# EECE 2322: Fundamentals of Digital Design and Computer Organization

## Lecture 10_2: Finite State Machine

Xiaolin Xu
Department of ECE
Northeastern University

# State of Digital System

- Outputs of sequential logic depend on current *and* prior input values – it has ***memory***.
- Some definitions:
  - **State:** all the information about a circuit necessary to explain its future behavior
  - **Latches and flip-flops:** state elements that store one bit of state
  - **Synchronous sequential circuits:** combinational logic followed by a bank of flip-flops

# State Machine Concept

- Your understanding?

# State Machine Concept

- From Wiki:
- A finite-state machine (FSM) is a mathematical model of computation.
- It is an **abstract** machine that can be in exactly one of a finite number of states at any given time.

# State Machine Concept

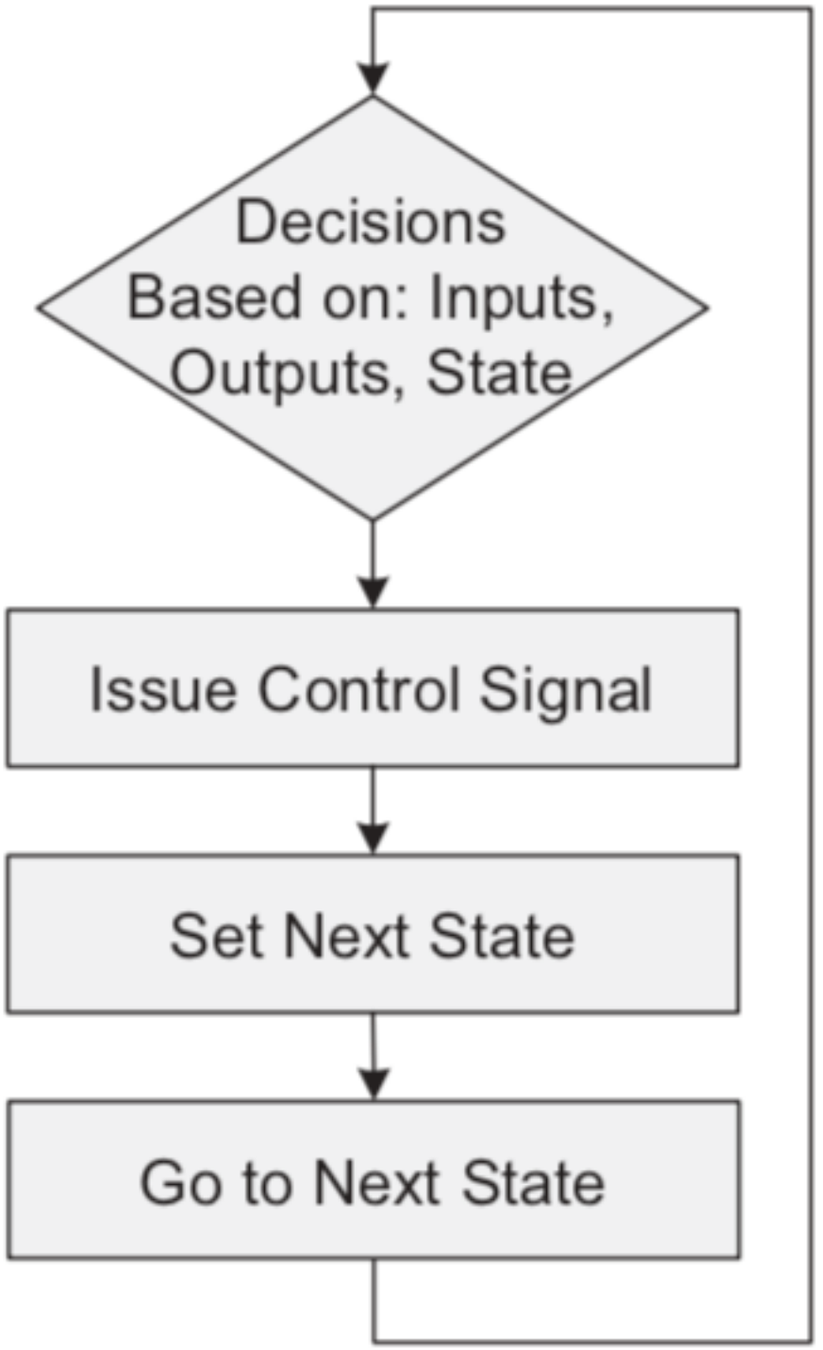
- From Wiki:
- The FSM can change from one state to another in response to some external inputs; the change from one state to another is called a transition.
- An FSM is defined by a list of its states, its initial state, and the conditions for each transition.

# Example1: Controller

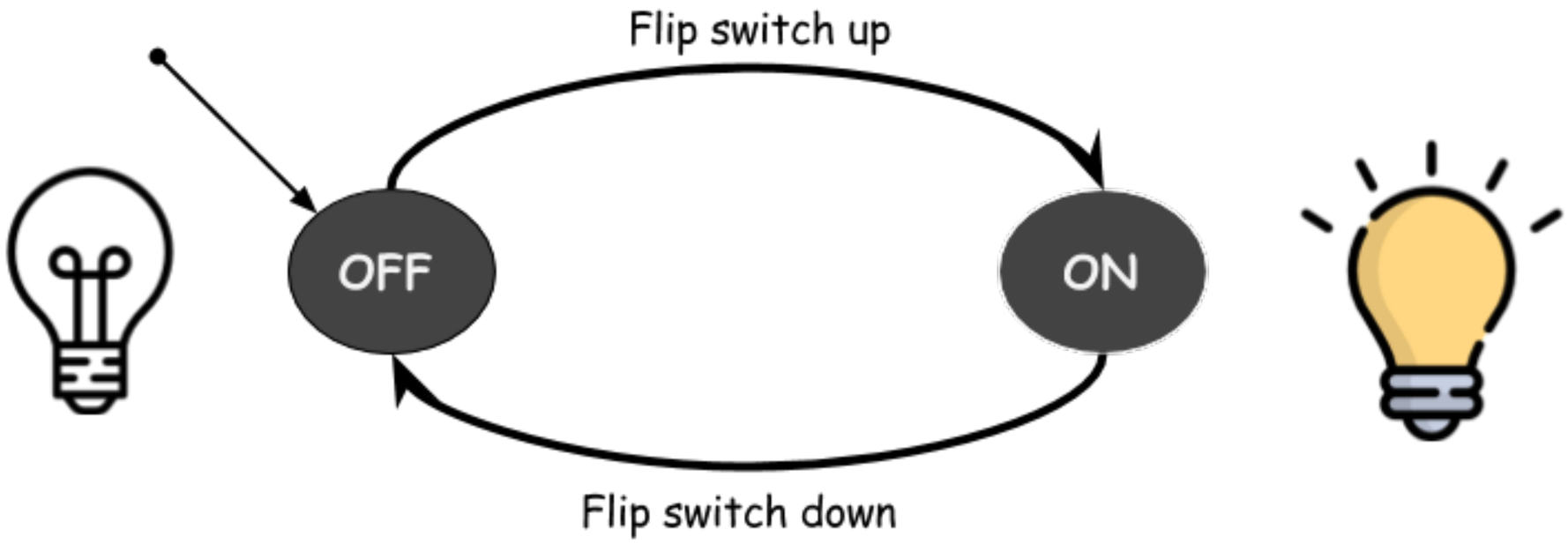
- Data components are put together in the data part of an RT level design, and the controller is wired to the data part to control its flow of data.
- Controllers can be
  - as easy as one flip-flop, handshaking handlers,
  - or as complex as several concurrent state machines

# Example1: Controller

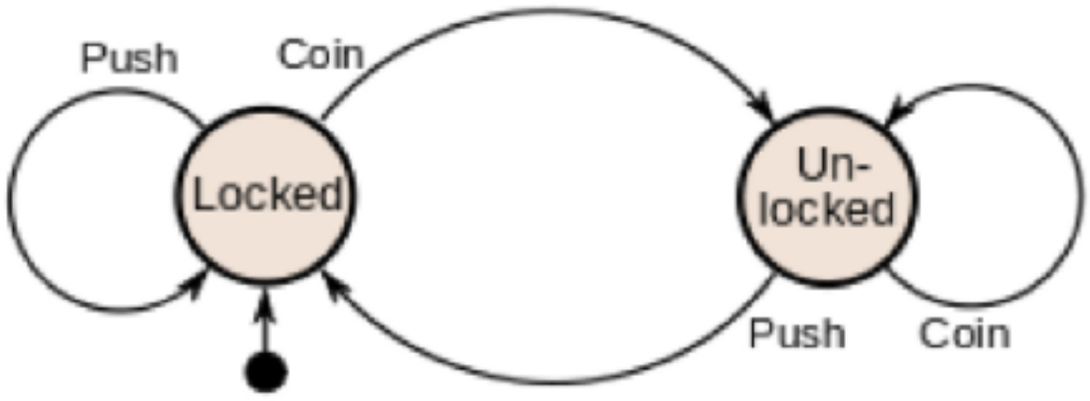
- A controller circuit. The inputs to the controller determine its next states and its outputs.
  - The controller monitors its inputs and makes decisions as to when and what output signals to assert.
  - Controllers keep past history of circuit data by switching to appropriate states.



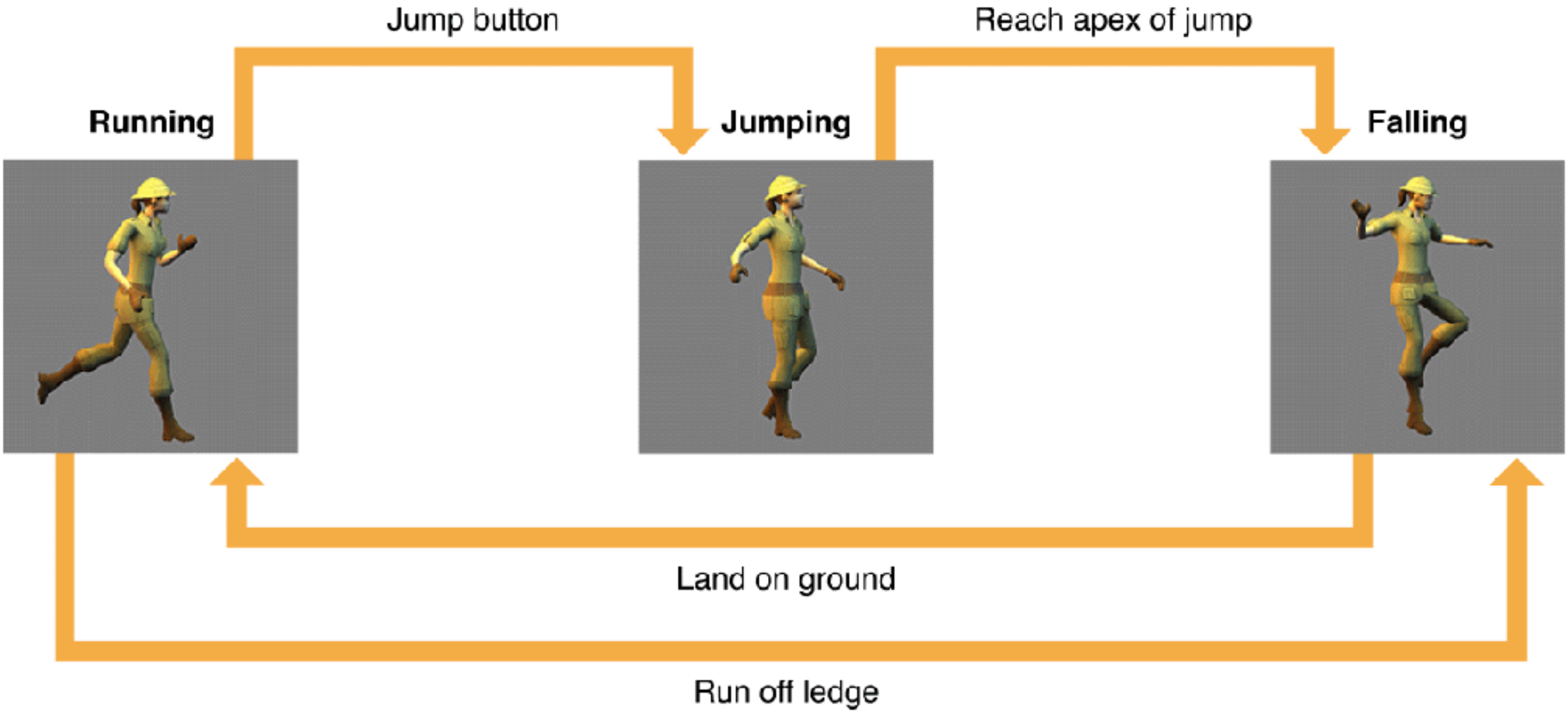
# Example1: Controller



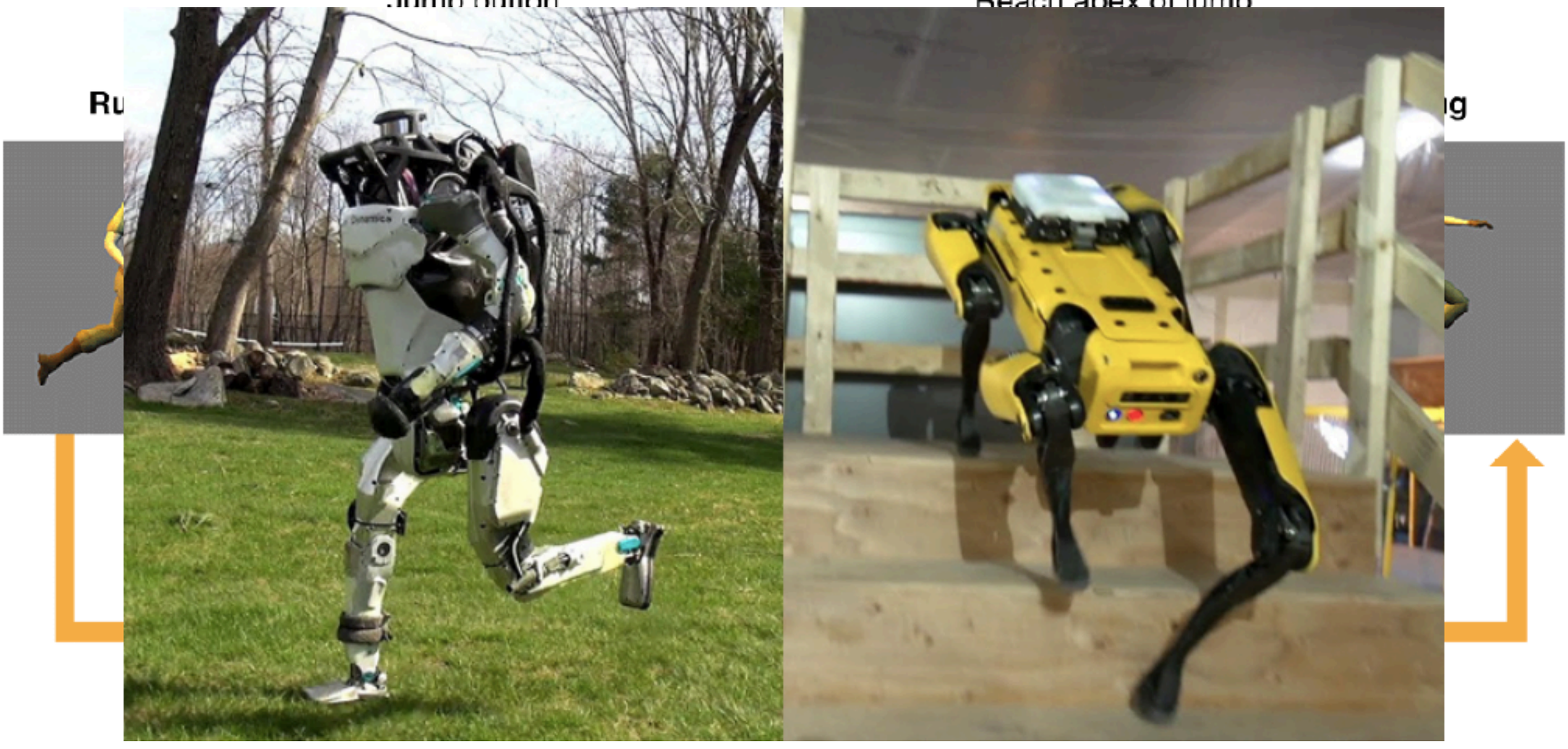
# Example1: Controller

# Example1: Controller

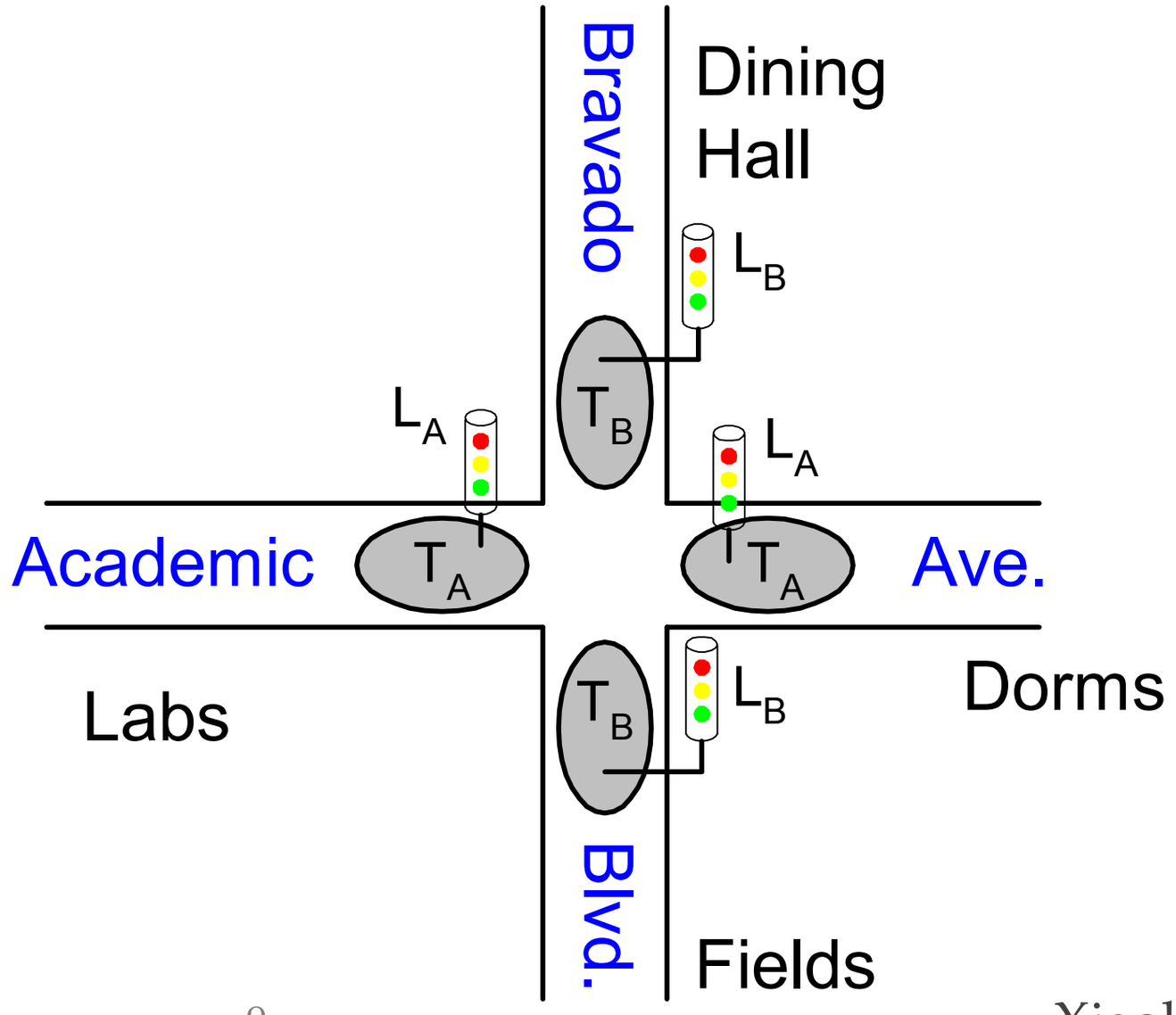


# Example1: Controller



# Example1: Controller

- Traffic light controller
  - Traffic sensors: $T_A$, $T_B$ (TRUE when there's traffic)
  - Lights: $L_A$, $L_B$

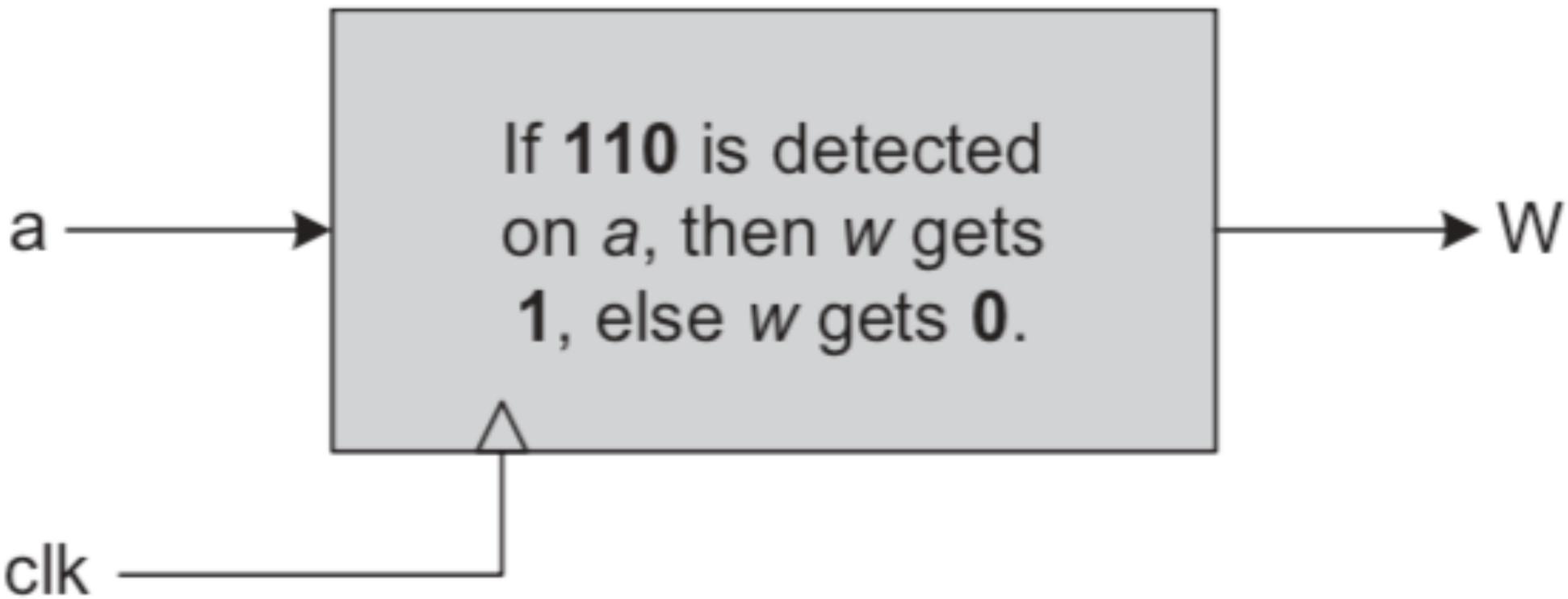


# Example2: Sequence Detector

- A sequence detector is **a good representation for the general class of controllers.**
- Sequence detectors are **simplified controllers**
- Instead of many inputs/outputs, sequence detectors have **one or two input and output lines, and instead of complex decision makings**

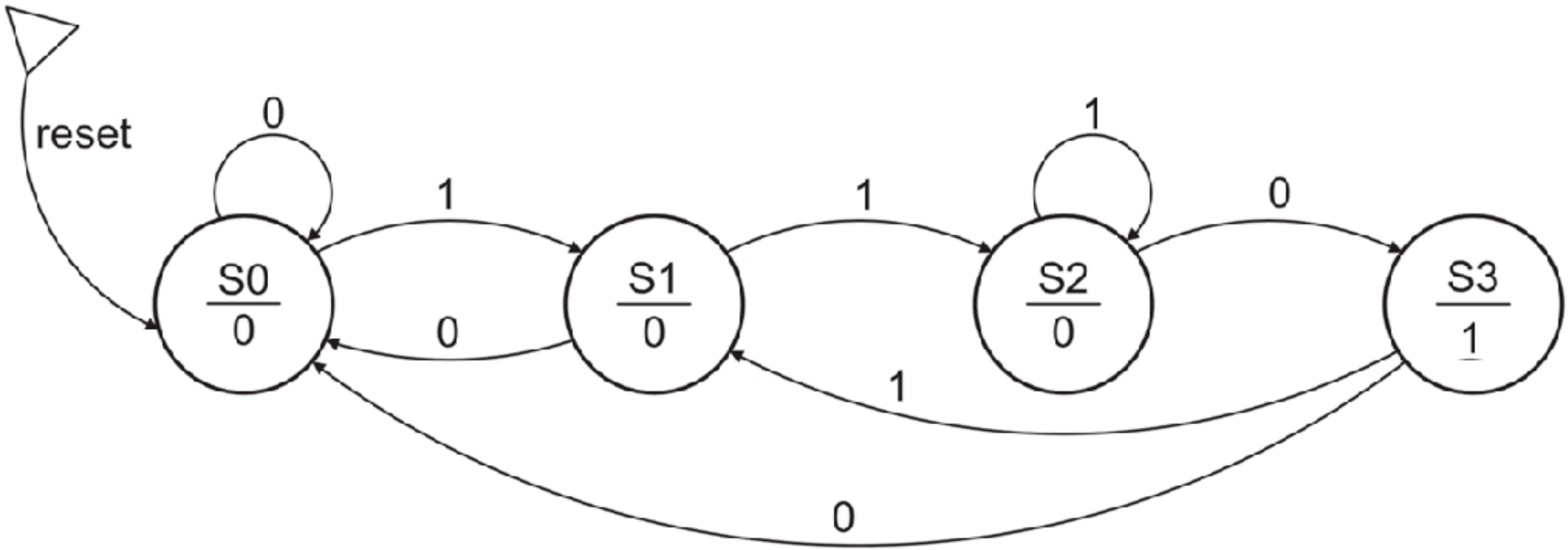
# Example2: Sequence Detector

- **A finite state machine-based sequence detector**
  - Searches on input for the 110 sequence
  - When this sequence is detected in three consecutive clock pulses, the output (w) becomes 1 and stays 1 for a complete clock cycle.



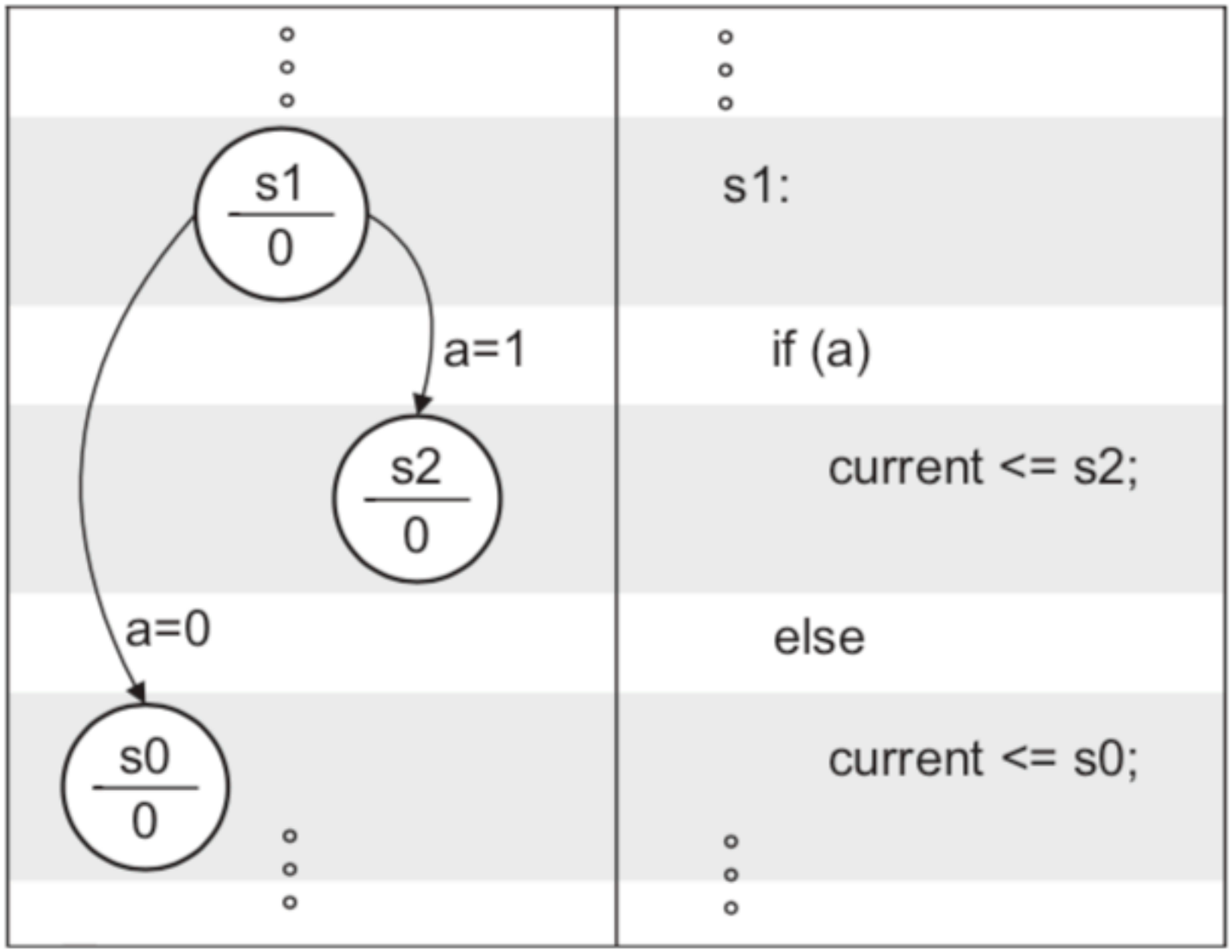
# Example2: Sequence Detector

- A sequence detector
  - Searches on input for the 110 sequence
  - When this sequence is detected in three consecutive clock pulses, the output (w) becomes 1 and stays 1 for a complete clock cycle.



# State Machine

- States: s0, s1, s2, and s3



# We Already Learned FSM

# State Machine

- States: s0, s1, s2, and s3
  - s0 is the reset state and s3 is the state in which the 110 sequence is detected

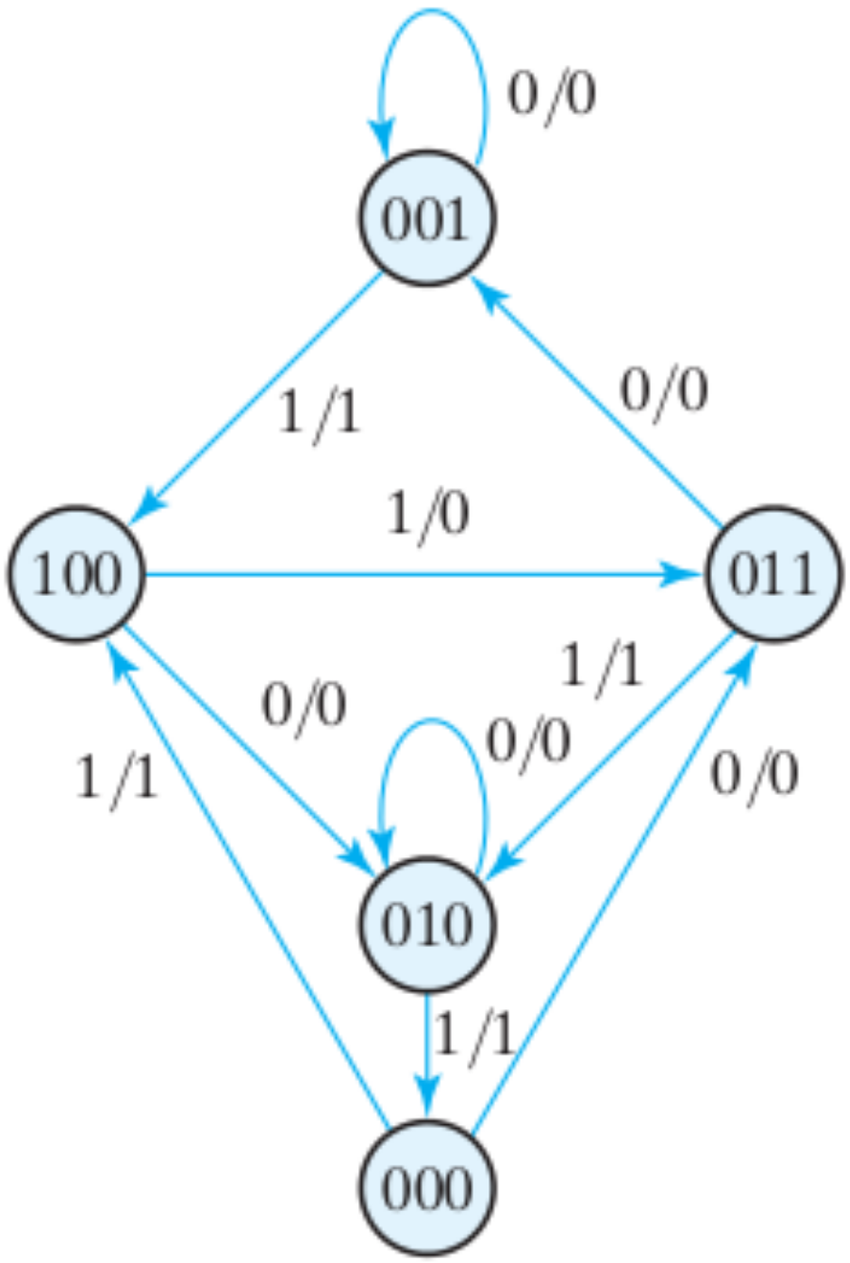
```
`timescale 1ns/100ps

module Detector110 (input a, clk, reset, output w);
   parameter [1:0] s0=2'b00, s1=2'b01, s2=2'b10, s3=2'b11;
   reg [1:0] current;

   always @ (posedge clk) begin
      if (reset) current = s0;
      else
          case (current)
           s0: if (a) current <= s1; else current <= s0;
           s1: if (a) current <= s2; else current <= s0;
           s2: if (a) current <= s2; else current <= s3;
           s3: if (a) current <= s1; else current <= s0;
          endcase
   end

   assign w = (current == s3) ? 1: 0;

endmodule
```

# Syntax of a Verilog Case Statement

- The case statement compares an expression to a series of cases and executes the statement or statement group associated with the first matching case:
  - case statement supports single or multiple statements.
  - Group multiple statements using begin and end keywords.
- Syntax of a case statement look as shown below.
  - case ()
  - < case1 > : < statement >
  - < case2 > : < statement >
  - .....
  - default : < statement >
  - endcase

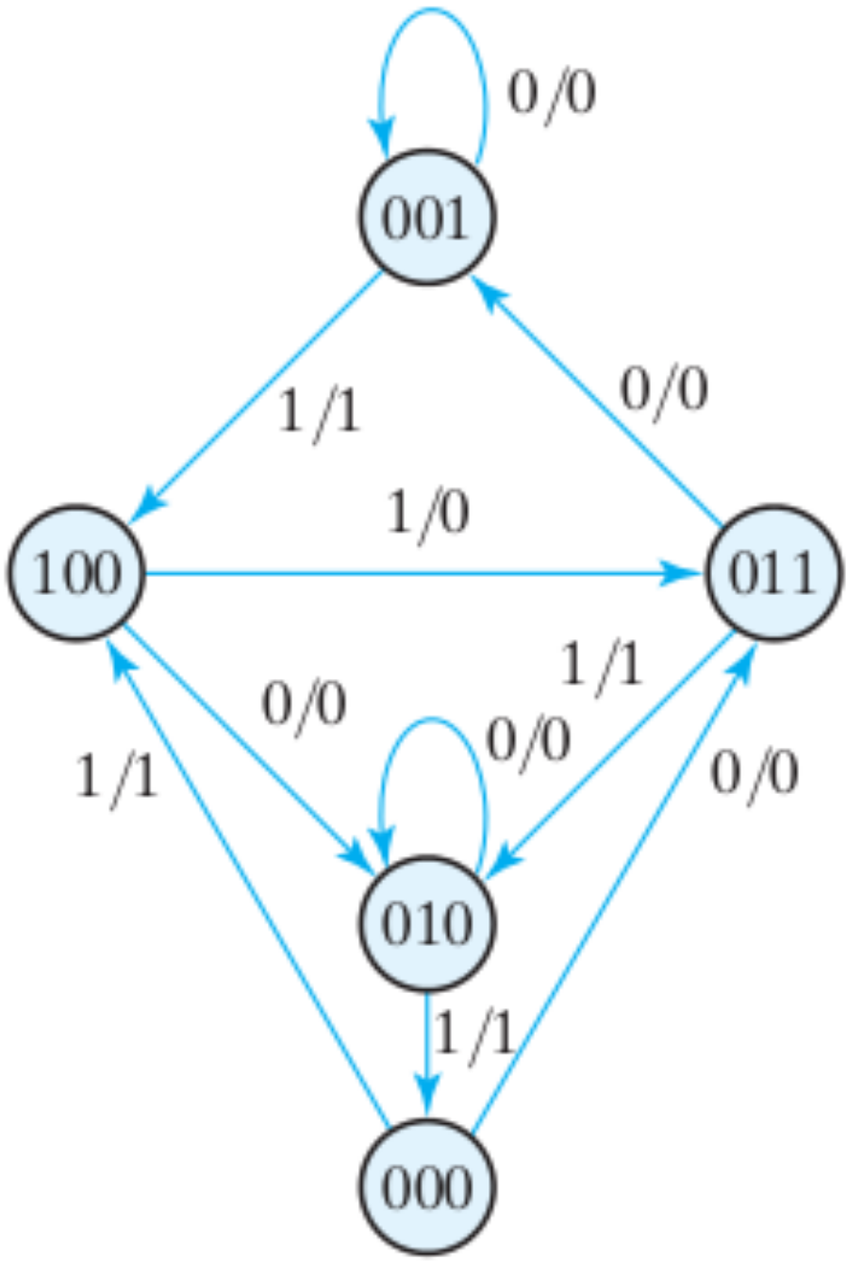
# Applications of Case Statement

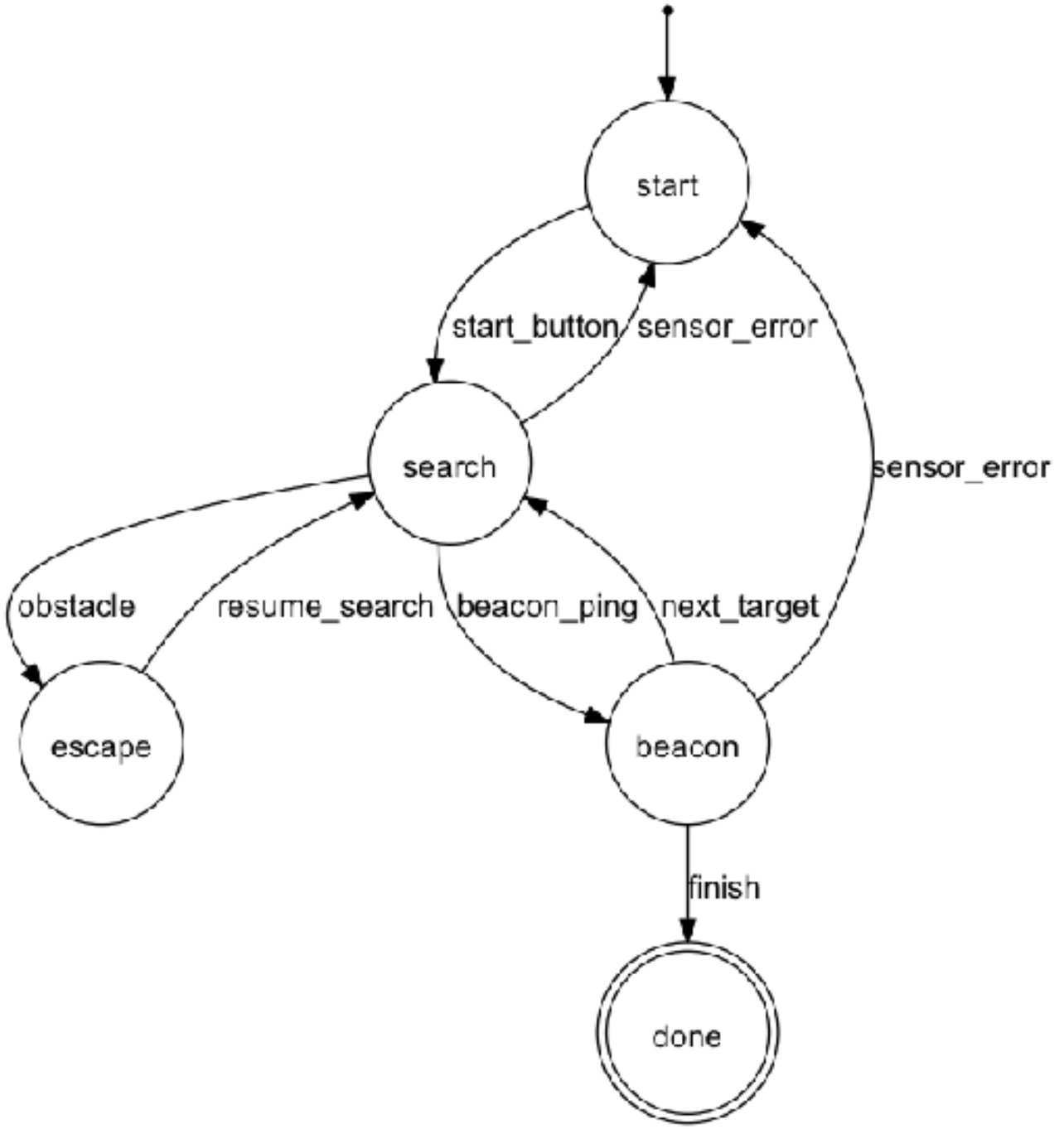
- State transition



# Applications of Case Statement

- State transition



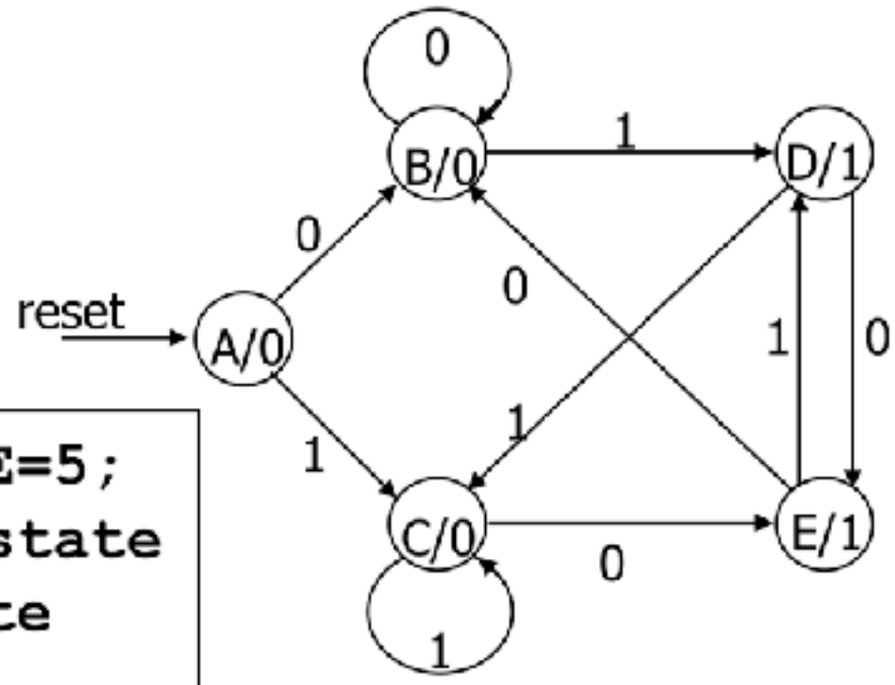
# Syntax of a Verilog Case Statement

```
module statem(clk, in, reset, out);

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

reg [3:0] out;
reg [1:0] state;

parameter zero=0, one=1, two=2, three=3;

always @(state)
     begin
          case (state)
               zero:
                    out = 4'b0000;
               one:
                    out = 4'b0001;
               two:
                    out = 4'b0010;
               three:
                    out = 4'b0100;
               default:
                    out = 4'b0000;
          endcase
     end
```

```
always @(posedge clk or posedge reset)
     begin
          if (reset)
               state = zero;
          else
               case (state)
                    zero:
                         state = one;
                    one:
                         if (in)
                              state = zero;
                         else
                              state = two;
                    two:
                         state = three;
                    three:
                         state = zero;
               endcase
     end

endmodule
```

# Syntax of a Verilog Case Statement

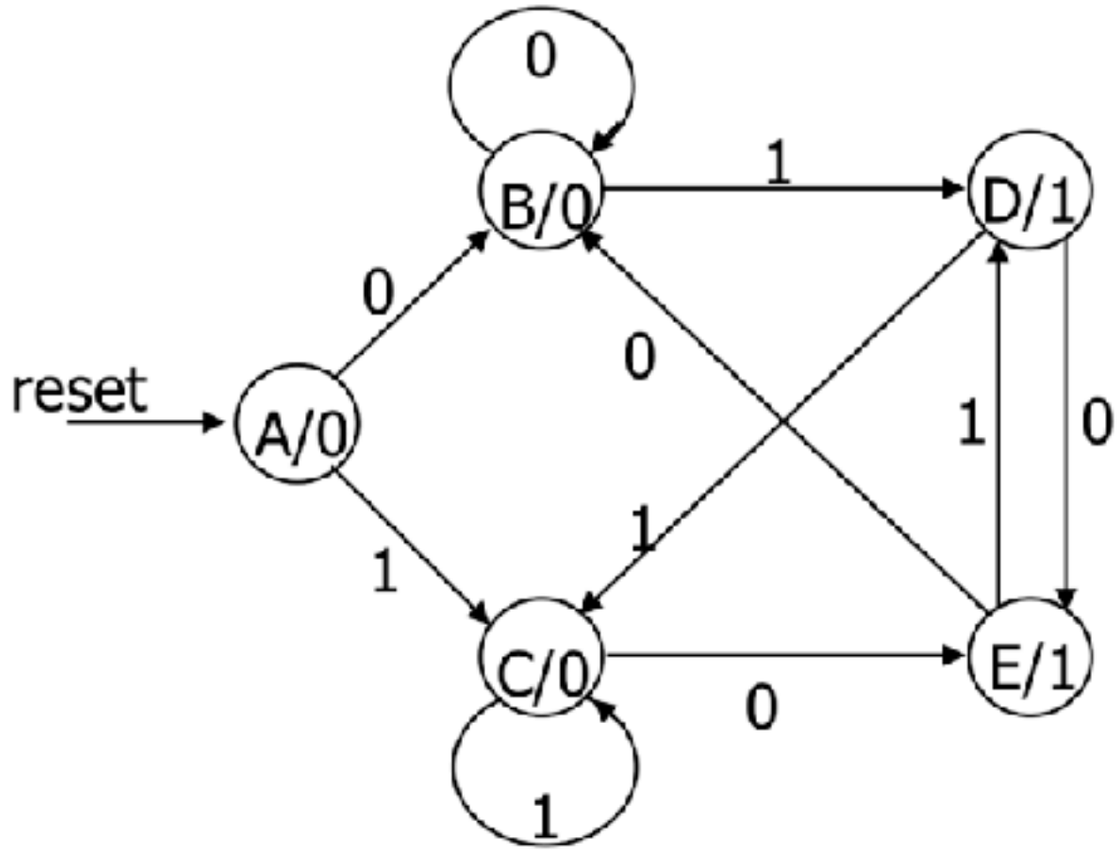
- Edge detector: Moore Machine




```
localparam A=0, B=1, C=2, D=4, E=5;
reg [2:0] state,     // Current state
          nxtState; // Next state

always @(posedge clk) begin
  if (reset) begin
    state <= A;    // Initial state
  end else begin
    state <= nxtState;
  end
end
```

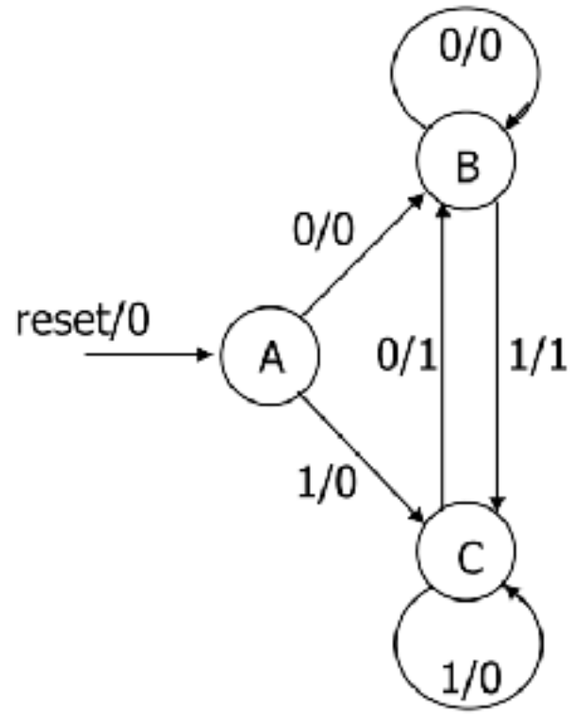
# Verilog case Implementation

```
always @(*) begin
  nxtState = state;
  out = 0;
  case (state)
    A : if (in) nxtState = C;
        else    nxtState = B;
    B : if (in) nxtState = D;
    C : if (~in) nxtState = E;
    D : begin
      out = 1;
      if (in) nxtState = C;
      else    nxtState = E;
    end
    E : begin
      out = 1;
      if (in) nxtState = D;
      else    nxtState = B;
    end
    default : begin
      out = 1'bX;
      nxtState = 3'bX;
    end
  endcase
end
```



# Syntax of a Verilog Case Statement

- Edge detector: Mealy Machine

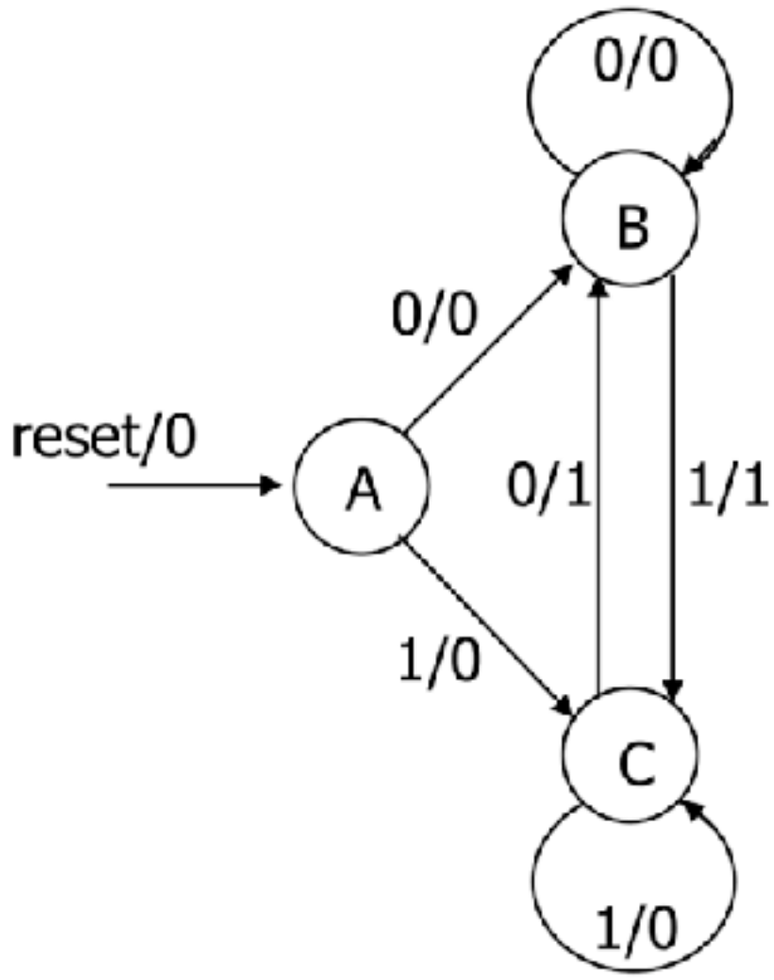



```
localparam A=0, B=1, C=2;
reg [1:0] state,     // Current state
          nxtState; // Next state

always @(posedge clk) begin
  if (reset) begin
    state <= A;    // Initial state
  end else begin
    state <= nxtState;
  end
end
```

# Verilog Case Implementation

Output depends on state and input

```
always @(*) begin
  nxtState = state;
  out = 0;
  case (state)
    A : if (in) nxtState = C;
        else    nxtState = B;
    B : if (in) begin
          out = 1;
          nxtState = C;
        end
    C : if (~in) begin
          out = 1;
          nxtState = B;
        end
    default : begin
      out = 1'bX;
      nxtState = 3'bX;
    end
  endcase
end
```



# State Elements

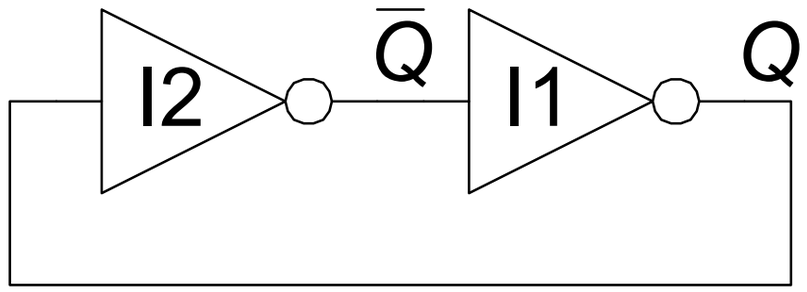
- The state of a circuit influences its future behavior
- State elements store state
- Bistable circuit
- SR Latch
- D Latch
- D Flip-flop

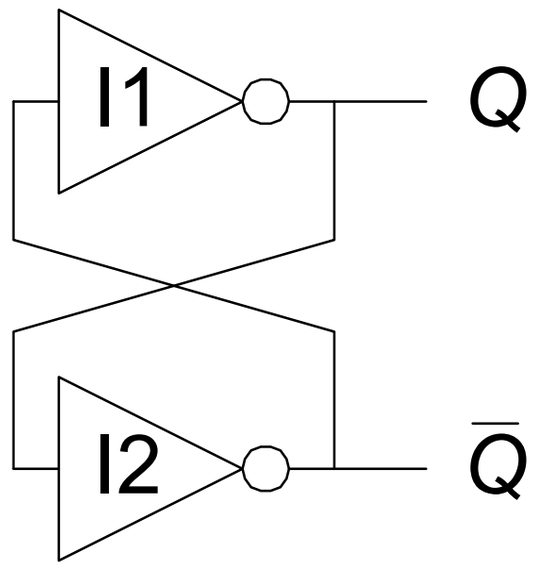
# Sequential Circuits

- Give sequence to events
- Have memory (short-term)
- Use feedback from output to input to store information

# Bistable Circuit

- Fundamental building block of other state elements
- Two outputs: Q, $\overline{Q}$
- No inputs

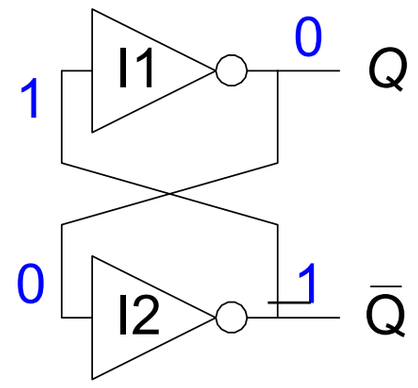


# Bistable Circuit Analysis

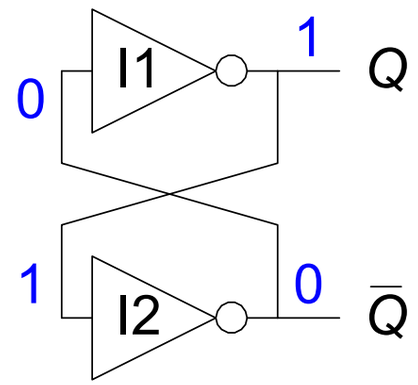
- Consider the two possible cases:
  - $Q = 0$:

    then $Q = 1$, $\bar{Q} = 0$ (consistent)




  - $Q = 1$:

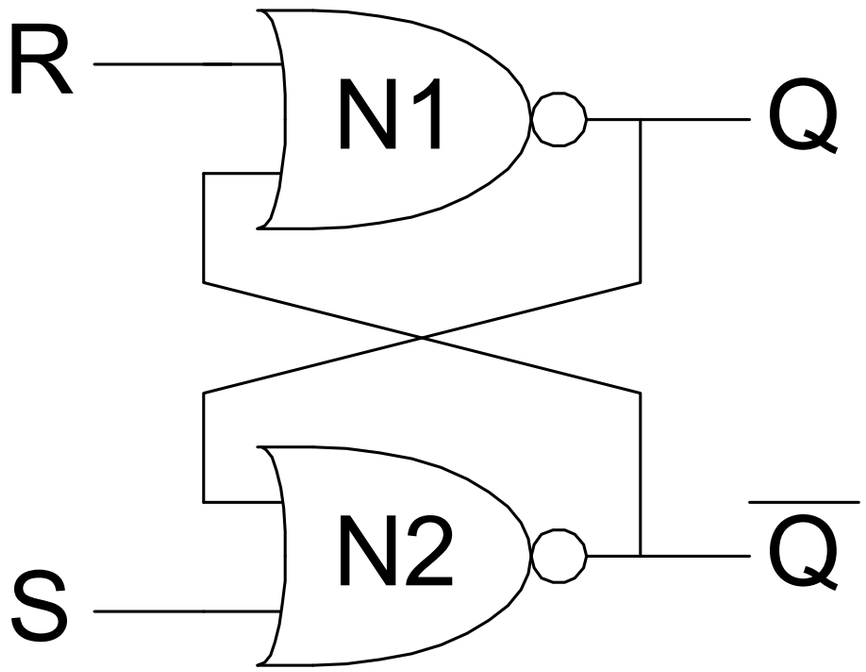
    then $Q = 0$, $\bar{Q} = 1$ (consistent)




- Stores 1 bit of state in the state variable, Q (or $\bar{Q}$)
- But there are **no inputs to control the state**

# SR (Set / Reset) Latch

- SR Latch

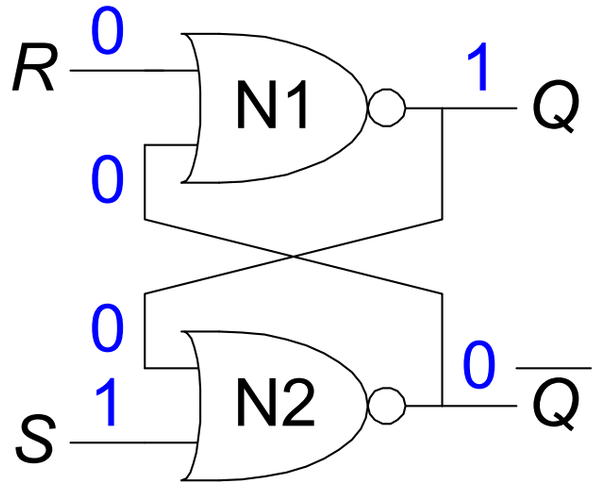



- Consider the four possible cases:
  - $S = 1, R = 0$
  - $S = 0, R = 1$
  - $S = 0, R = 0$
  - $S = 1, R = 1$

# SR Latch Analysis

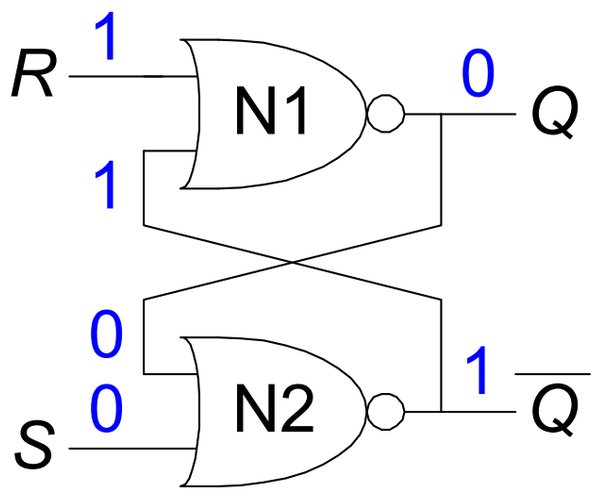
– $S = 1, R = 0$:

then $Q = 1$ and $\overline{Q} = 0$




– $S = 0, R = 1$:

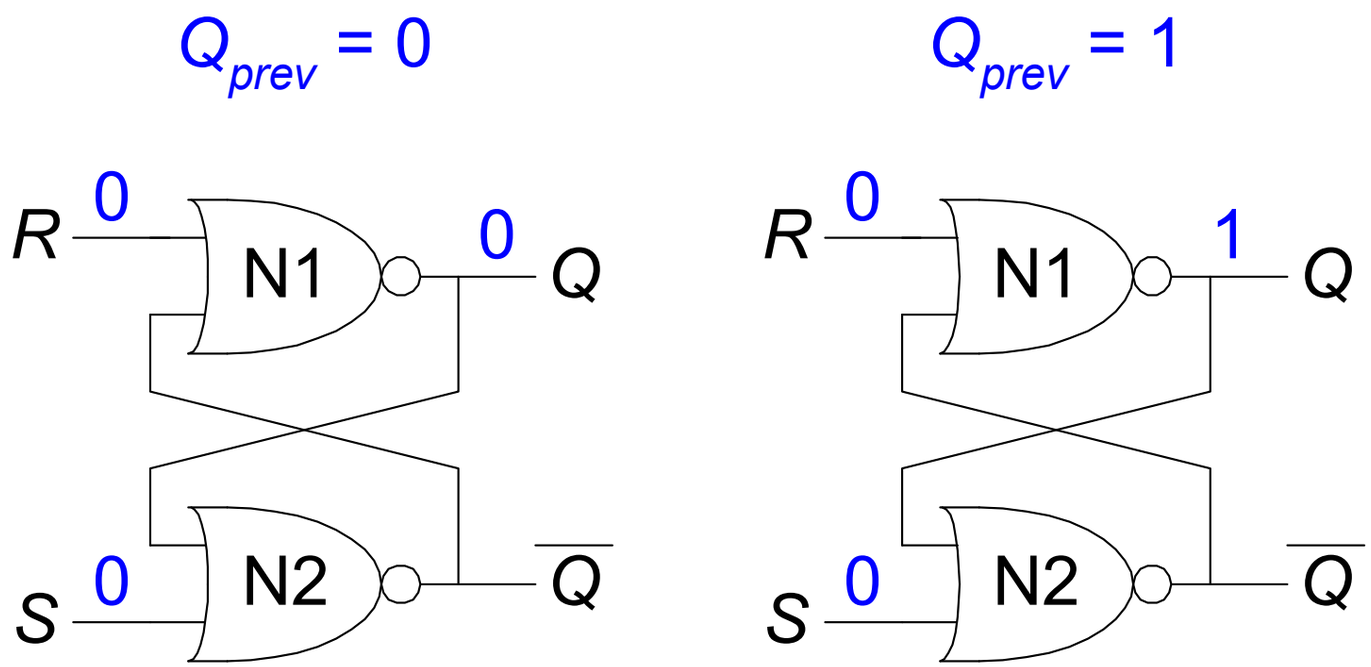
then $Q = 1$ and $\overline{Q} = 0$



# SR Latch Analysis

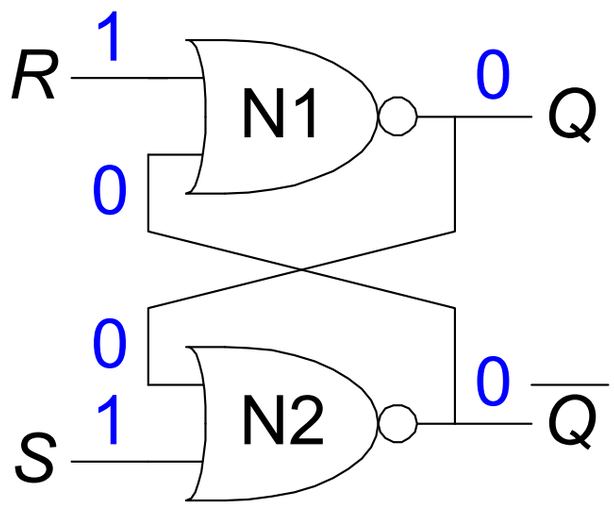
– $S = 0, R = 0$:

then $Q = Q_{prev}$




– $S = 1, R = 1$:

then $Q = 0, \overline{Q} = 0$

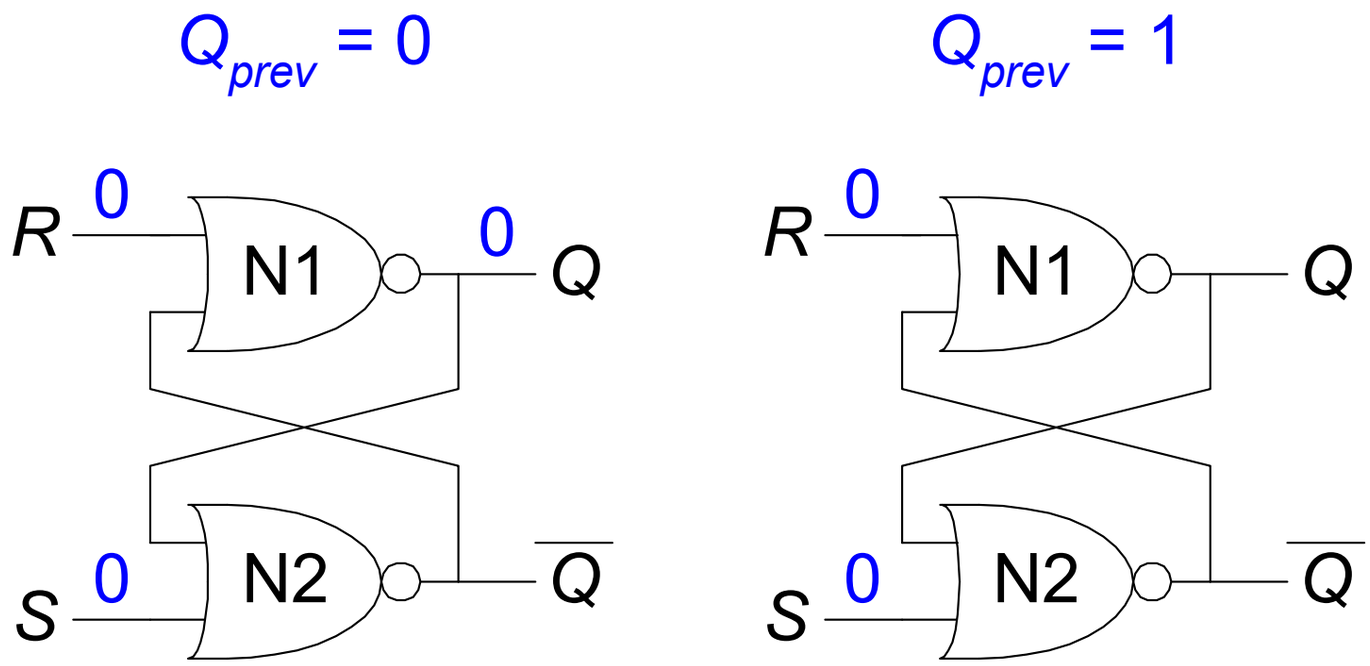


# SR Latch Analysis

– $S = 0, R = 0$:

then $Q = Q_{prev}$

– **Memory!**

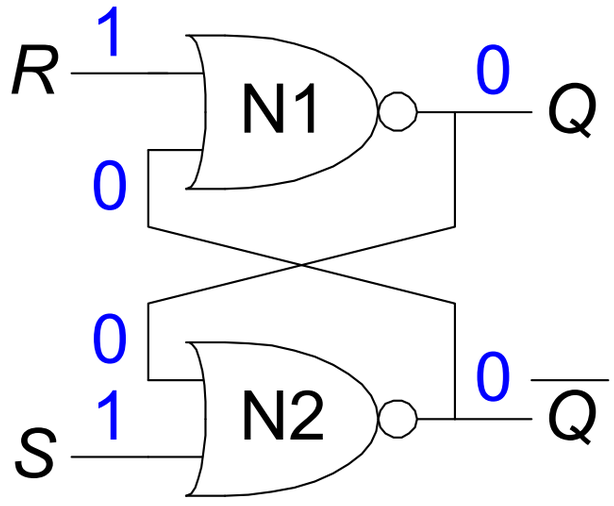



– $S = 1, R = 1$:

then $Q = 0, \overline{Q} = 0$

– **Invalid State**

$\overline{Q} \neq$ NOT $Q$



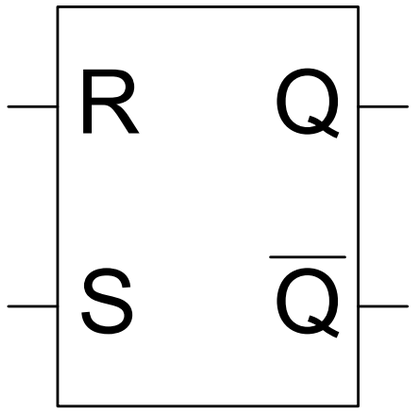
# SR Latch Symbol

- SR stands for Set/Reset Latch
  - Stores one bit of state ($Q$)
- Control what value is being stored with $S$, $R$ inputs
  - **Set:** Make the output 1

    $(S = 1, R = 0, Q = \mathbf{1})$
  - **Reset:** Make the output 0

    $(S = 0, R = 1, Q = \mathbf{0})$
- **Must do something to avoid invalid state (when $S = R = 1$)**

SR Latch Symbol



# D Latch

- Two inputs: *CLK*, *D*
  - ***CLK*:** controls *when* the output changes
  - ***D*** (the data input): controls *what* the output changes to
- Function
  - When ***CLK* = 1**,

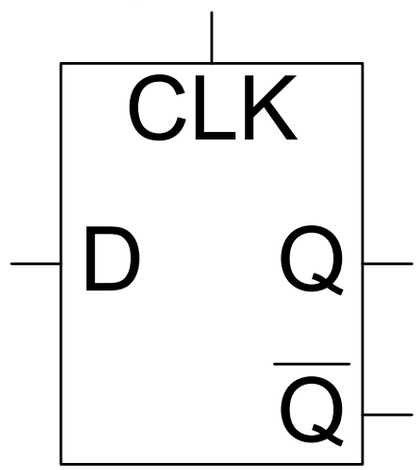
    *D* passes through to *Q* (*transparent*)
  - When ***CLK* = 0**,

    *Q* holds its previous value (*opaque*)
- Avoids invalid ~~c~~ase when

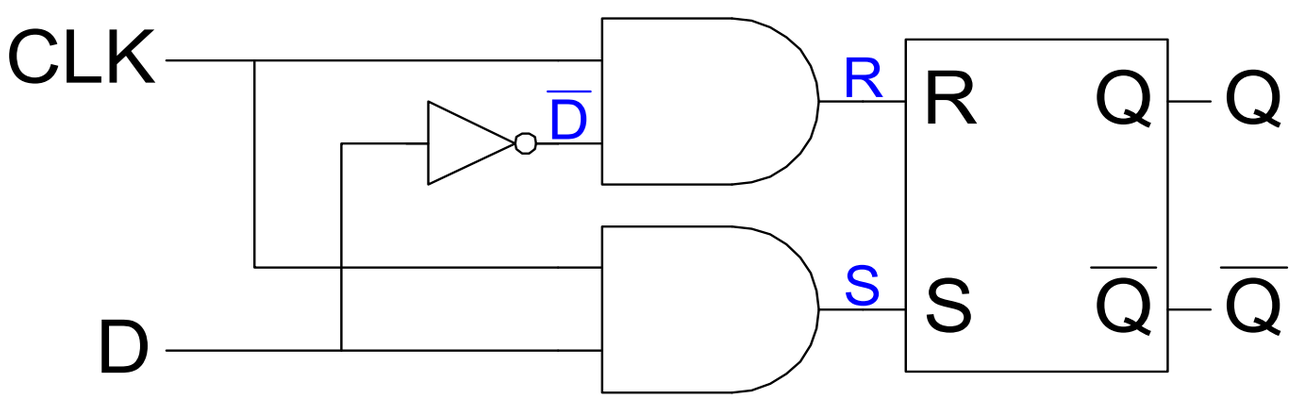
  $Q \neq$ NOT $Q$

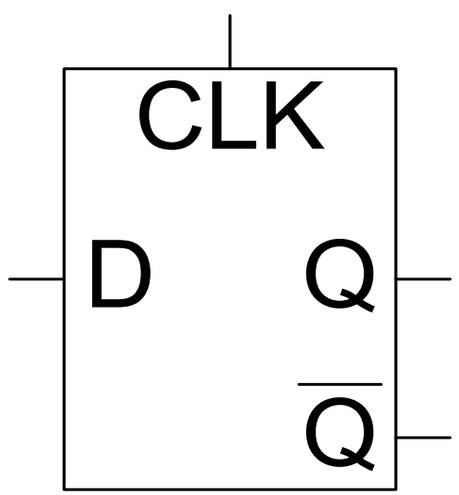
D Latch

Symbol



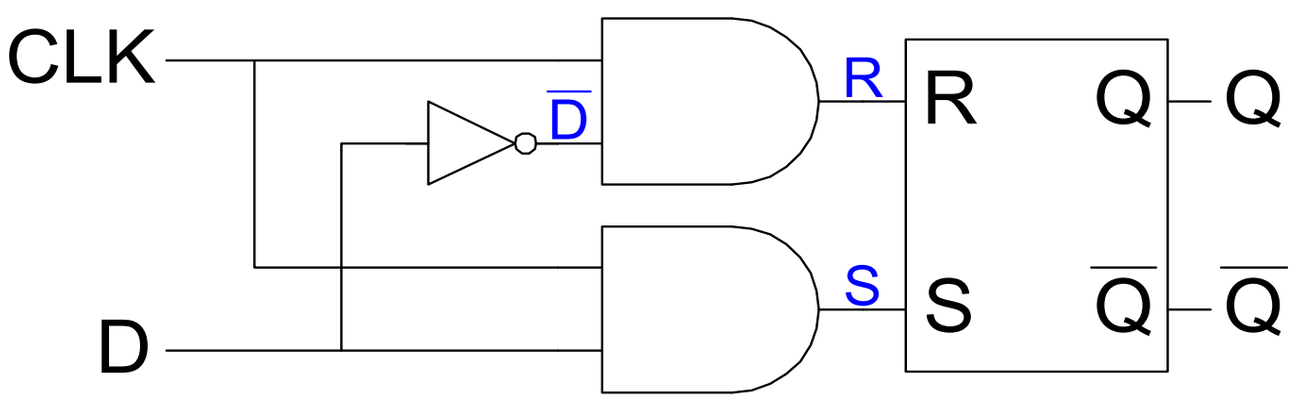
# D Latch Internal Circuit

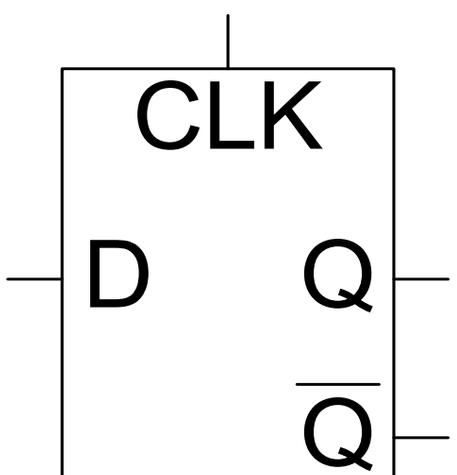



| *CLK* | *D* | $\overline{D}$ | *S* | *R* | Q | $\overline{Q}$ |
|---|---|---|---|---|---|---|
| 0 | X | | | | | |
| 1 | 0 | | | | | |
| 1 | 1 | | | | | |

# D Latch Internal Circuit



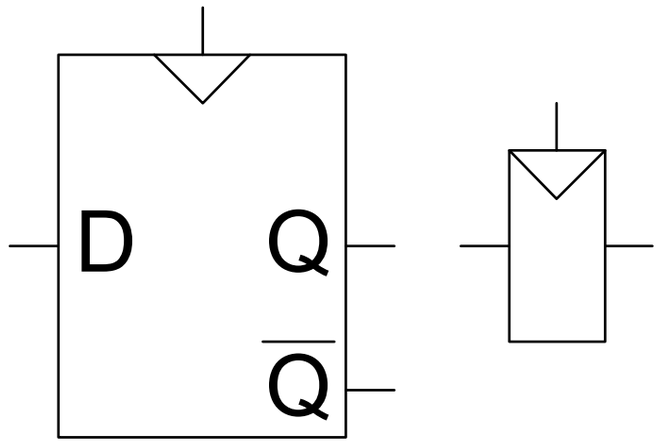




| *CLK* | *D* | $\overline{D}$ | *S* | *R* | *Q* | $\overline{Q}$ |
|---|---|---|---|---|---|---|
| 0 | X | $\overline{X}$ | 0 | 0 | $Q_{prev}$ | $\overline{Q}_{prev}$ |
| 1 | 0 | 1 | 0 | 1 | 0 | 1 |
| 1 | 1 | 0 | 1 | 0 | 1 | 0 |

# D Flip-Flop

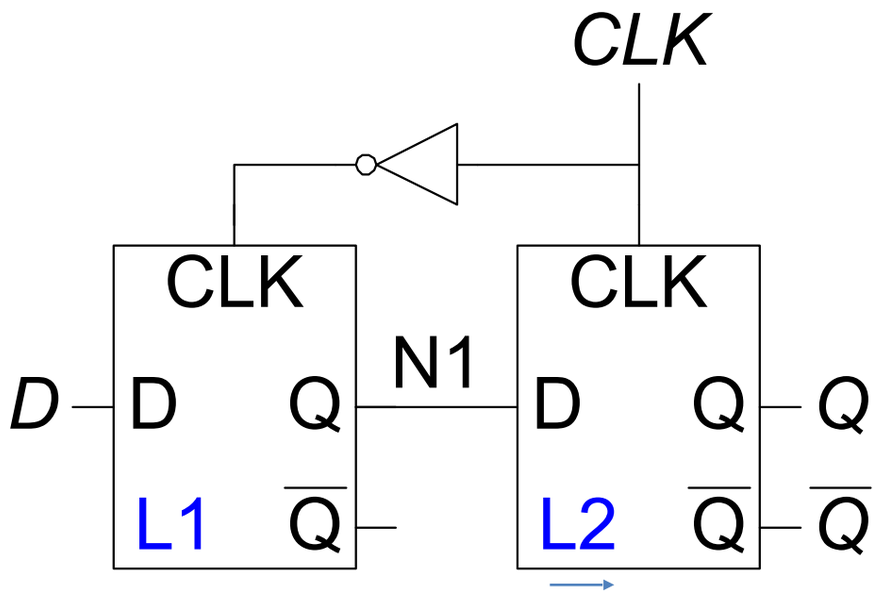
- **Inputs:** *CLK*, *D*
- **Function**
  - Samples *D* on rising edge of *CLK*
    - When *CLK* rises from 0 to 1, *D* passes through to *Q*
    - Otherwise, *Q* holds its previous value
  - *Q* changes only on rising edge of *CLK*
- Called *edge-triggered*
- Activated on the clock edge

D Flip-Flop Symbols

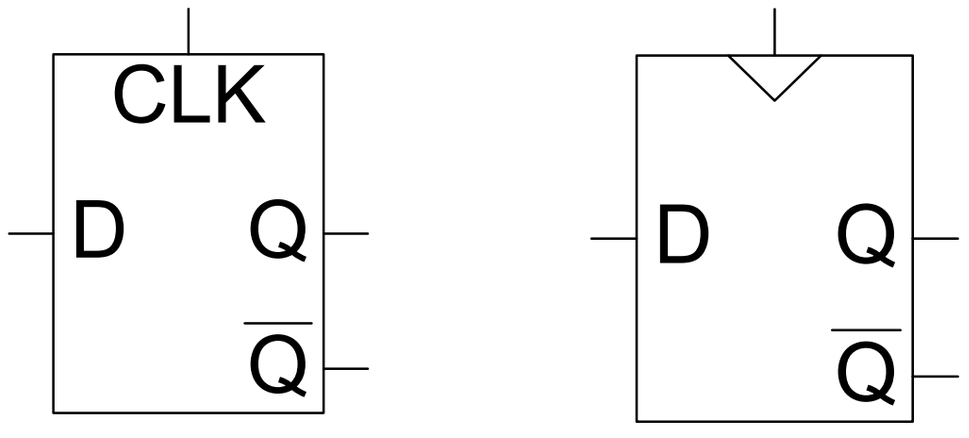


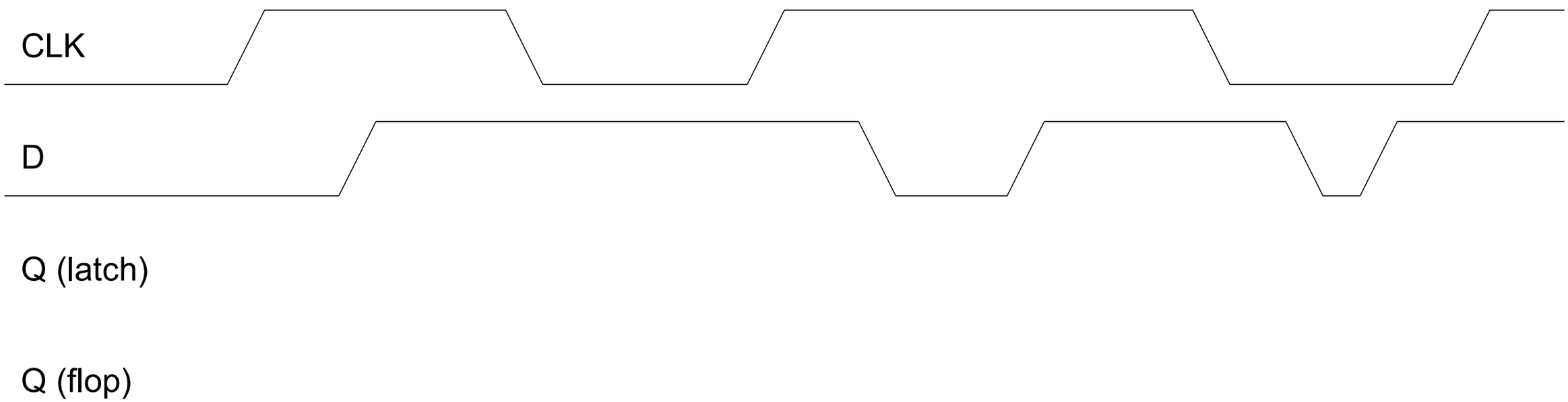
# D Flip-Flop Internal Circuit

- Two back-to-back latches (L1 and L2) controlled by complementary clocks
- When ***CLK* = 0**
  - L1 is transparent
  - L2 is opaque
  - *D* passes through to N1
- When ***CLK* = 1**
  - L2 is transparent
  - L1 is opaque
  - N1 passes through to *Q*
- Thus, on the edge of the clock (when ***CLK* rises from 0  1**)
  - *D* passes through to *Q*

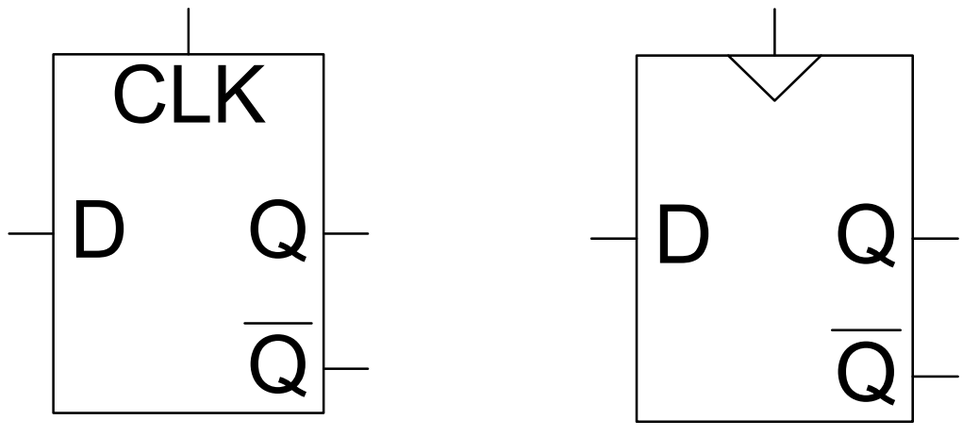


# D Latch vs. D Flip-Flop



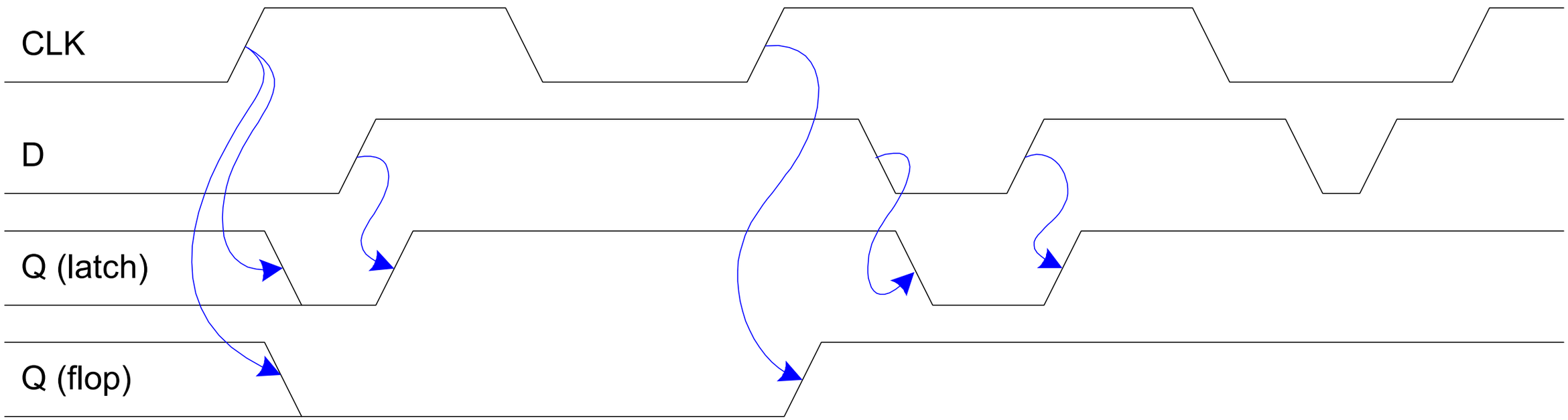



# D Latch vs. D Flip-Flop

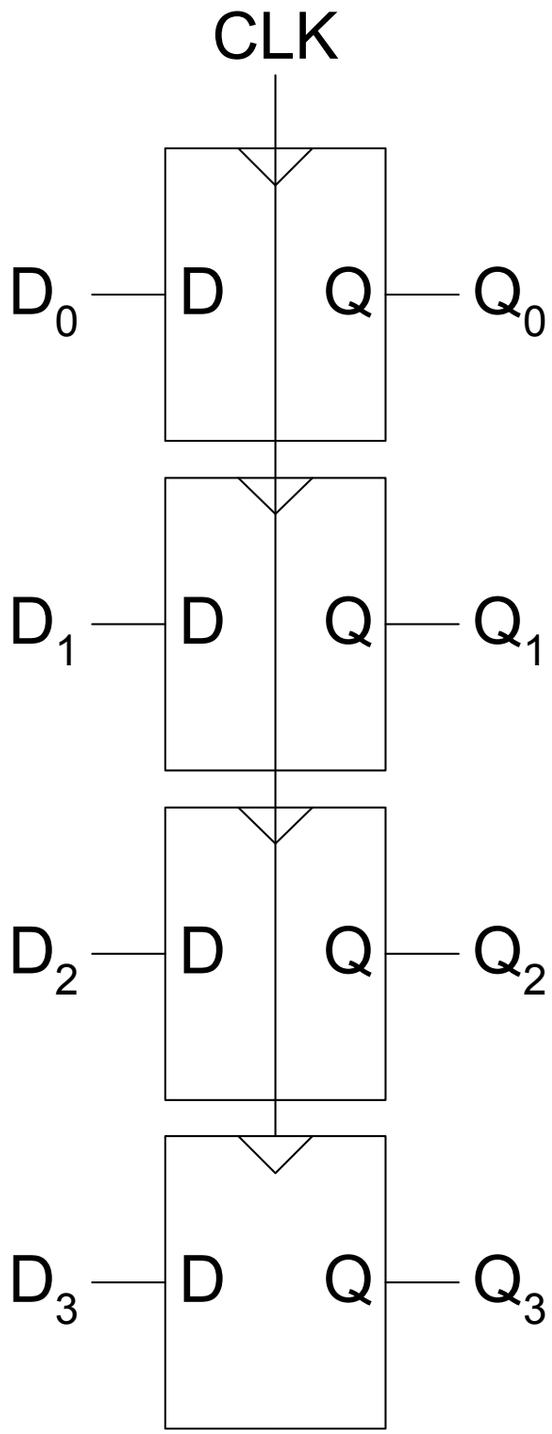


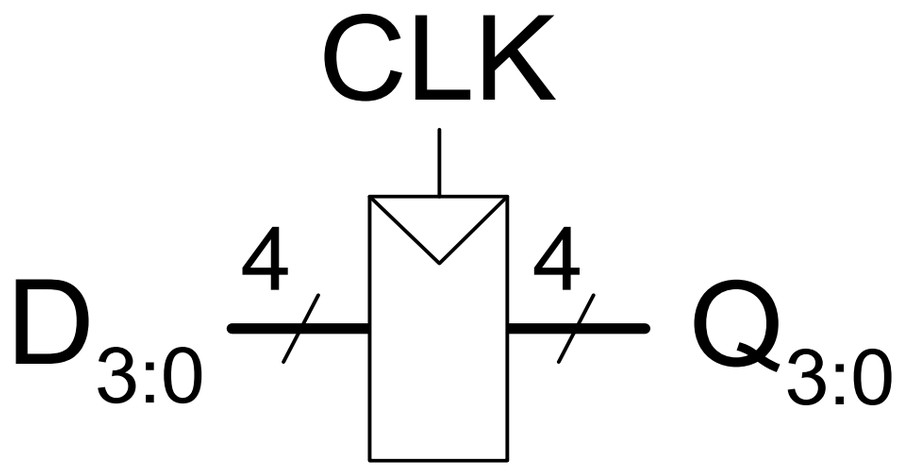






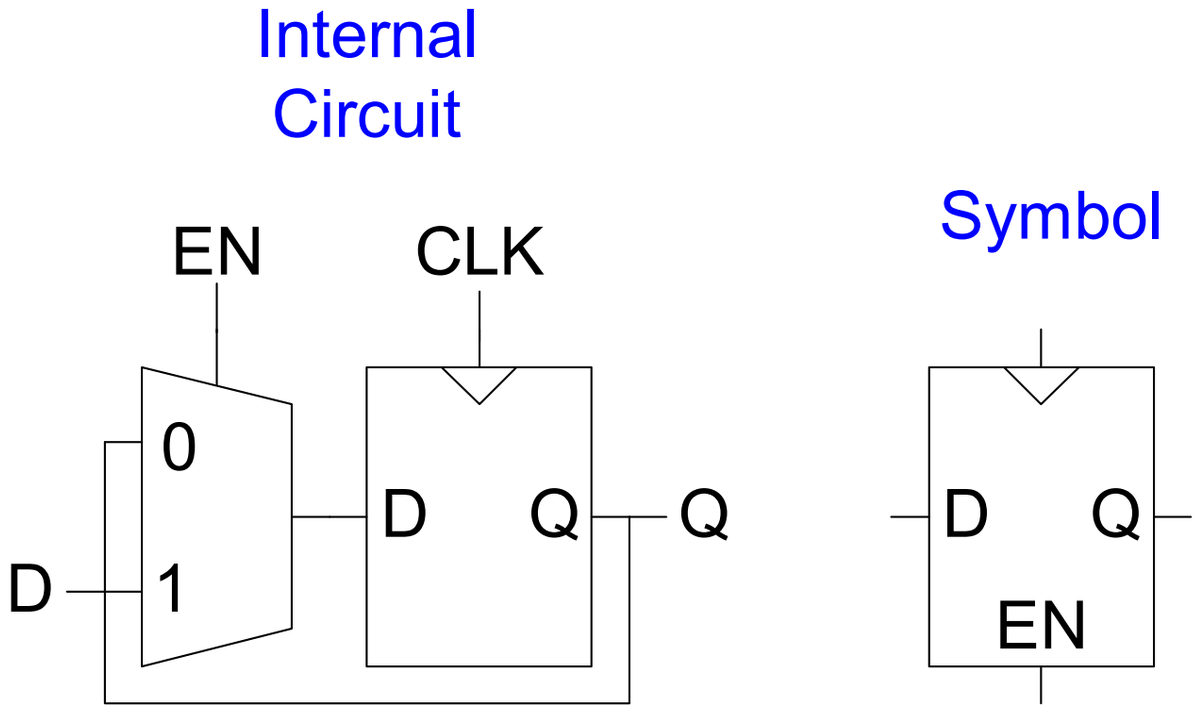
# Registers



# Enabled Flip-Flops

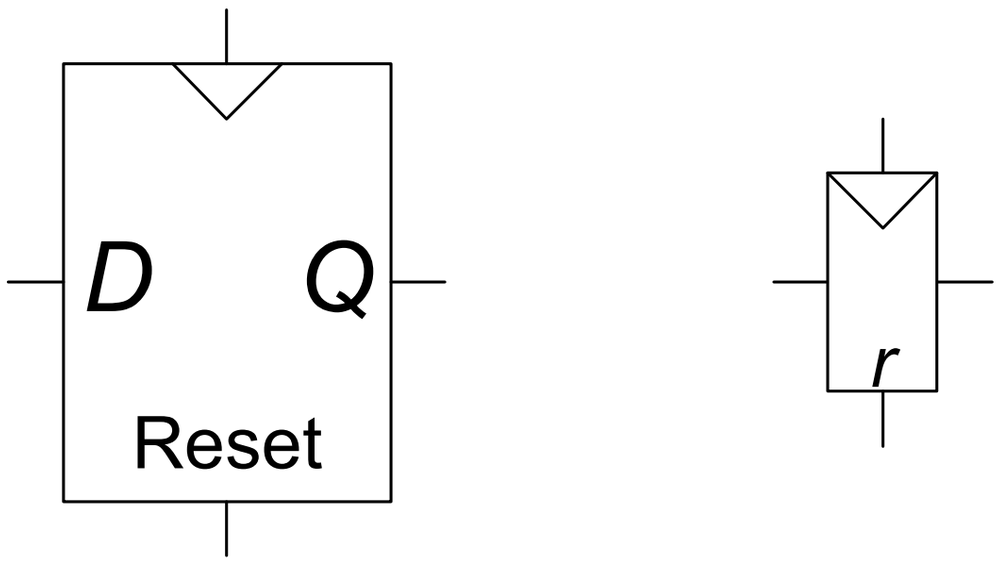
- **Inputs:** *CLK*, *D*, *EN*
  - The enable input (*EN*) controls when new data (*D*) is stored
- **Function**
  - ***EN* = 1:** *D* passes through to *Q* on the clock edge
  - ***EN* = 0:** the flip-flop retains its previous state



# Resettable Flip-Flops

- **Inputs:** *CLK*, *D*, *Reset*
- **Function:**
  - ***Reset* = 1:** *Q* is forced to 0
  - ***Reset* = 0:** flip-flop behaves as ordinary D flip-flop

## Symbols

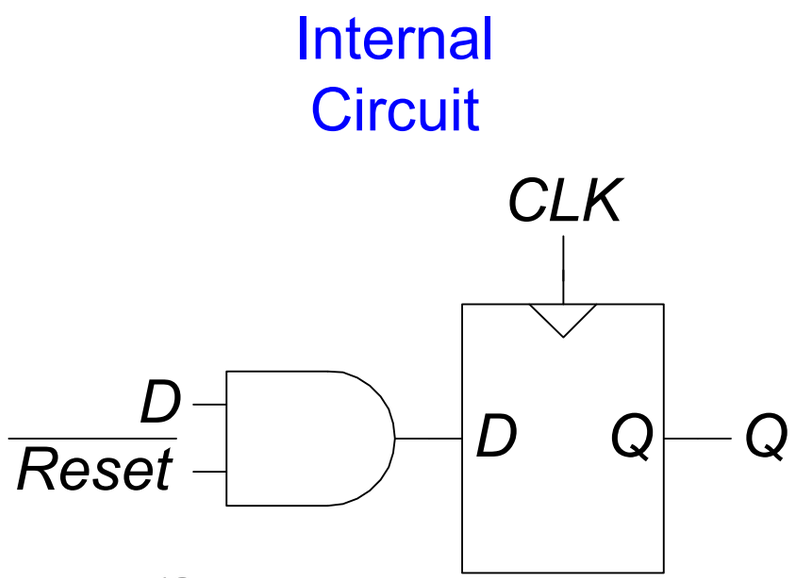


# Resettable Flip-Flops

- Two types:
  - **Synchronous:** resets at the clock edge only
  - **Asynchronous:** resets immediately when *Reset* = 1
- Asynchronously resettable flip-flop requires changing the internal circuitry of the flip-flop
- Synchronously resettable flip-flop?

# Resettable Flip-Flops

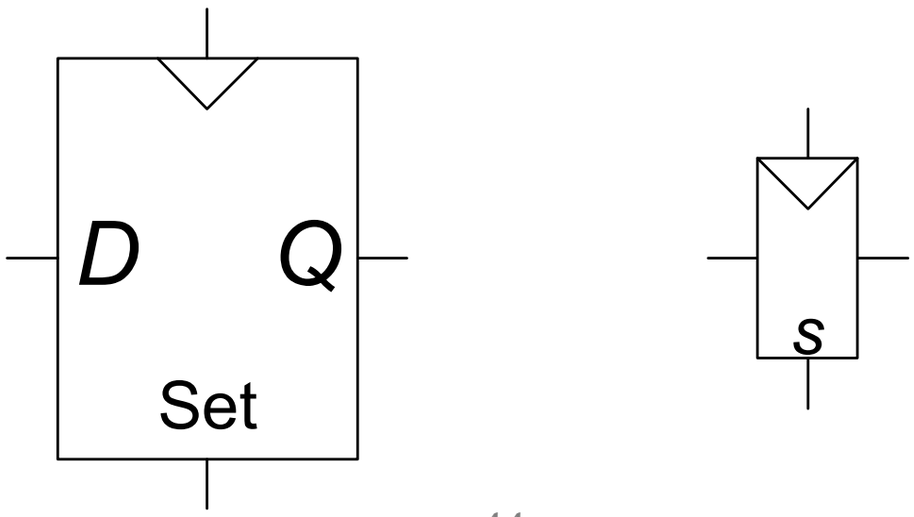
- Two types:
  - **Synchronous:** resets at the clock edge only
  - **Asynchronous:** resets immediately when *Reset* = 1
- Asynchronously resettable flip-flop requires changing the internal circuitry of the flip-flop
- Synchronously resettable flip-flop?



# Settable Flip-Flops

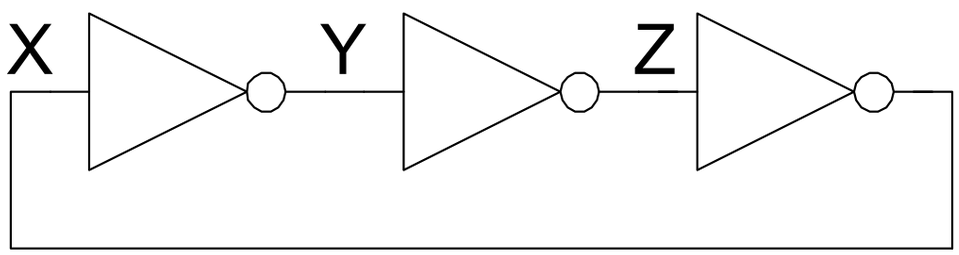
- **Inputs:** *CLK*, *D*, *Set*
- **Function:**
  - ***Set* = 1:** *Q* is set to 1
  - ***Set* = 0:** the flip-flop behaves as ordinary D flip-flop

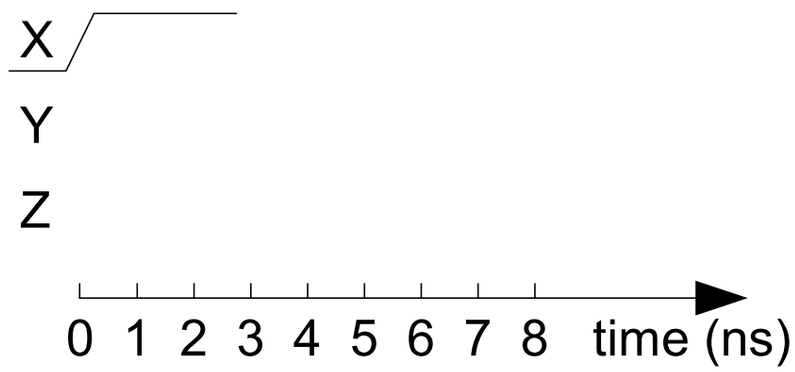
## Symbols



# Sequential Logic

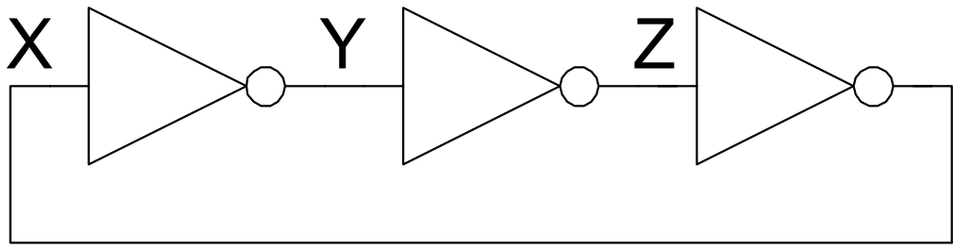
- Sequential circuits: all circuits that aren't combinational
- A problematic circuit:

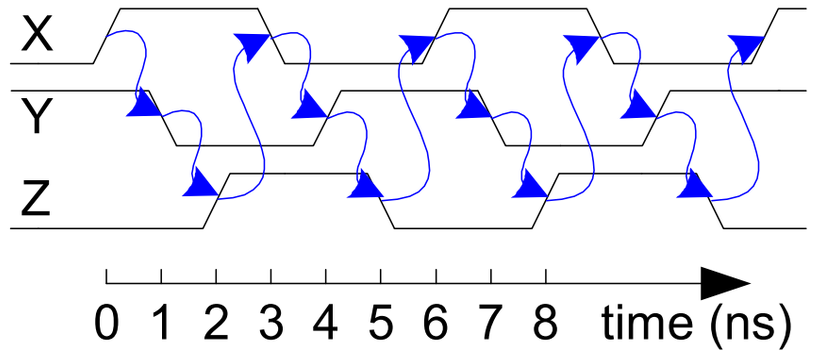


# Sequential Logic

- Sequential circuits: all circuits that aren't combinational
- A problematic circuit:




- No inputs and 1-3 outputs
- Astable circuit, oscillates
- Period depends on inverter delay
- It has a ***cyclic path***: output fed back to input