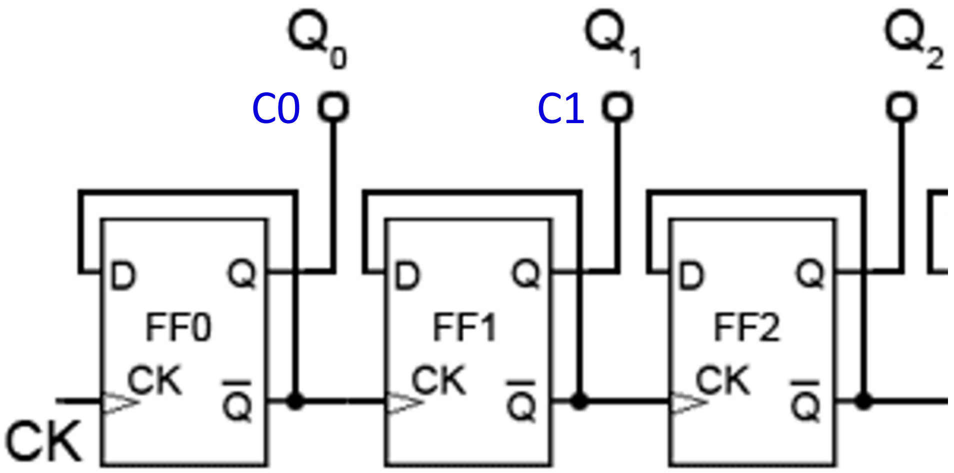




Counters: Binary Ripple Counters

```
// a 3-bit ripple counter module example
module ripple_counter(clk, qout);
input                   clk;
output reg   [2:0]  qout;
wire                    c0, c1;
// the body of the 3-bit ripple counter
assign c0 = qout[0], c1 = qout[1];
always @(negedge clk)
       qout[0] <= ~qout[0];
always @(negedge c0)
       qout[1] <= ~qout[1];
always @(negedge c1)
       qout[2] <= ~qout[2];
```





Counters: Binary Ripple Counters

```
// a 3-bit ripple counter with enable control
module ripple_counter_enable(clk, enable, reset_n, qout);
input               clk, enable, reset_n;
output reg   [2:0]  qout;
wire                c0, c1;
assign c0 = qout[0], c1 = qout[1];
always @(posedge clk or negedge reset_n)
    if (!reset_n)         qout[0] <= 1'b0;
    else if (enable)      qout[0] <= ~qout[0];
always @(posedge c0 or negedge reset_n)
    if (!reset_n)         qout[1] <= 1'b0;
    else if (enable)      qout[1] <= ~qout[1];
always @(posedge c1 or negedge reset_n)
    if (!reset_n)         qout[2] <= 1'b0;
    else if (enable)      qout[2] <= ~qout[2];
```

Counters: Binary Ripple Counters

```
// an N-bit ripple counter using generate blocks
parameter                 N = 4; // define the size of counter
…
output reg   [N-1:0]      qout;
genvar                    i;

generate for (i = 0; i < N; i = i + 1) begin: ripple_counter
   if (i == 0) // specify LSB
       always @(negedge clk or negedge reset_n)
            if (!reset_n) qout[0] <= 1'b0;    else qout[0] <= ~qout[0];
   else          // specify the rest bits
       always @(negedge qout[i-1] or negedge reset_n)
            if (!reset_n) qout[i] <= 1'b0;    else qout[i] <= ~qout[i];
end endgenerate
```

Counters: Binary Up/Down Counters

```
module updn_bincounter (clk, reset, eup, edn, qout, cout, bout);
parameter                 N = 4;
input                     clk, reset, eup, edn;
output reg    [N-1:0]     qout;
output                    cout, bout;

always @(posedge clk)
   if (reset)       qout <= {N{1'b0}};  // synchronous reset
   else if (eup)    qout <= qout + 1;
   else if (edn)    qout <= qout - 1;
assign #1 cout = (&qout)& eup;          // generate carry out
assign #1 bout = (~|qout)& edn;         // generate borrow out
```

Outline

- Flip-Flops
- Memory elements
- Shift registers
- Counters
- Finite-state machine (FSM)
  - Definition
  - Modeling styles
  - FSM examples

FSM: Definition

- A finite-state machine (FSM) is

  $M = (I, O, S, \delta, \lambda)$

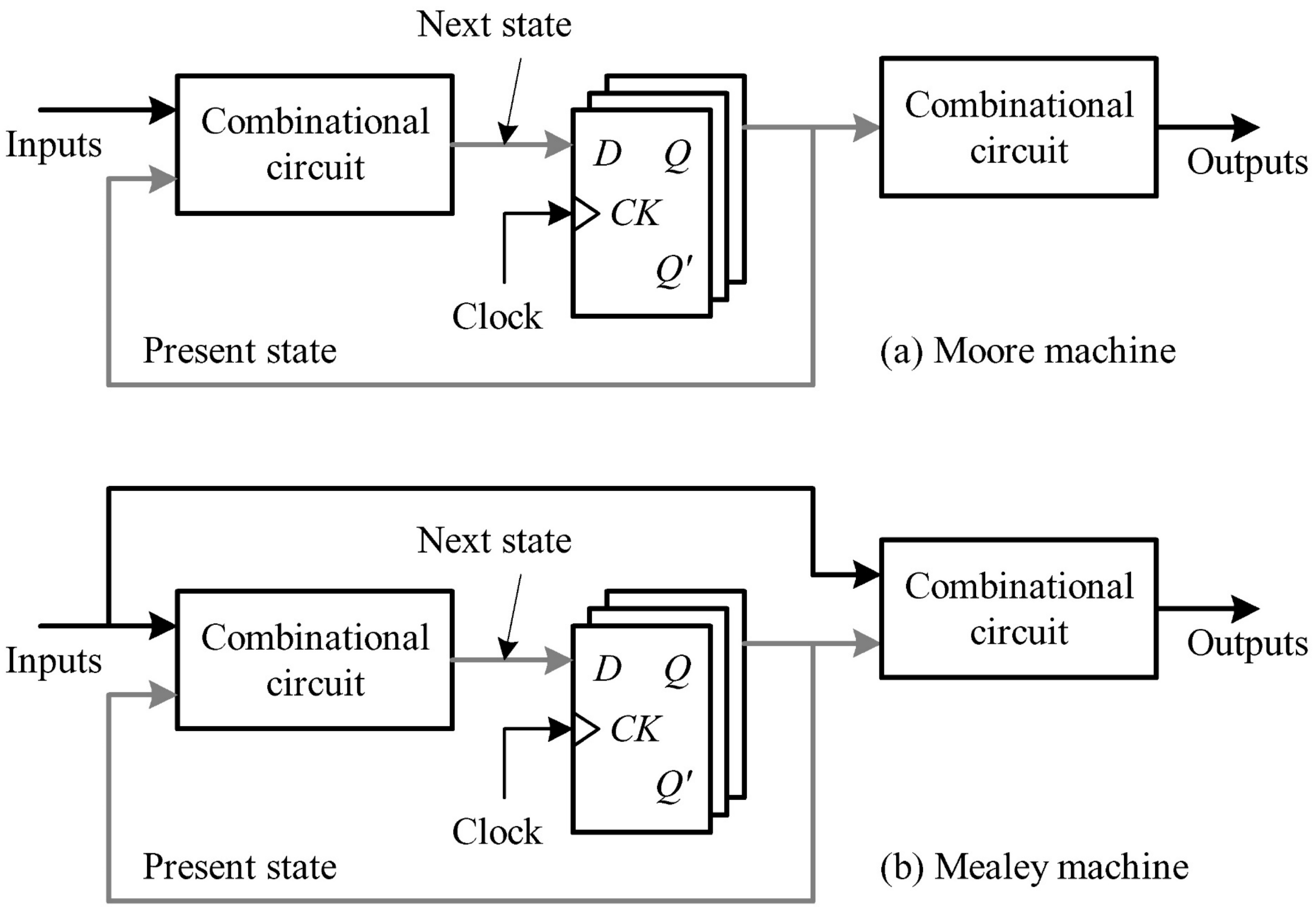
  where *I*, *O*, and *S* are finite, nonempty sets of inputs, outputs, and states

  - State transition function
    $\delta: I \times S \rightarrow S$

  - Output function
    $\lambda : I \times S \rightarrow O$ (Mealy machine)
    $\lambda : S \rightarrow O$ (Moore machine)

FSM: Definition

- Types of FSM: Moore and Mealy



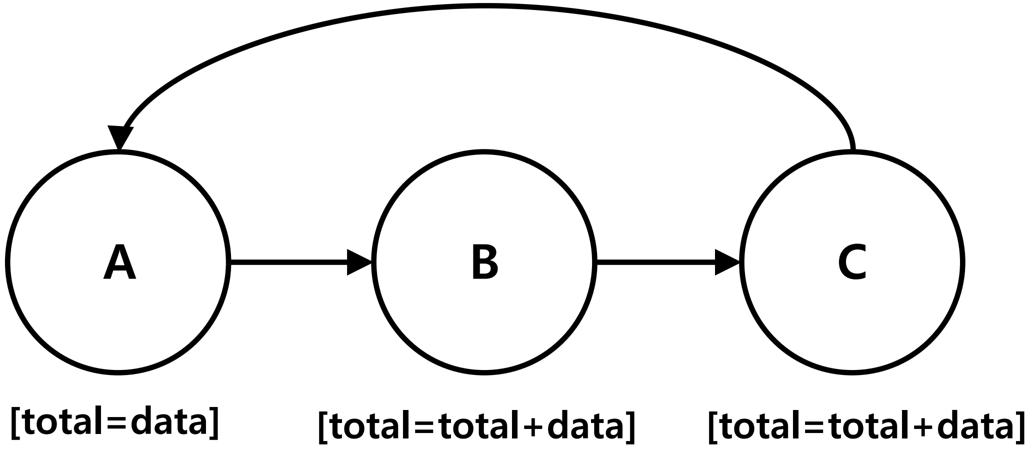
(a) Moore machine

(b) Mealey machine

FSM: Definition

- Example FSM

```
module sum_3data_explicit(clk, data, total);
parameter                    N = 8;
input                        clk, reset;
input          [N-1:0]       data;
output reg     [N-1:0]       total;
reg            [2:0]         state;
parameter                    A = 3'b001,
                             B = 3'b010,
                             C = 3'b100;
always @(posedge clk and posedge reset)
   if (reset)   state <= A;
   else
       case (state)
              A: begin total <= data; state <= B; end
              B: begin total <= total + data; state <= C; end
              C: begin total <= total + data; state <= A; end
       endcase
endmodule
```



data 1 2 3 4 5 6 7 8 …
total 1 3 6 4 9 15 7 15 …

FSM: Modeling Styles – Moore Machine

- **3 parts of an FSM (Moore Machine)**
  - part 1: initialize and update the state register

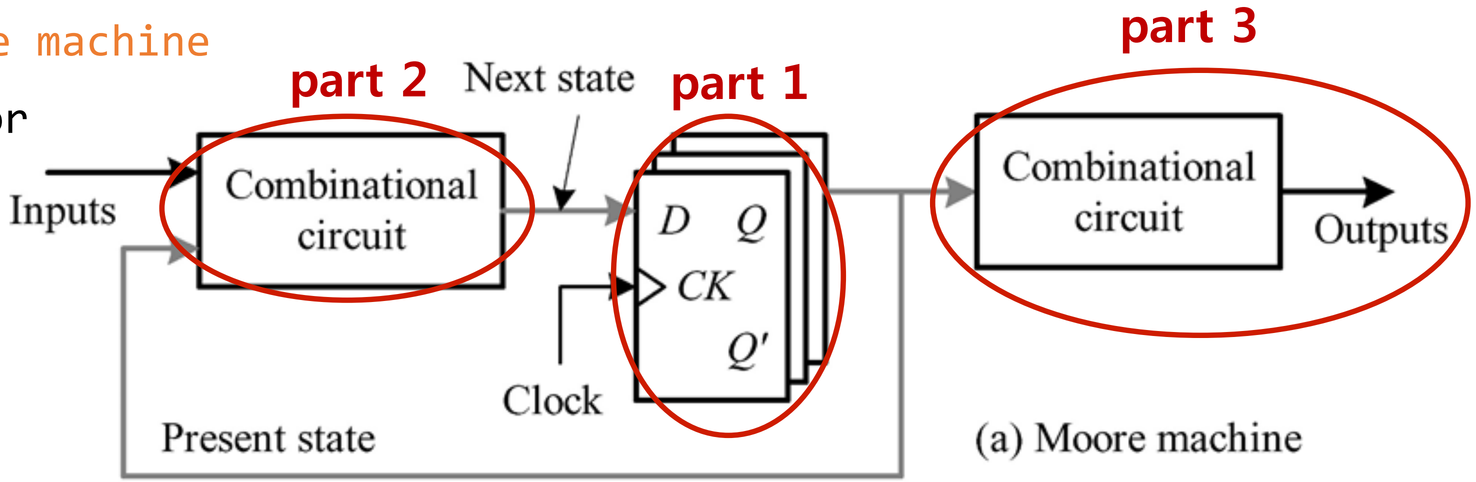
```
always @(posedge clk or negedge reset_n)
       if (!reset_n)   present_state <= A;
       else            present_state <= next_state;
```

  - part 2: determine next state

```
always@(present_state or x)
       case (present_state) …
               next_state <= ; or
assignment statements // always is common
```

  - part 3: determine output

```
always @(present_state) // Moore machine

       case (present_state) … or
assignment statement
```



(a) Moore machine

FSM: Modeling Styles – Mealy Machine

- **3 parts of an FSM (Mealy Machine)**
  - part 1: initialize and update the state register

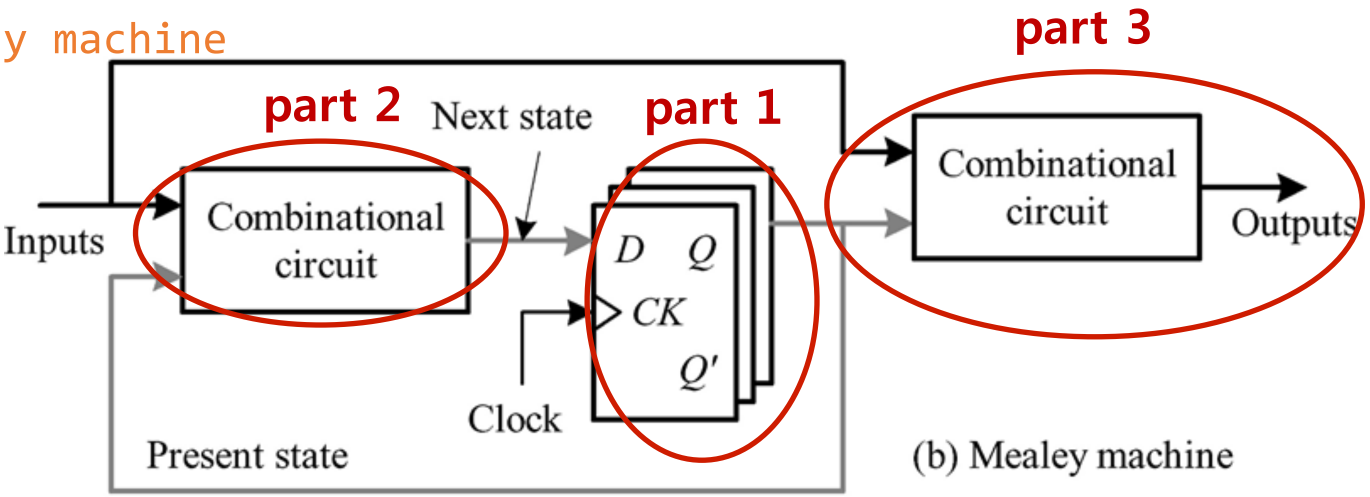
```
always @(posedge clk or negedge reset_n)
       if (!reset_n)   present_state <= A;
       else            present_state <= next_state;
```

  - part 2: determine next state

```
always@(present_state or x)
       case (present_state) …
              next_state <= ; or
assignment statements // always is common
```

  - part 3: determine output

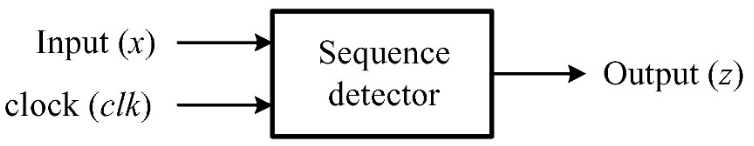
```
always @(present_state or x) // Mealy machine

       case (present_state) … or
assignment statement
```



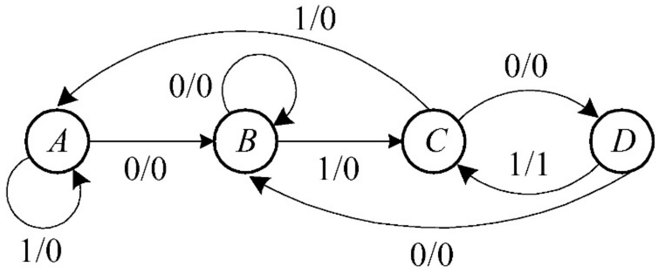
(b) Mealey machine

FSM Example #1: 0101 Sequence Detector

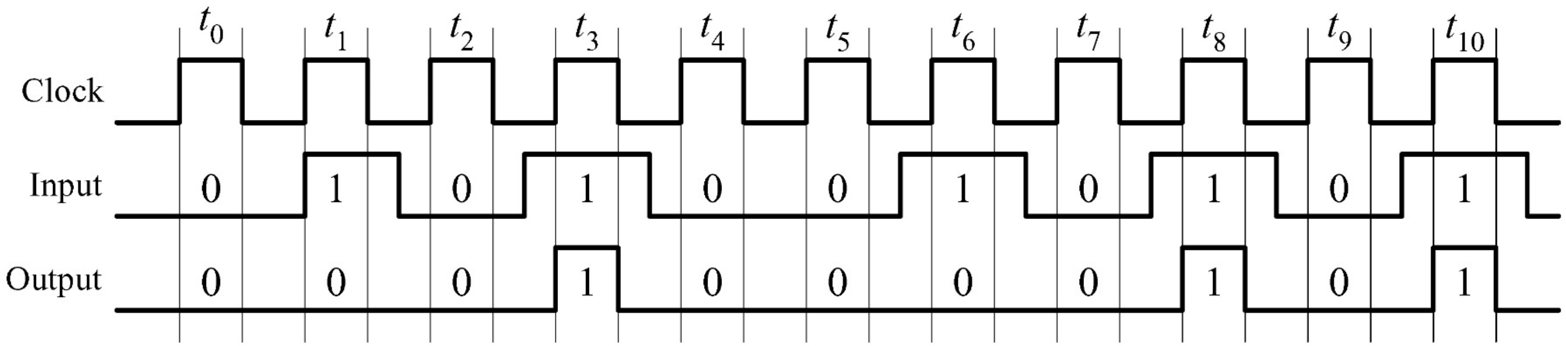
- Design an FSM to detect the pattern 0101 in the input sequence *x*
  - Assume that overlapping pattern is allowed



(a) Block diagram



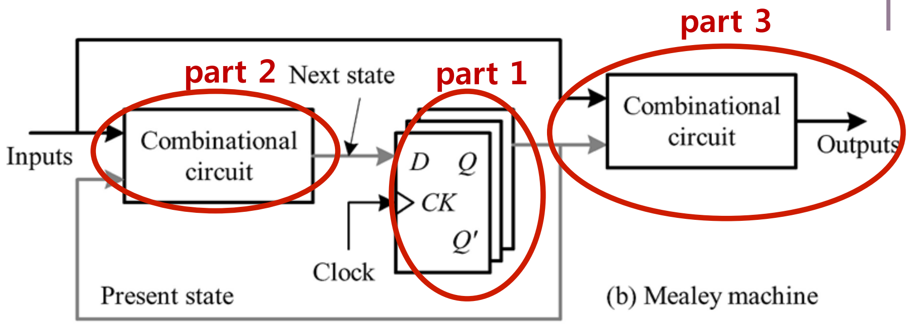
(b) State diagram



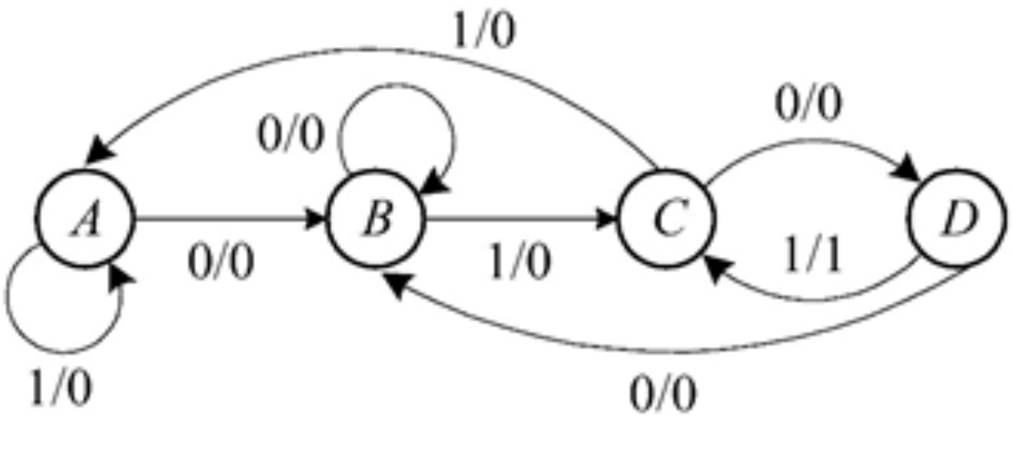
(c) Timing chart

FSM Example #1: 0101 Sequence Detector

```
module sequence_detector_mealy (clk, reset_n, x, z);
   input                       clk, reset_n, x; output reg z;
   reg                 [1:0]    present_state, next_state; // present state and next state
   parameter                   A = 2'b00, B = 2'b01, C = 2'b10, D = 2'b11;
// part 1:  initialize to state A and update present state register
   always @(posedge clk or negedge reset_n)
        if(!reset_n) present_state <= A; else present_state <= next_state;
// part 2: determine next state
   always @(present_state or x)
      case(present_state)
         A: if (x) next_state = A; else next_state = B;
         B: if (x) next_state = C; else next_state = B;
         C: if (x) next_state = A; else next_state = D;
         D: if (x) next_state = C; else next_state = B;
      endcase
// part 3: evaluate output z
   always @(present_state or x) // mealey machine
      case (present_state)
         A: if (x) z = 1'b0; else z = 1'b0;
         B: if (x) z = 1'b0; else z = 1'b0;
         C: if (x) z = 1'b0; else z = 1'b0;
         D: if (x) z = 1'b1; else z = 1'b0;
      endcase
endmodule
```



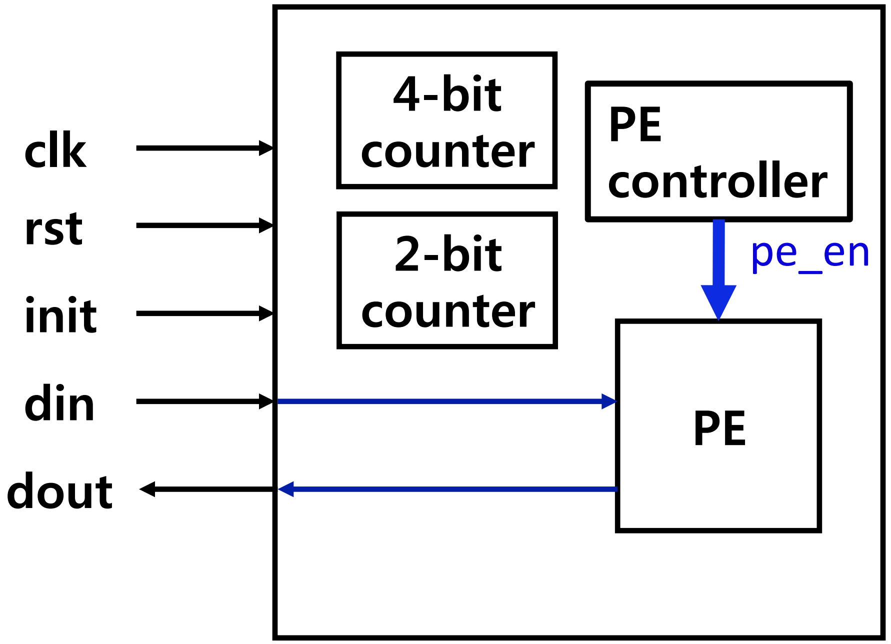
(b) Mealey machine

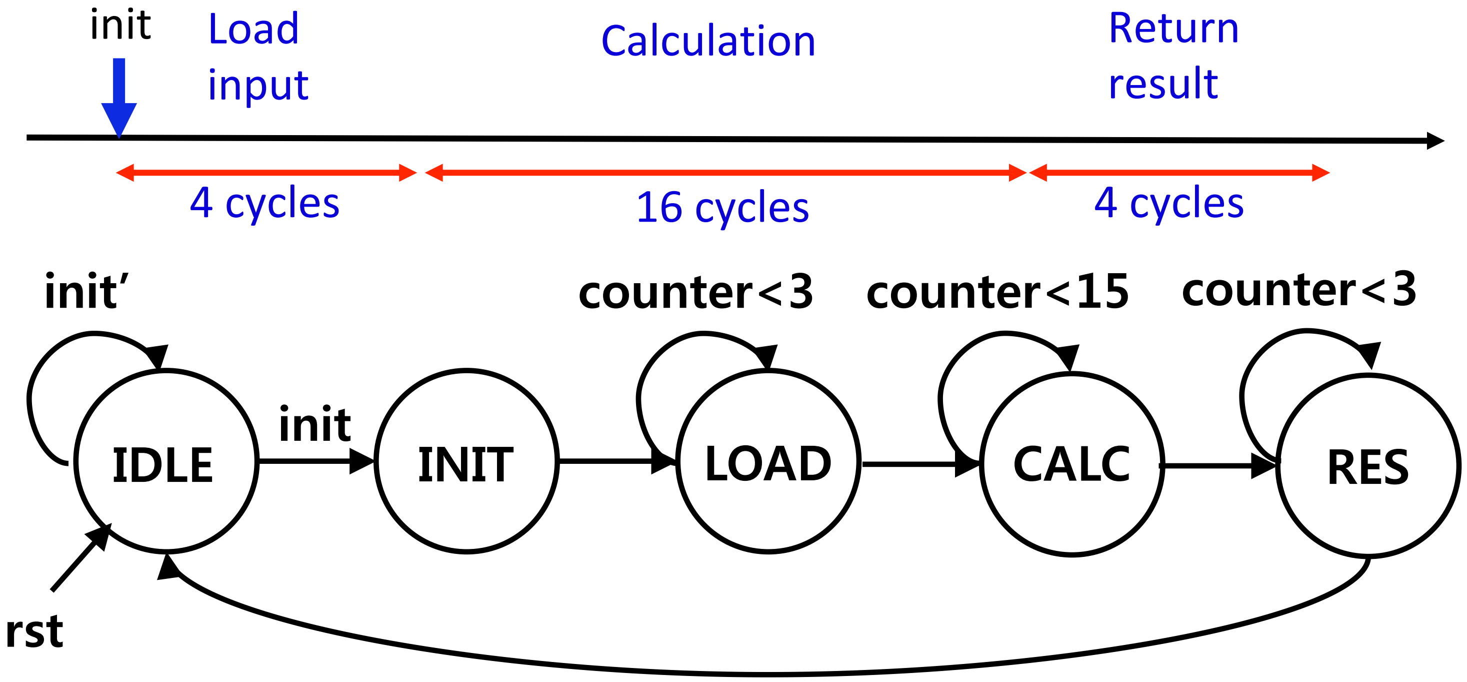


(b) State diagram

FSM Example #2: Simple PE Controller

- Simple PE (Processing Element) controller
  - **IDLE:** PE waits for input data.
  - **INIT:** When ***init*** is changed into 1. PE controller starts to get data from outside (***din***). Internal signal, ‘pe_en’, is changed into 1.
  - **LOAD:** PE controller gets 4 data entries from outside (***din***) and gives it to PE for 4 cycles.
  - **CALC:** PE computes output data for 16 cycles.
  - **RES:** PE controller gets 4 data entries (result) from PE and gives it to outside (***dout***) for 4 cycles. State is turned into IDLE and ‘pe_en’ is changed into 0 when it finishes operation



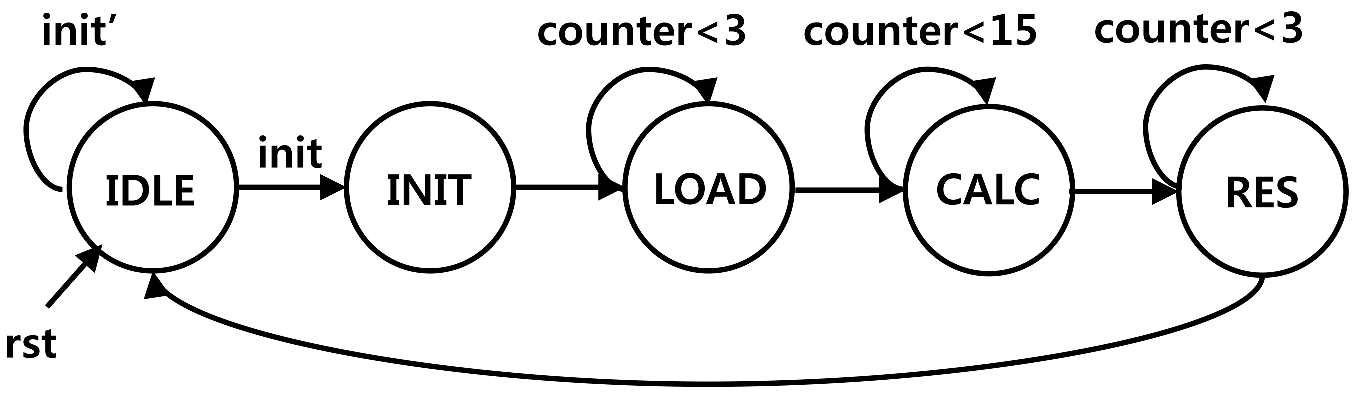


FSM Example #2: Simple PE Controller

```
module pe_controller (clk, rst, init, din, dout);
   input               clk, rst, init;
   input [31:0]                  din;
   output [31:0]                 dout;
   reg                 [1:0]     present_state, next_state; // present state and next state
   reg                 [3:0]     count15; // register for counter
   reg                 [1:0]     count3;
   reg                           counter_rst15;
   reg                           counter_rst3;
   reg                           pe_en; // enable for PE
   parameter                     IDLE = 3'd0, INIT = 3'd1, LOAD = 3'd2, CALC = 3'd3, RES = 3'd4;

// PE
   PE my_pe (
          .clk(clk),
          .rst(rst),
          .en(pe_en),
          .din(din),
          .dout(dout)
   );
// counter
   always @(posedge clk or posedge counter_rst15)
        if(counter_rst15) count15 <= 0;
        else count15 <= count15+1;
   always @(posedge clk or posedge counter_rst3)
        if(counter_rst3) count3 <= 0;
        else count3 <= count3+1;
```

FSM Example #2: Simple PE Controller

```
// part 1:  initialize to state IDLE and update present state register
   always @(posedge clk or posedge rst)
        if(rst) present_state <= IDLE; else present_state <= next_state;
// part 2: determine next state
   always @(*)
      case(present_state)
           IDLE: if(init) next_state = INIT; else next_state = present_state;
           INIT: next_state = LOAD;
           LOAD: if(count3==3) next_state = CALC; else next_state = present_state;
           CALC: if(count15==15) next_state = RES; else next_state = present_state;
           RES:  if(count3==3) next_state = IDLE; else next_state = present_state;
      endcase
// part 3: evaluate output (in this case internal registers)
   always @(*)
      case (present_state)
           CALC: counter_rst15 = 0; // counter for CALC continues to tick
           default: counter_rst15 = 1;
      endcase
   always @(*)
      case (present_state)
           LOAD: counter_rst3 = 0;
           RES: counter_rst3 = 0;
           default: counter_rst3 = 1;
      endcase
   always @(*)
      case (present_state)
           IDLE: pe_en = 0;
           default: pe_en = 1;
      endcase
endmodule
```


init'
counter<3
counter<15
counter<3
init
IDLE
INIT
LOAD
CALC
RES
rst