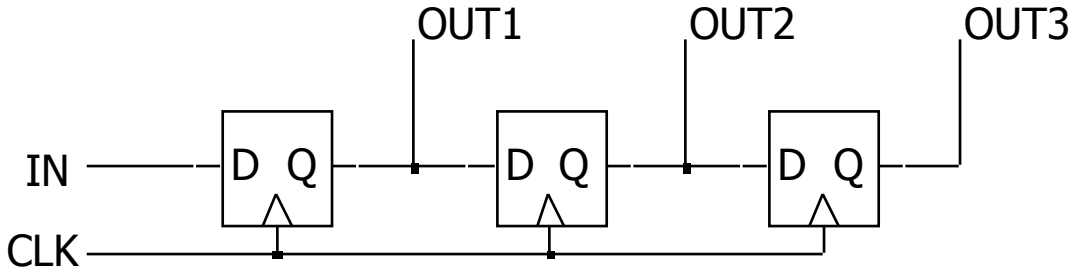
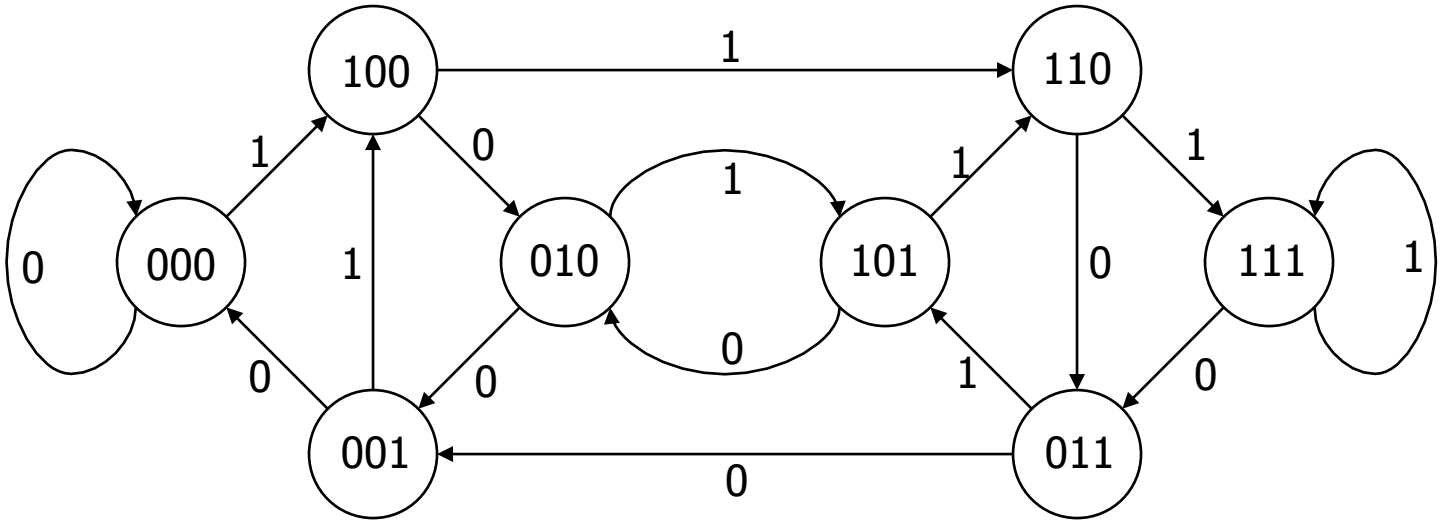
Sequential logic

- State assignment (encoding state) in state machines
  - Sequential
  - One-hot
  - Output-oriented
- Communicating finite-state machines
  - Registers
  - Shift-registers
  - Counters

Can any sequential system be represented with a state diagram?

- Shift register
  - input value shown on transition arcs
  - output values shown within state node



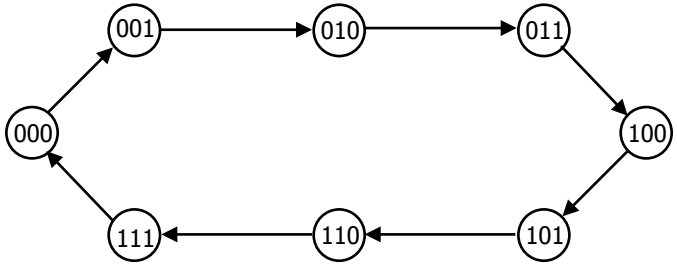


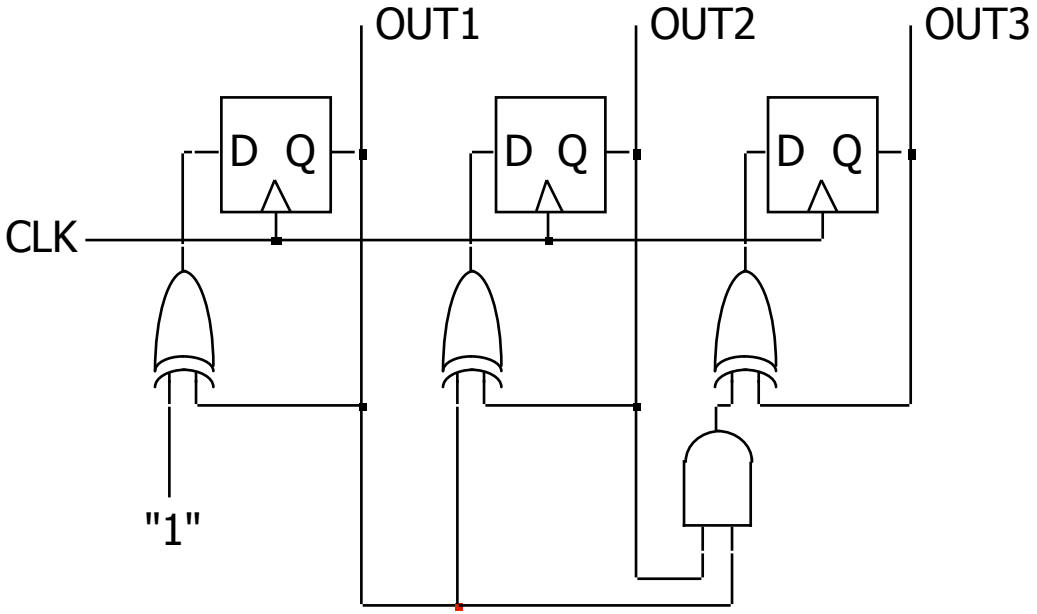
Shift register in Verilog

```
module shift_register (clk, in, out);

  input  clk;
  input  in;
  output [0:3] out;

  reg [0:3] out;

  initial begin
    out = 0;  // out[0:3] = {0, 0, 0, 0};
  end

  always @(posedge clk) begin
    out = {in, out [0:2]};
  end

endmodule
```

Binary counter

- Counter
  - 3 flip-flops to hold state
  - Logic to compute next state
  - Much simpler state diagram -> more complex logic
- Is there a relationship between state diagram and logic needed?





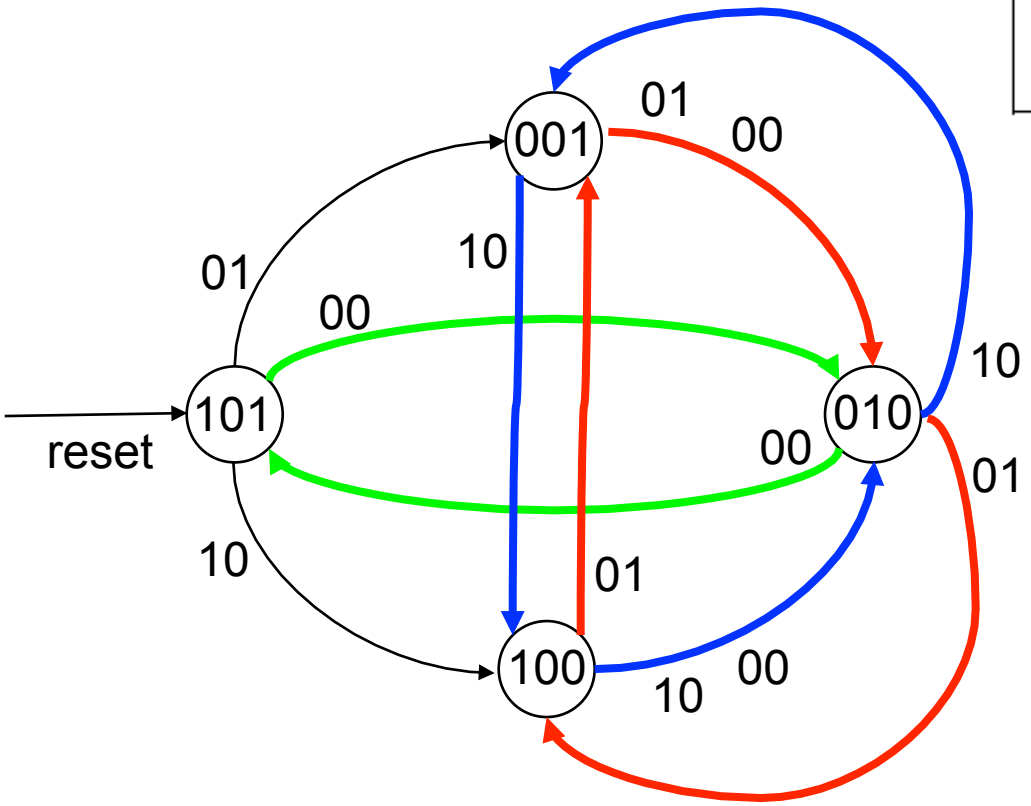
Binary counter in Verilog

```
module binary_counter (clk, c8, c4, c2, c1);
  input     clk;
  output    c8, c4, c2, c1;

   reg [3:0] count;

  initial begin count = 0; end

  always @(posedge clk) begin
     count = count + 4'b0001;
  end

  assign c8 = count[3];
  assign c4 = count[2];
  assign c2 = count[1];
  assign c1 = count[0];

endmodule
```

State machine for Lab 5

SW[1]	SW[0]	Meaning	Pattern (LEDR[2:0])
0	0	Calm	1 0 1 0 1 0
0	1	Right to Left	0 0 1 0 1 0 1 0 0
1	0	Left to Right	1 0 0 0 1 0 0 0 1

- Only 4 states
  - Output = state



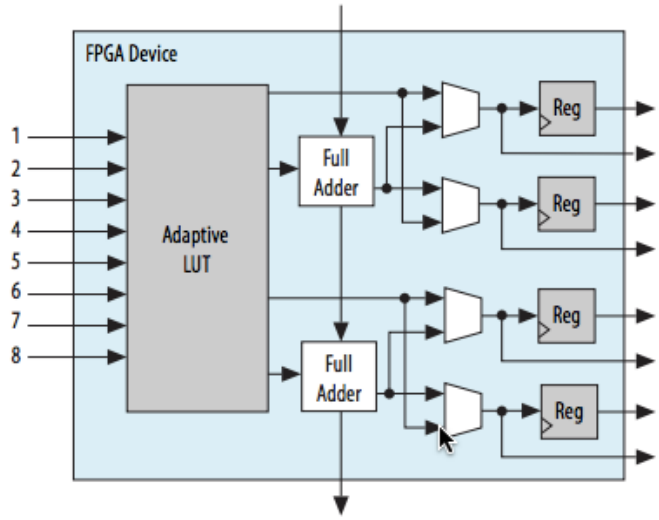
State assignment in Lab 5

- 8 states
- Sequential state assignments
  - Number states 0 to 7 – 3 bits
  - Need logic to translate state number to outputs
- One-hot
  - 8 bits, one is true for each state
  - Need logic to translate state number to outputs
- Output-oriented
  - Exploit outputs as part of state
  - Easy translation to output
  - See previous slide

SW[1]	SW[0]	Meaning	Pattern (LEDR[2:0])
0	0	Calm	1 0 1 0 1 0
0	1	Right to Left	0 0 1 0 1 0 1 0 0
1	0	Left to Right	1 0 0 0 1 0 0 0 1

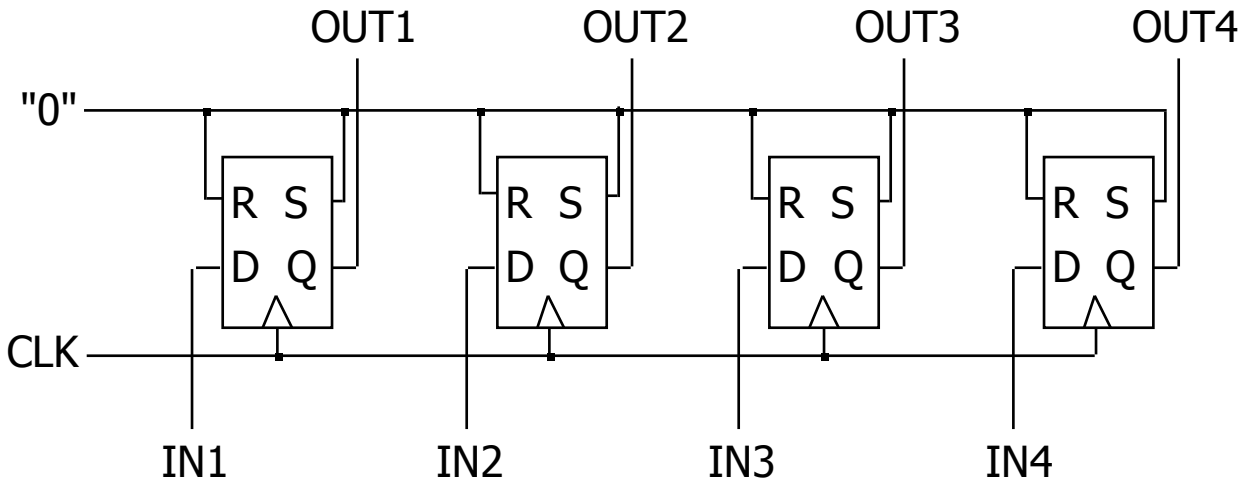
ALM: adaptive logic module

- The following types of functions can be realized in a single ALM:
  - Two independent 4-input functions
  - An independent 5-input function and an independent 3-input function
  - A 5-input function and a 4-input function, if they share one input
  - Two 5-input functions, if they share two inputs
  - An independent 6-input function
  - Two 6-input functions, if they share four inputs and share function
  - Some 7-input functions
- An ALM also has 4 bits of memory
  - We'll discuss later when we talk about sequential logic



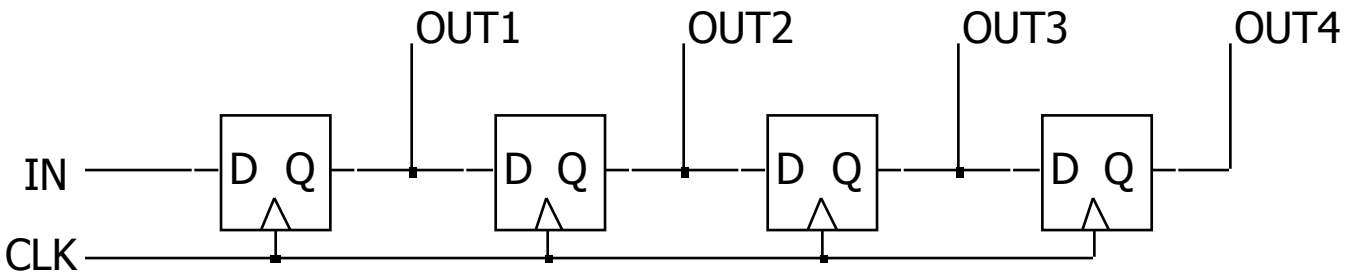
Registers

- Collections of flip-flops with similar controls and logic
  - stored values somehow related (for example, form binary value)
  - share clock, reset, and set lines
  - similar logic at each stage
- Examples
  - shift registers
  - counters



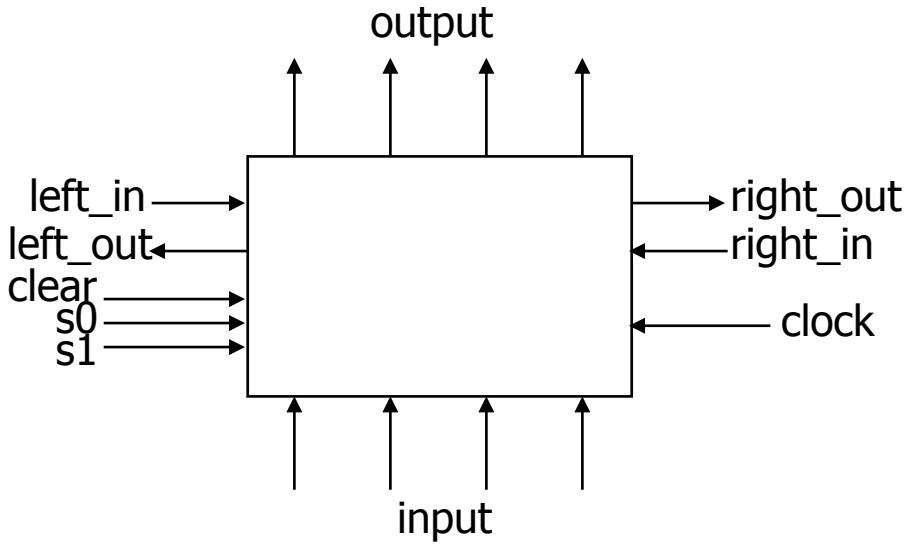
Shift register

- Holds samples of input
  - store last 4 input values in sequence
  - 4-bit shift register:



Universal shift register

- Holds 4 values
  - serial or parallel inputs
  - serial or parallel outputs
  - permits shift left or right
  - shift in new values from left or right



clear sets the register contents and output to 0

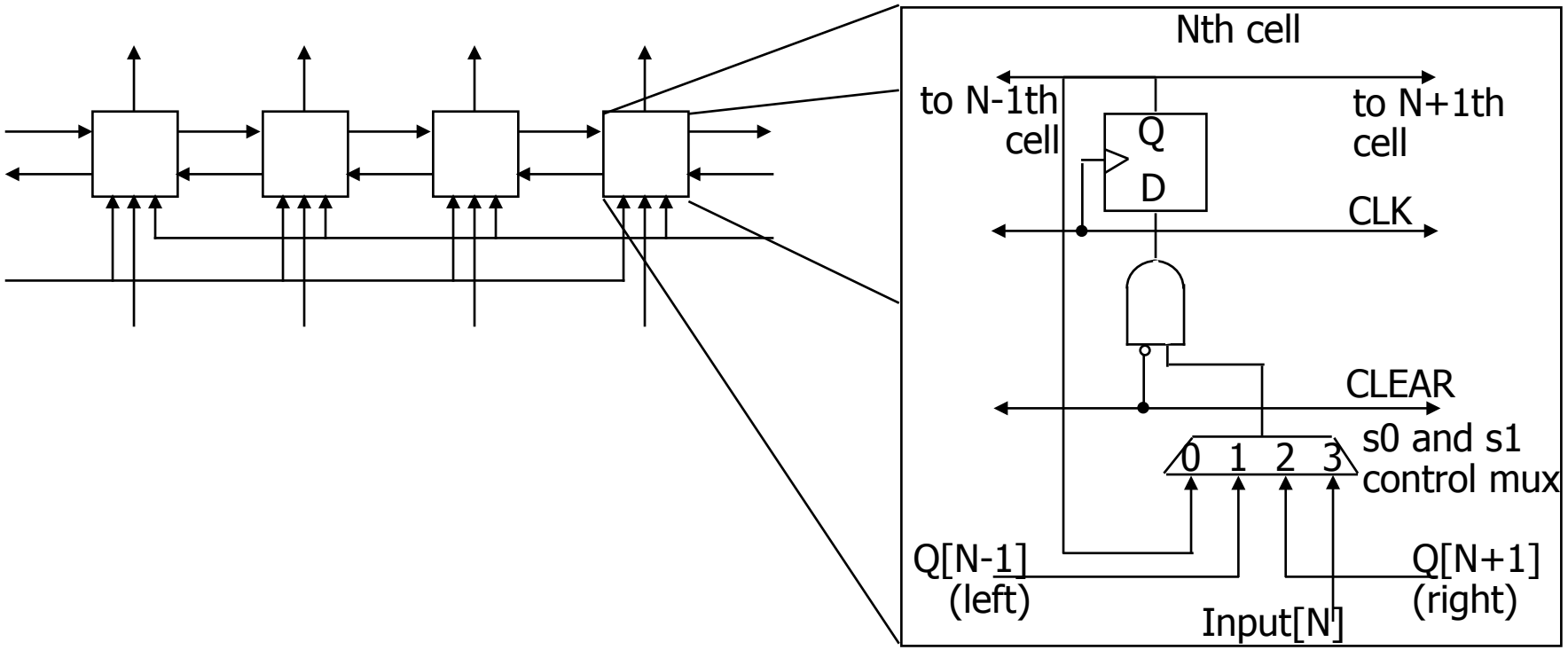
s1 and s0 determine the shift function

s0	s1	function
0	0	hold state
0	1	shift right
1	0	shift left
1	1	load new input

Design of universal shift register

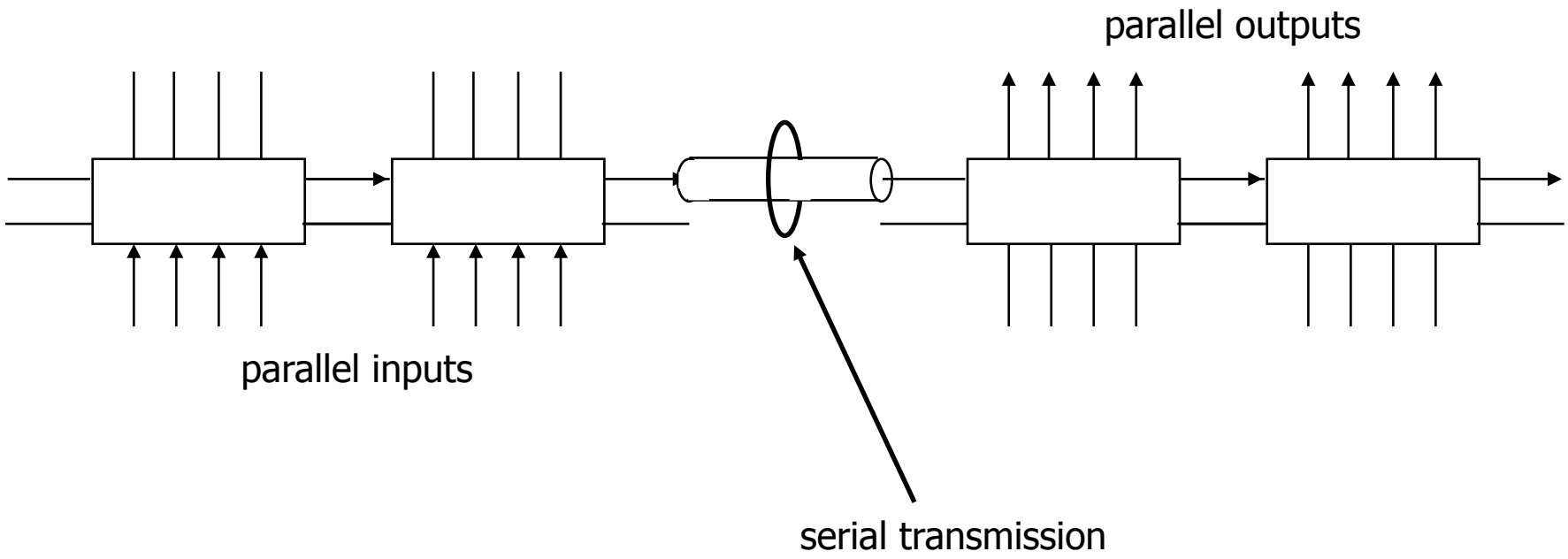
- Consider one of the 4 flip-flops
  - new value at next clock cycle:

clear	s0	s1	new value
1	–	–	0
0	0	0	output
0	0	1	output value of FF to left (shift right)
0	1	0	output value of FF to right (shift left)
0	1	1	input



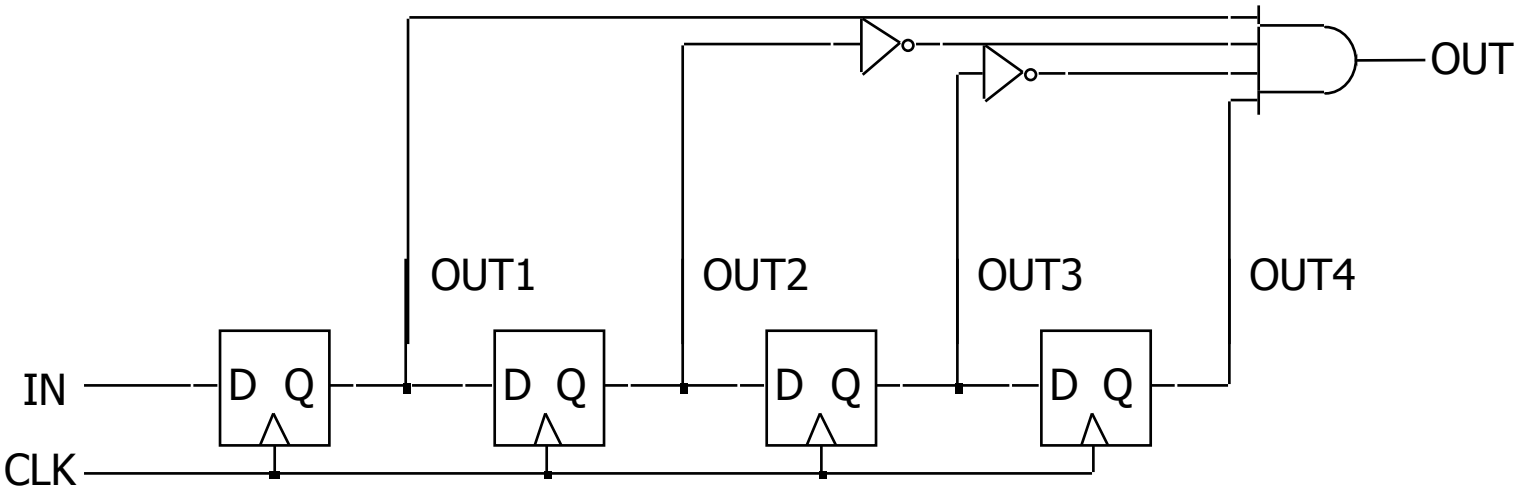
Shift register application

- Parallel-to-serial conversion for serial transmission



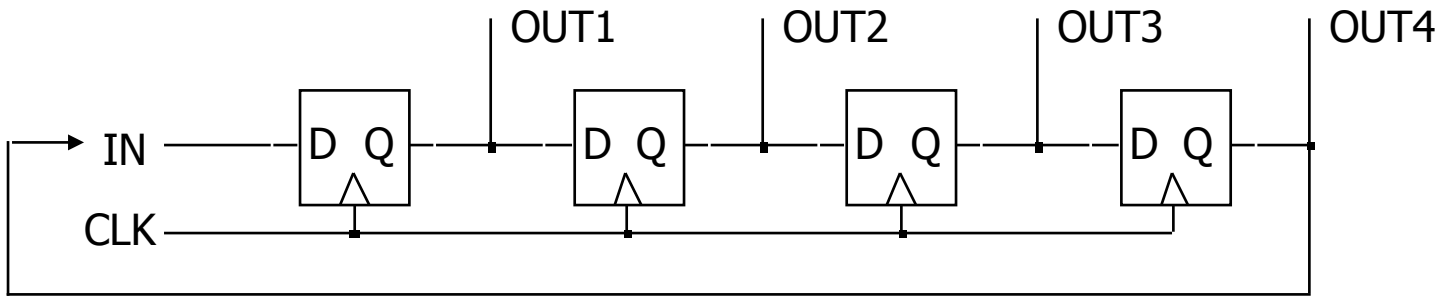
Pattern recognizer

- Combinational function of input samples
  - in this case, recognizing the pattern 1001 on the single input signal



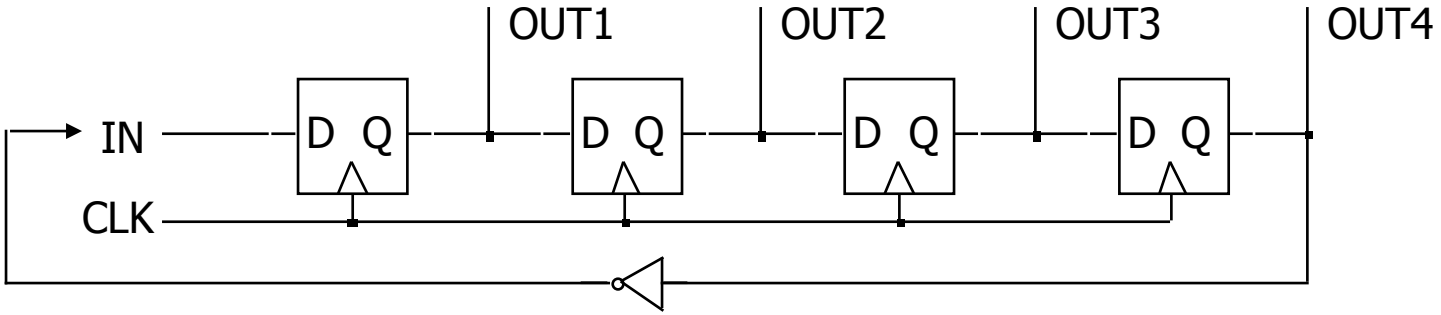
Counters

- Sequences through a fixed set of patterns
  - in this case, 1000, 0100, 0010, 0001
  - if one of the patterns is its initial state (by loading or set/reset)



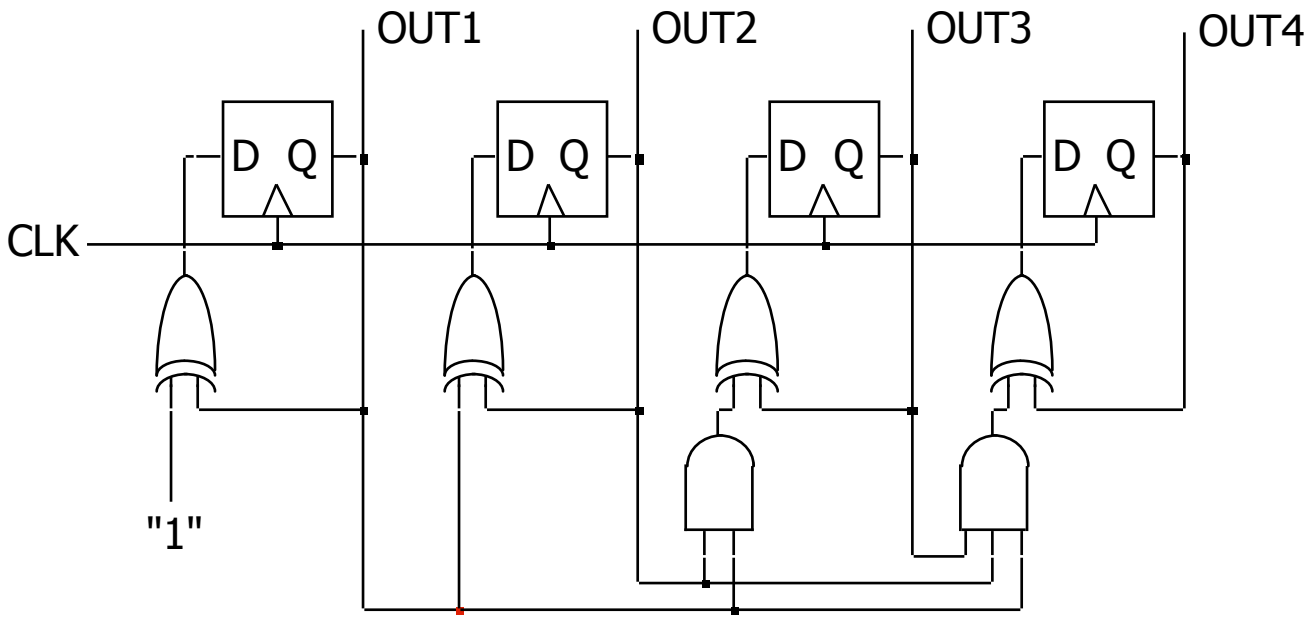
Mobius counter

- How does this counter work (assuming it starts in state 0000)?



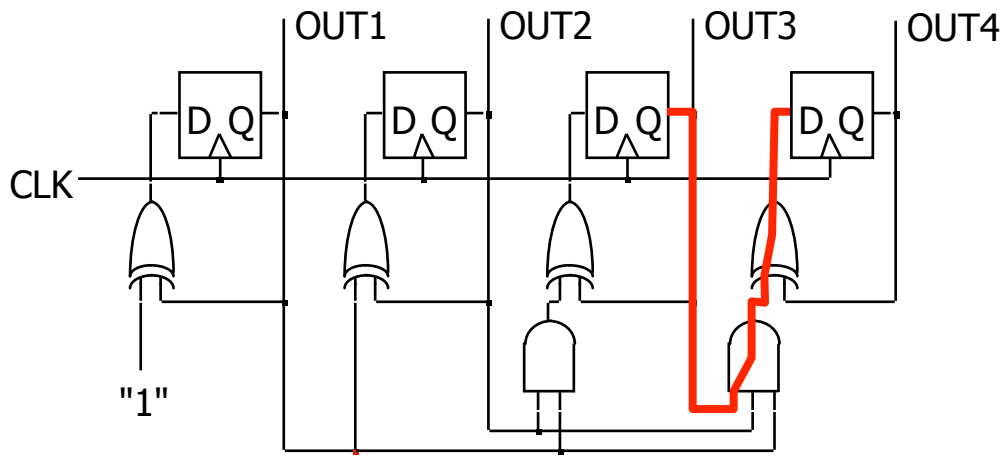
Binary counter

- Logic between registers (not just multiplexer)
  - XOR decides when bit should be toggled
  - always for low-order bit,
    only when first bit is true for second bit,
    and so on



How fast is our counter?

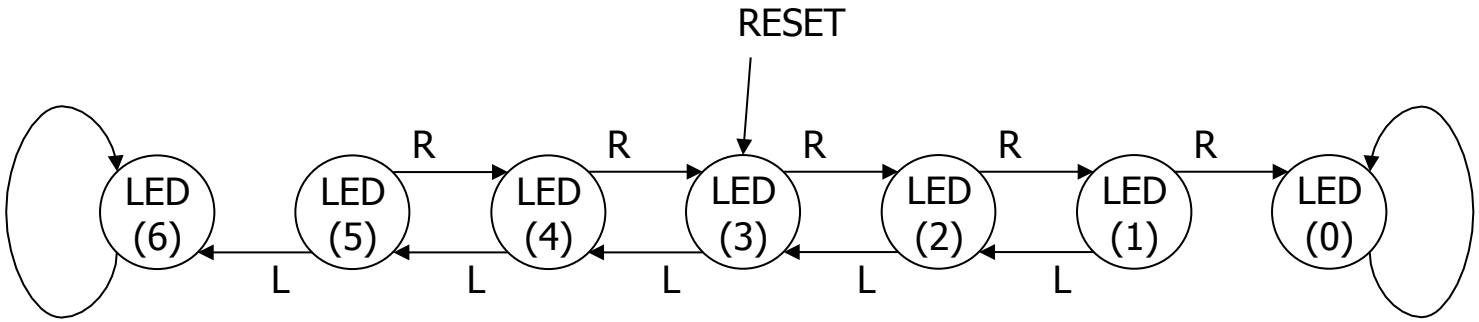
- Period > $T_{propFF} + T_{propCL} + T_{setupFF}$



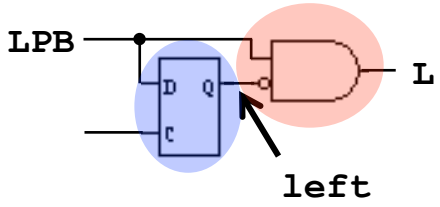
- Period > 3.6ns + max(T_{propCL}) {over all paths from Qs to Ds} + 1.8ns
- Period > 5.4ns + $T_{propXOR} + T_{propAND3}$
- Frequency < 1/Period

Tug-of-War Game FSM (NOT the same as Lab #6)

- Tug of War game
  - 7 LEDS, 2 push buttons (LPB, RPB)



Light Game FSM Verilog



positive edge detector

```
module Light_Game (LEDS, LPB, RPB, CLK, RESET);

    input LPB ;
    input RPB ;
    input CLK ;
    input RESET;
    output [6:0] LEDS ;

    reg [6:0] position;
    reg left;
    reg right;
```

combinational logic and wires

```
    wire L, R;
    assign L = ~left && LPB;
    assign R = ~right && RPB;
    assign LEDS = position;
```

sequential logic

```
    always @(posedge CLK)
        begin
            left <= LPB;
            right <= RPB;
            if (RESET) position <= 7'b0001000;
            else if ((position == 7'b0000001) || (position == 7'b1000000)) ;
            else if (L) position <= position << 1;
            else if (R) position <= position >> 1;
        end

endmodule
```

Do you see a problem with this game?

Activity

- Where is the problem? What is the fix?

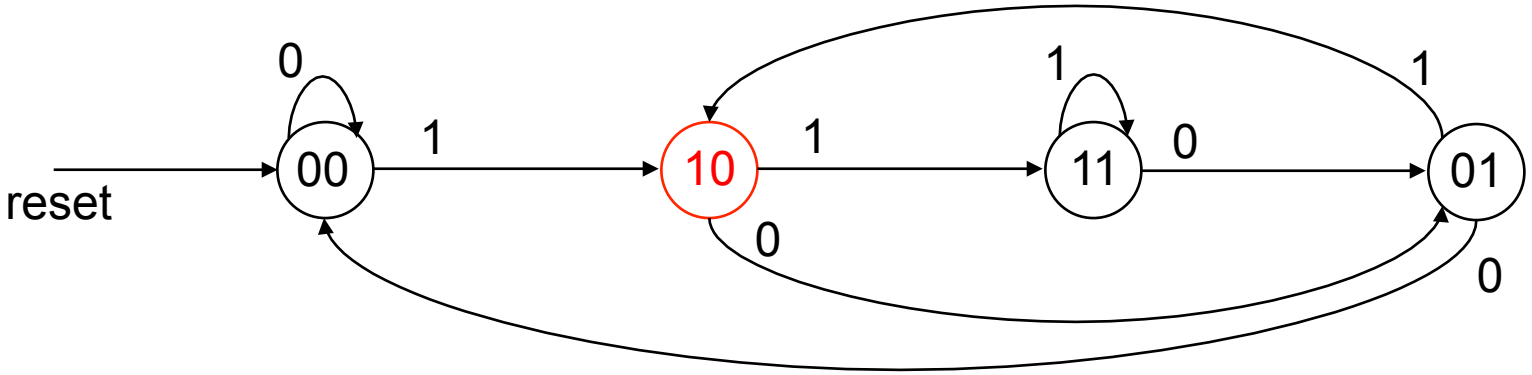
```
always @(posedge CLK) begin
   left <= LPB;
   right <= RPB;
   if (RESET) position <= 7'b0001000;
   else if ( (position == 7'b0000001) || (position == 7'b1000000) )
               position <= position;
   else if (L) position <= position << 1;
   else if (R) position <= position >> 1;
end
```

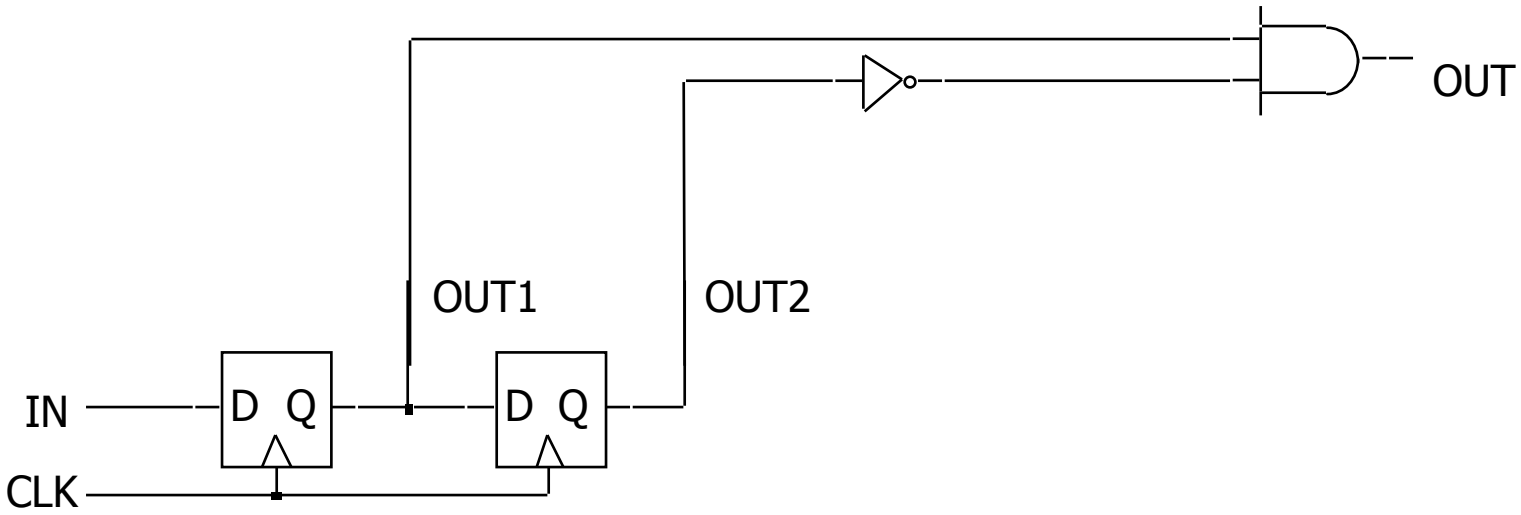
```
always @(posedge CLK) begin  // no longer biased in favor of L player
   left <= LPB;
   right <= RPB;
   if (RESET) position <= 7'b0001000;
   else if ( (position == 7'b0000001) || (position == 7'b1000000) )
                     position <= position;
   else if (L & ~R) position <= position << 1; // correct error in state diag.
   else if (R & ~L) position <= position >> 1; // favoring L player
   else             position <= position;      // otherwise, just hold
```

Lab #6

- 2 user-input FSMs for push buttons
  - One cycle pulse every time a button is pressed
- Tug-of-war game with a separate FSM for each LED (9 of them)
  - 1 center-light (lights up on reset)
  - 8 normal-lights (goes dark on reset)
- Victory logic
  - Input from first and last FSM and two buttons
  - If left-most light is on and left button pressed, then left wins
  - Similarly for right
  - Control HEX0 to display winner: 0 or 1, L or R, etc.
- Consider how a game ends
  - Do you want both push buttons still working after a player wins?

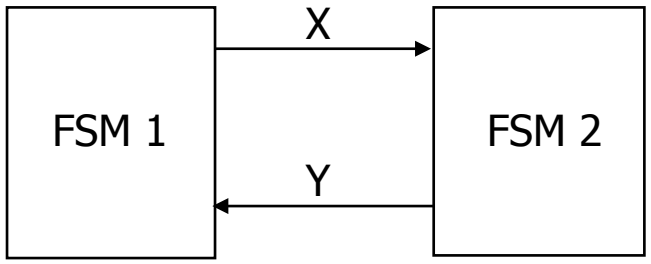
Input FSM – simple shift register

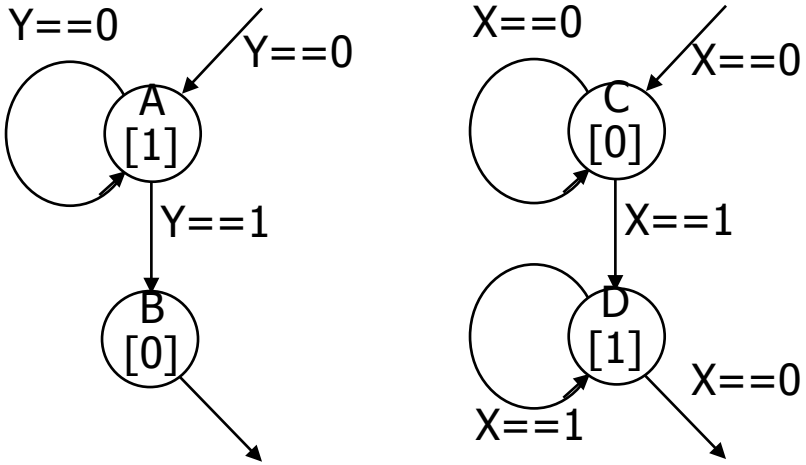


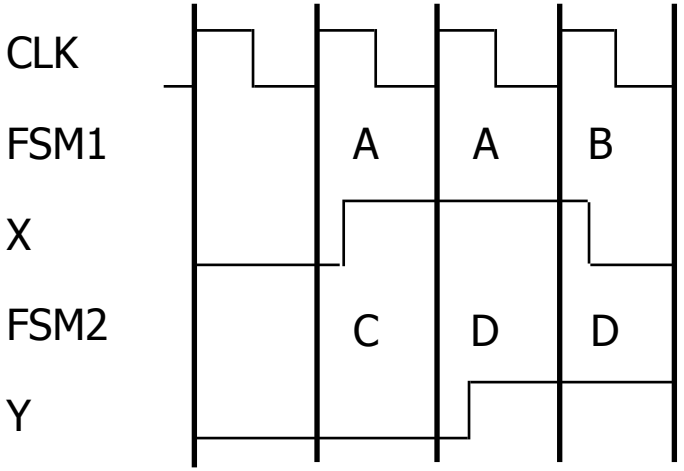


Communicating finite state machines

- One machine's output is another machine's input







machines advance in lock step
initial inputs/outputs: X = 0, Y = 0