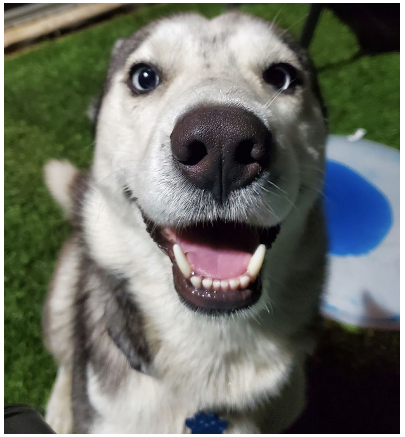
EECS 270 W25
Midterm II Exam Review

March 27th, 2025
By: Mick Gordinier

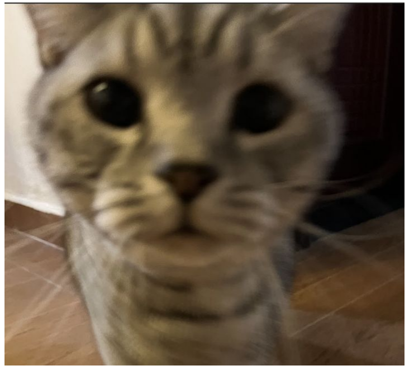


Midterm Exam Logistics (Date/Time)

- Midterm Exam Date/Time: **Tuesday April 1st, 6:00 - 8:00 PM**
  - **A - L: 220 CHRYS**
  - **M - R: 1010 DOW**
  - **S - V: 1017 DOW**
  - **W - Z: 1018 DOW**

- SSD Accommodations
  - **Tuesday April 1, 5:00 pm - 8:00 pm, EECS 270 Lab (2322 EECS)**

- Closed book except for **TWO 8.5"x11" sheet**
- No electronics



DOUBLE CHECK THE CANVAS ANNOUNCEMENT TO CONFIRM / UPDATES

Midterm II Exam Logistics (Topics Covered)

1. Exam 1 Coverage **(Watch previous Midterm Review)**
2. Latches & Flip-Flops
3. Sequential Circuit Analysis / Design
4. Two Level Logic Minimization
5. Sequential Building Blocks (Counters/Shift Regs/…)
6. Register Transfer Logic (RTL) Design

Let’s Get Started

The 2 Types of Digital Logic Circuits (Combinational vs. Sequential)

Combinational Circuits Overview (Projects 1 - 4)

- Outputs depend ONLY on present combination of current inputs
- Stateless / No Memory
- Verilog ‘assign’ - Continuous assignment to a wire data type
- Verilog ‘always @*’ - Procedural “combinational” block

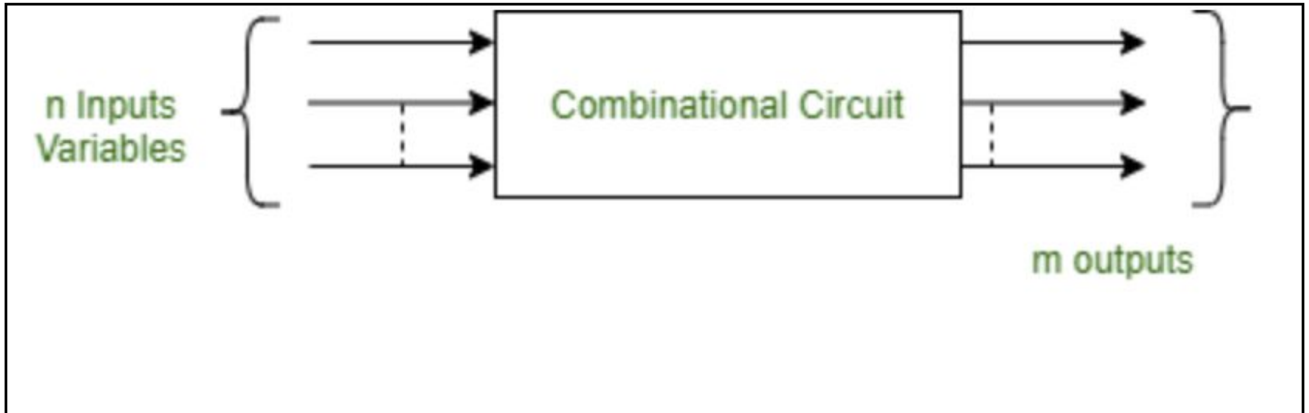
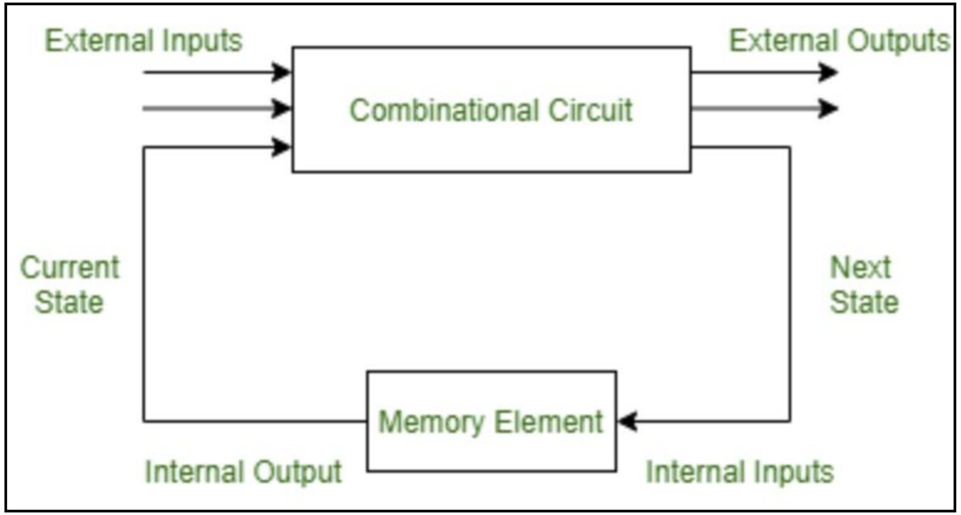


Figure - Block diagram of Combinational circuit

SRC

Sequential Circuits Overview (Project 5 - 7)

- Combinational Circuit + **Memory Element**
- Outputs depend on present combination of inputs and current state
- Use of memory to store previous state/output

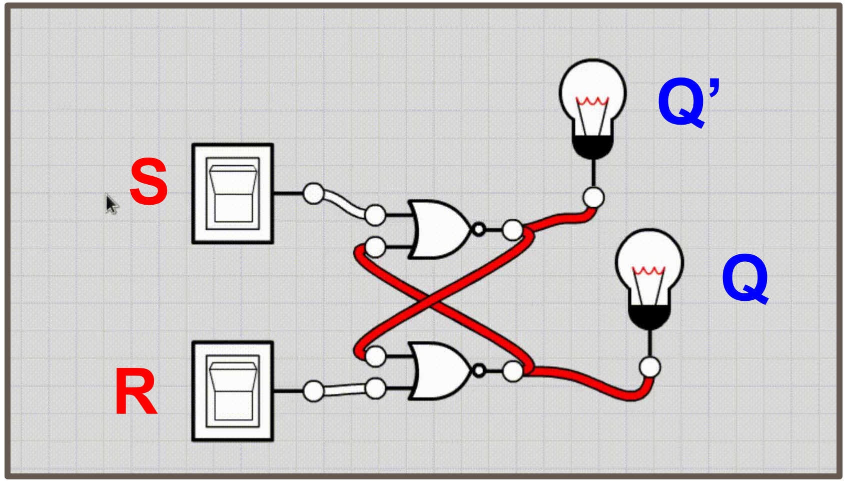


Latches and Flip-Flops
(The Memory Element)

Latches Overview

- Basic type of memory storage unit
    - Used as temporary storage elements to store binary information
    - Can hold/store 1 bit of information

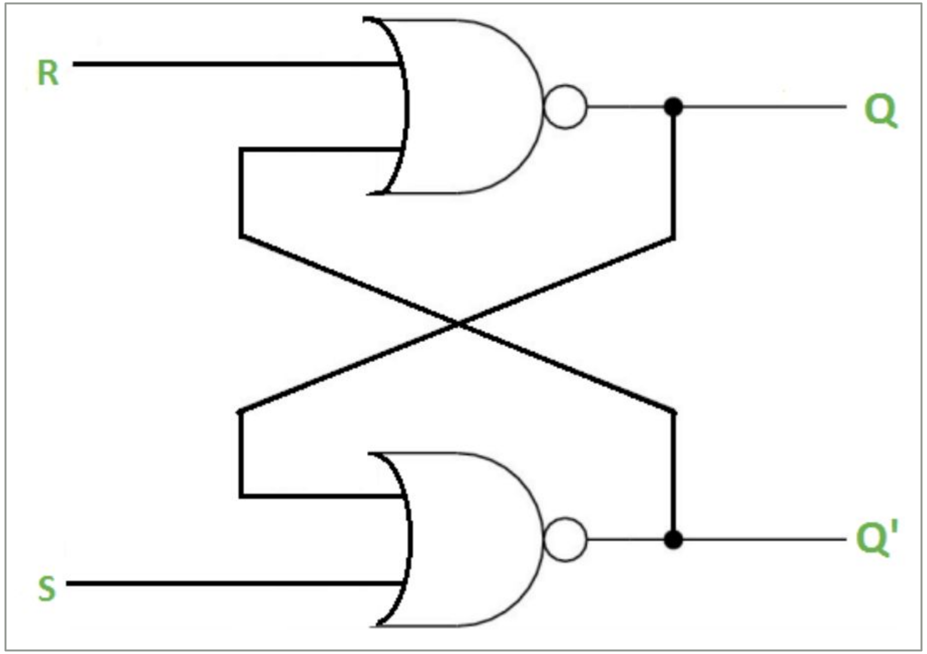
- **NO CLOCK === Asynchronous**
    - **Level-Sensitive Devices**
    - **Immediate changes to Output**



Great Visual Example of an SR Latch!!

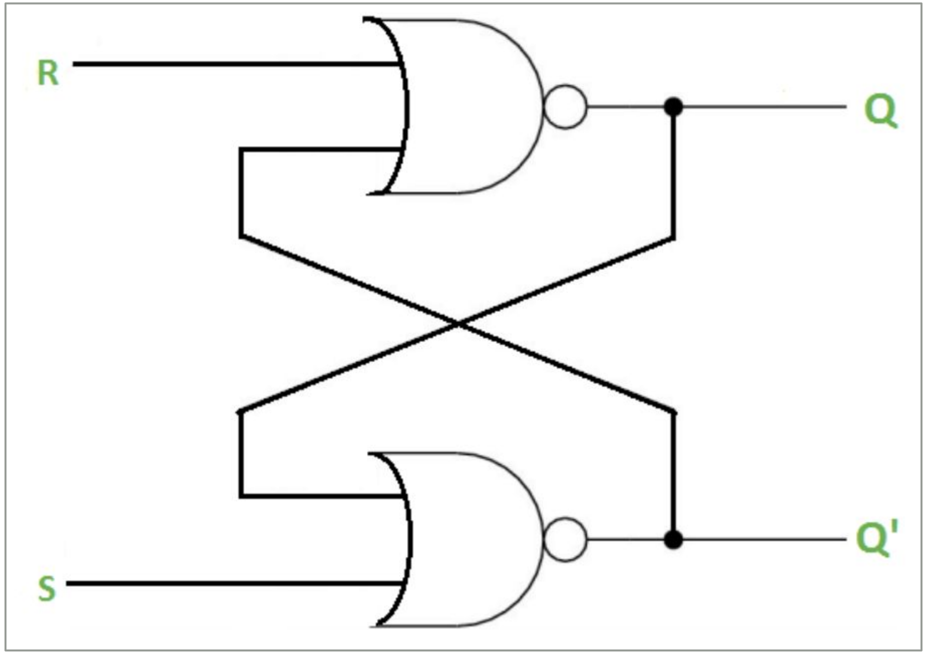
SRC

Viewing a Set-Reset (SR) Latch



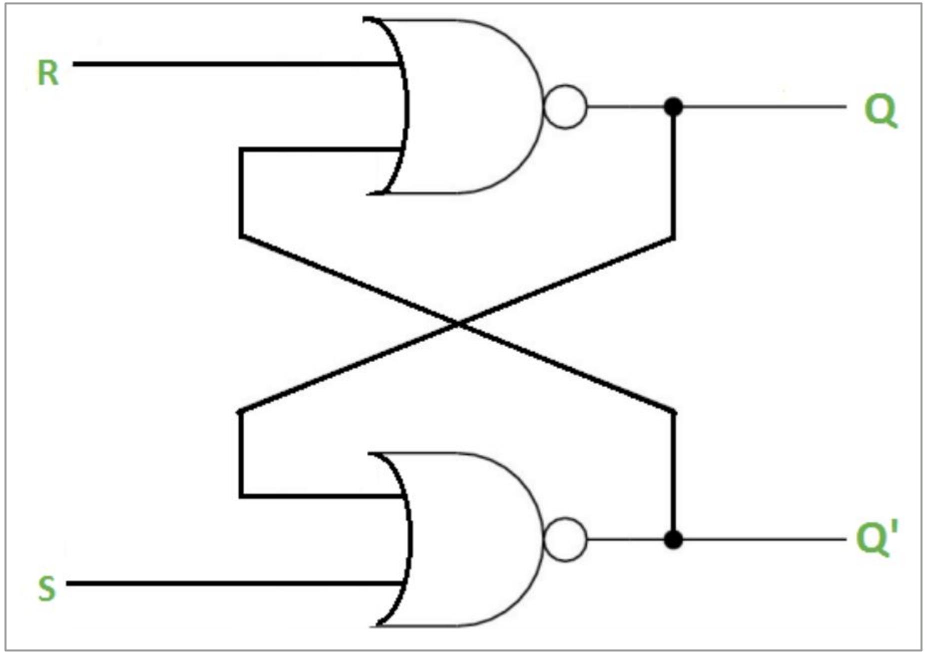
S	R	Q	Q’	Name
0	0			
0	1			
1	0			
1	1			

Viewing a Set-Reset (SR) Latch



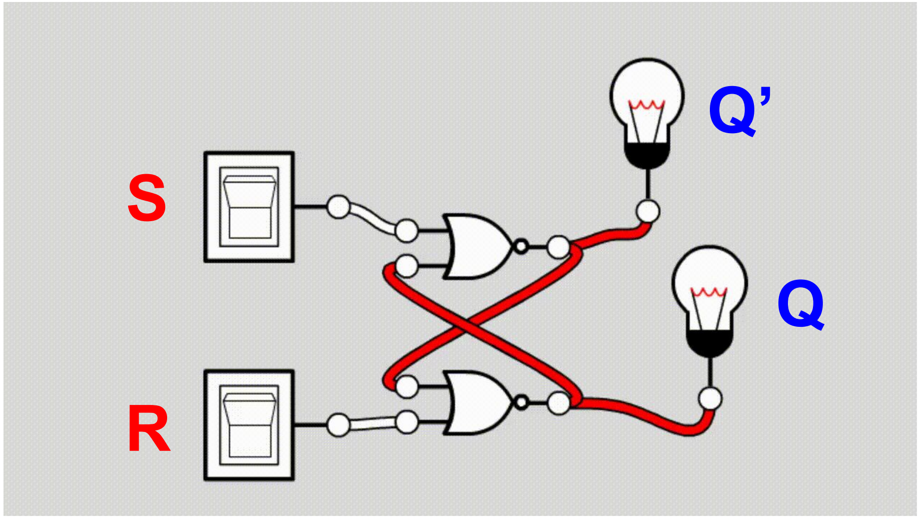
S	R	Q	Q’	Name
0	0			
0	1			
1	0	1		
1	1			

Viewing a Set-Reset (SR) Latch



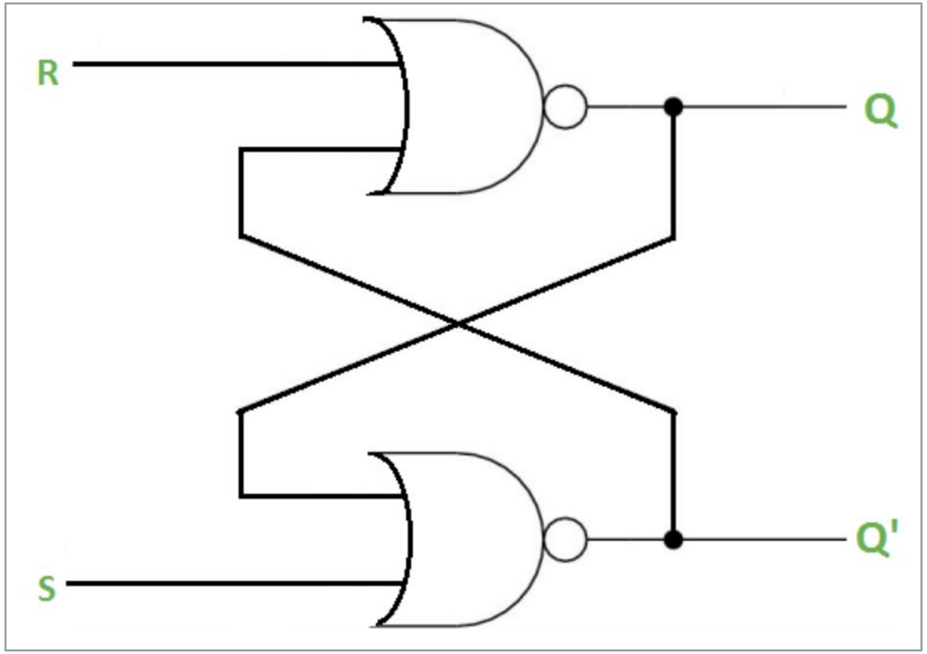
S	R	Q	Q'	Name
0	0			
0	1			
1	0	**1**	**0**	
1	1			

Viewing a Set-Reset (SR) Latch



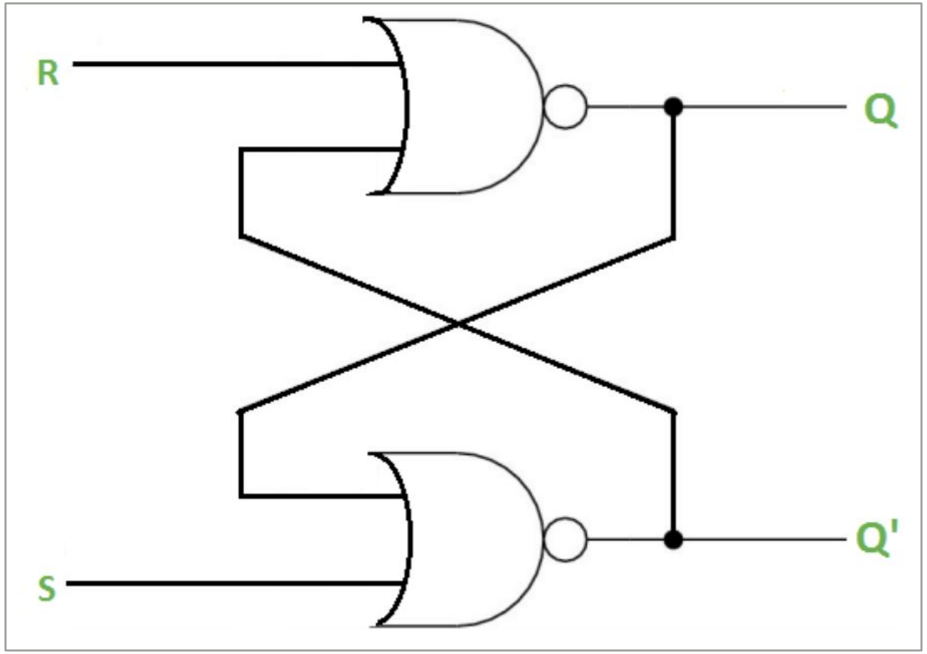
S	R	Q	Q’	Name
0	0			
0	1			
1	0	1	0	SET
1	1			

Viewing a Set-Reset (SR) Latch



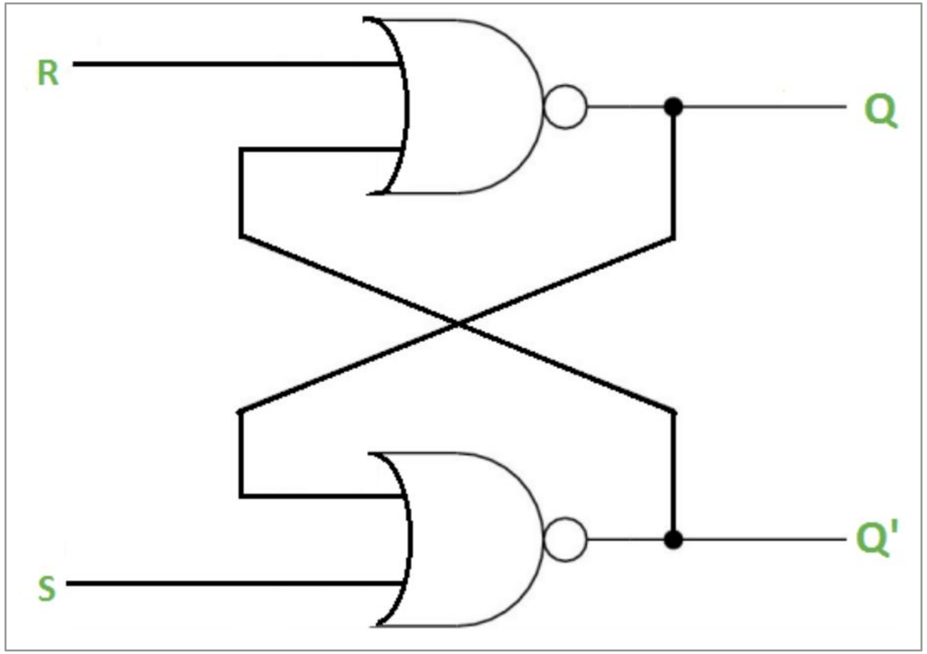
S	R	Q	Q’	Name
0	0			
0	1	**0**		
1	0	**1**	**0**	**SET**
1	1			

Viewing a Set-Reset (SR) Latch



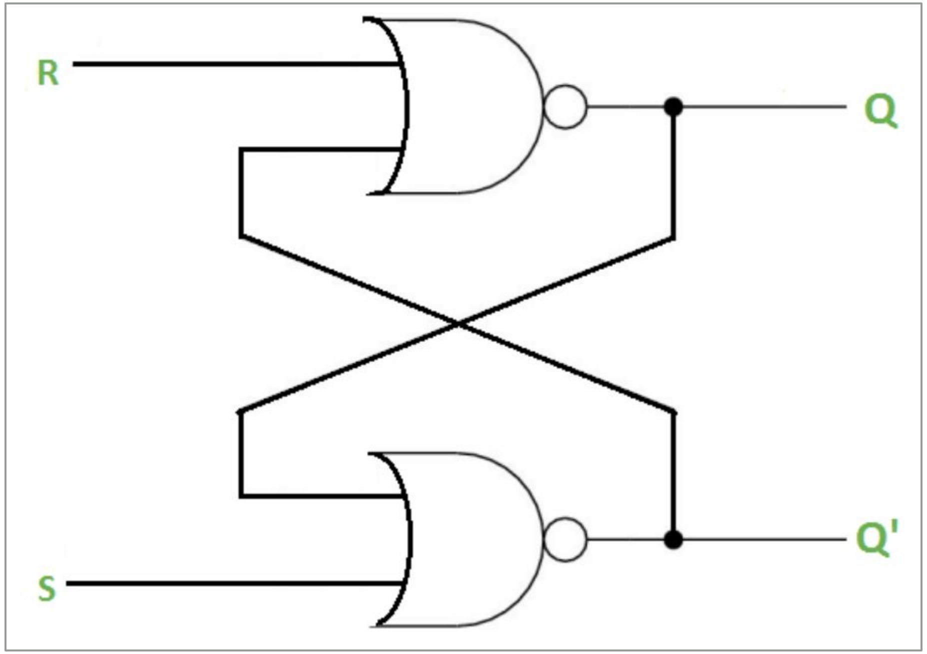
S	R	Q	Q’	Name
0	0			
0	1	**0**	**1**	
1	0	**1**	**0**	**SET**
1	1			

Viewing a Set-Reset (SR) Latch



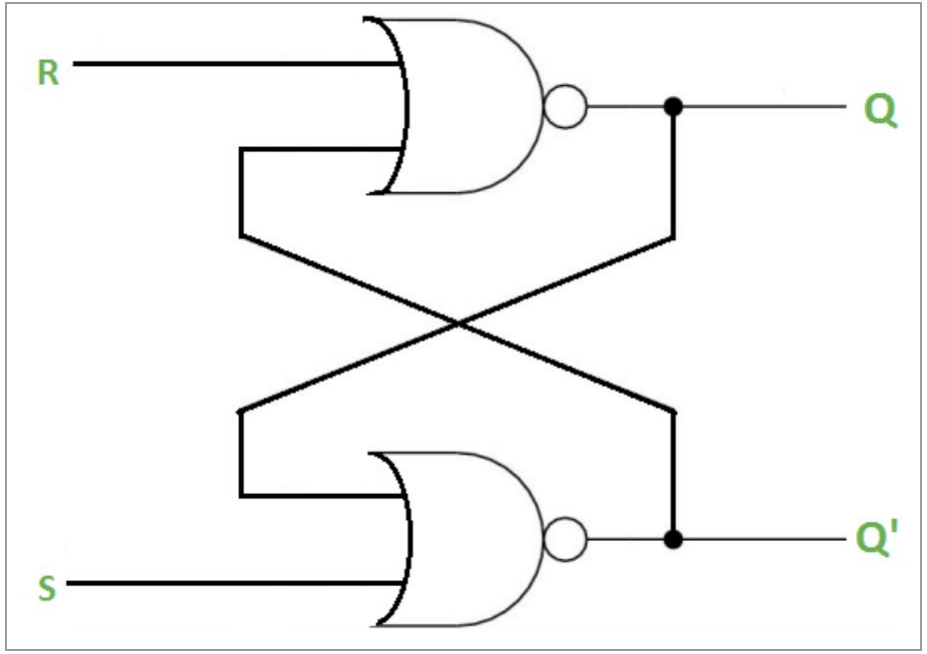
S	R	Q	Q’	Name
0	0			
0	1	**0**	**1**	**RESET**
1	0	**1**	**0**	**SET**
1	1			

Viewing a Set-Reset (SR) Latch



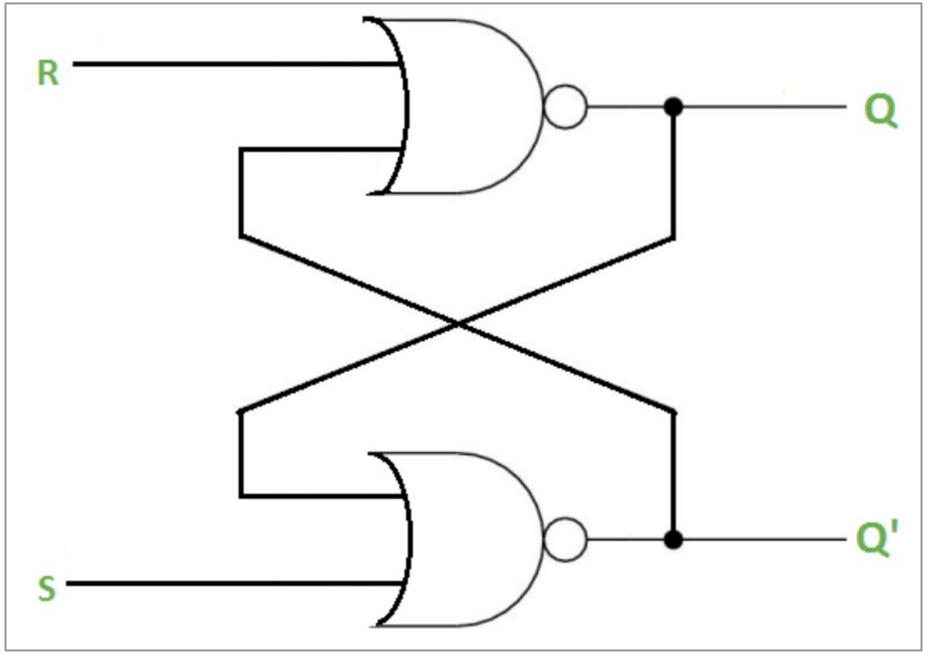
S	R	Q	Q’	Name
0	0	**Q**		
0	1	**0**	**1**	**RESET**
1	0	**1**	**0**	**SET**
1	1			

Viewing a Set-Reset (SR) Latch



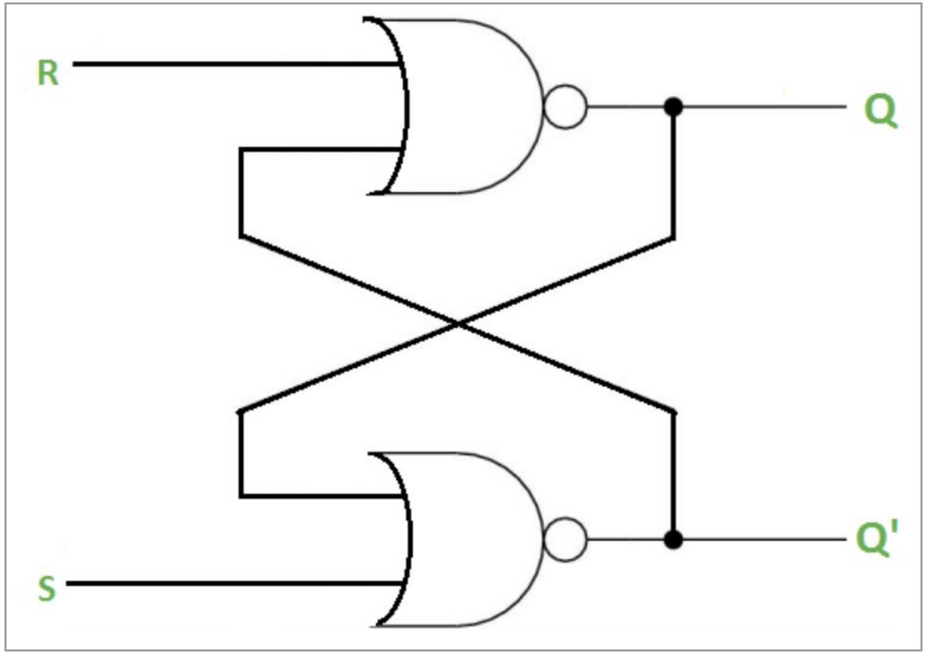
S	R	Q	Q’	Name
0	0	**Q**	**Q’**	
0	1	**0**	**1**	**RESET**
1	0	**1**	**0**	**SET**
1	1			

Viewing a Set-Reset (SR) Latch



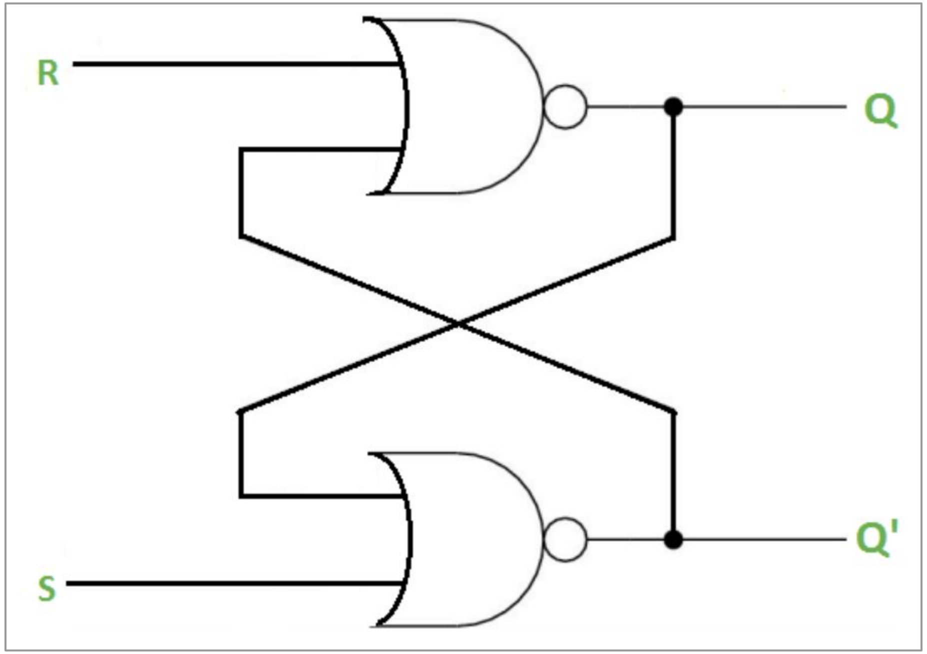
S	R	Q	Q’	Name
0	0	**Q**	**Q’**	**HOLD**
0	1	**0**	**1**	**RESET**
1	0	**1**	**0**	**SET**
1	1			

Viewing a Set-Reset (SR) Latch



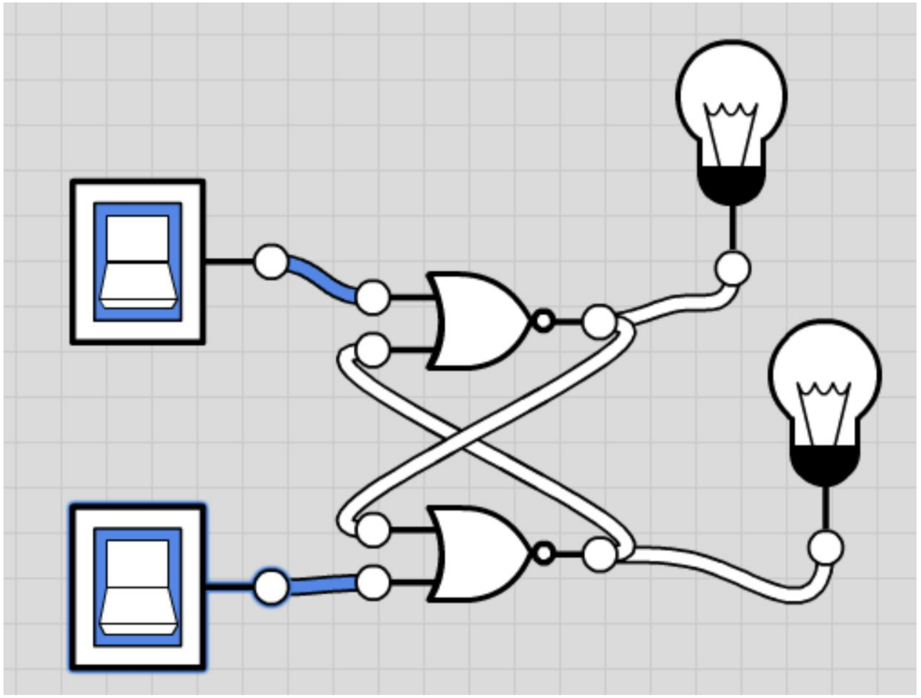
S	R	Q	Q’	Name
0	0	**Q**	**Q’**	**HOLD**
0	1	**0**	**1**	**RESET**
1	0	**1**	**0**	**SET**
1	1	**0**		

Viewing a Set-Reset (SR) Latch



S	R	Q	Q’	Name
0	0	**Q**	**Q’**	**HOLD**
0	1	**0**	**1**	**RESET**
1	0	**1**	**0**	**SET**
1	1	**0**	**0**	

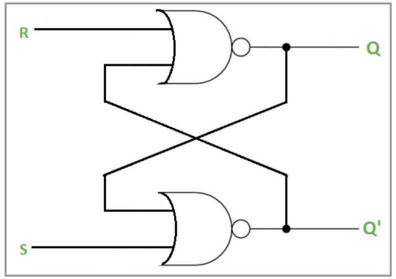
Viewing a Set-Reset (SR) Latch



S	R	Q	Q’	Name
0	0	Q	Q’	HOLD
0	1	0	1	RESET
1	0	1	0	SET
1	1	0	0	INVALID

Overall Notes about Set-Reset (SR) Latches

- Simplest form of a latch
  - 2-inputs: Set (S) and Reset (R)
  - + Can now hold a value! (When Set and Reset = 0)



- Forms the basic building blocks of all other types of flip-flops

- ISSUE: Has an “Invalid” state

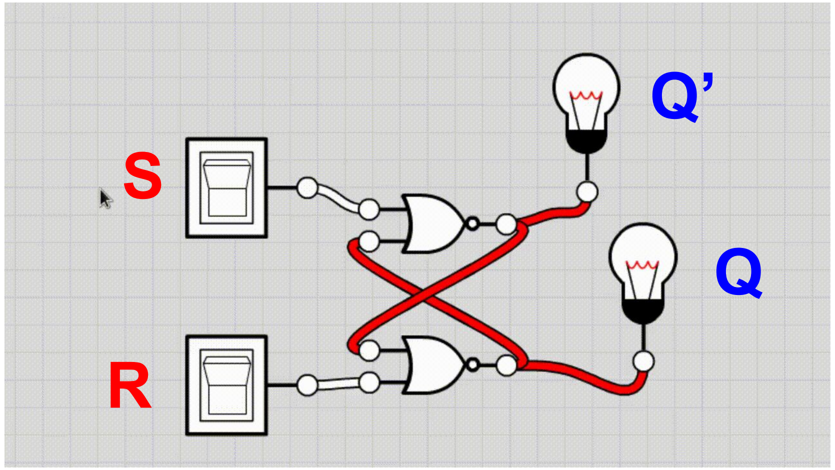
S	R	Q	Q’	Name
0	0	**Q**	**Q’**	**HOLD**
0	1	**0**	**1**	**RESET**
1	0	**1**	**0**	**SET**
1	1	**0**	**0**	**INVALID**

Bonus: Building an SR Latch in Verilog

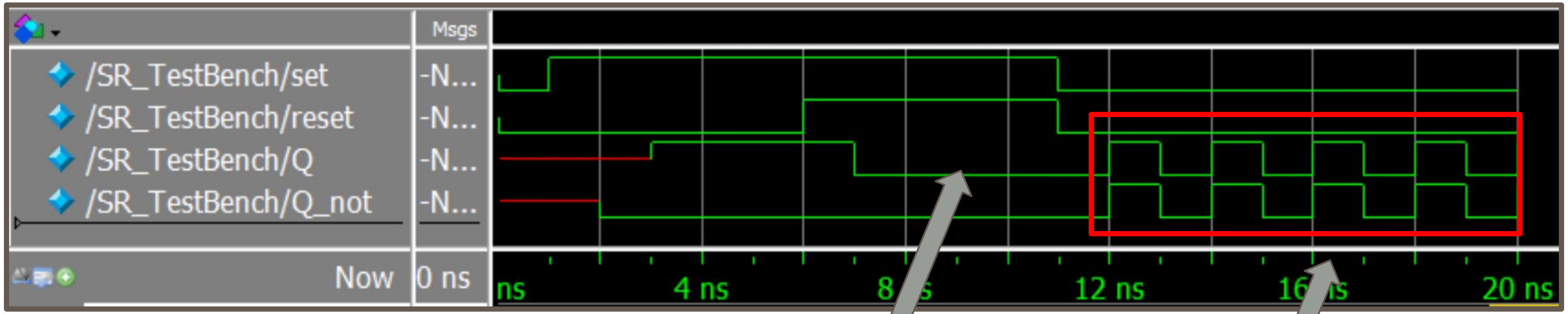
```
module SR_Latch_Assign (
        input set, reset,
        output Q, Q_not);

        assign Q = ~(reset | Q_not);
        assign Q_not = ~(set | Q);
endmodule
```

Level-Sensitive (Asynchronous)!

```
module SR_Latch_Procedural (
        input set, reset,
        output reg Q, Q_not);

        always @(*) begin
                Q = ~(reset | Q_not);
                Q_not = ~(set | Q);
        end
endmodule
```

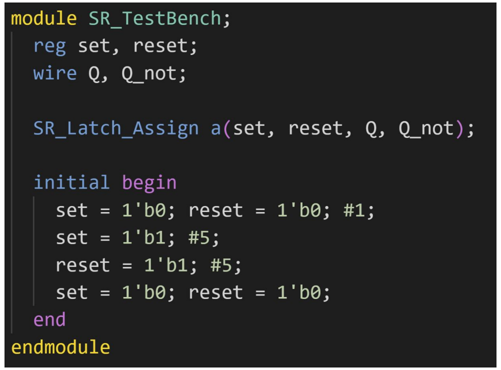


Bonus: Viewing SR Latch Invalid Behavior!

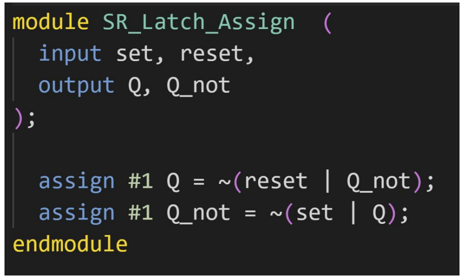


Invalid State
(S, R) = (0, 0)
→ (Q, Q') = (0,0)

Hold State
(S, R) = (0, 0)
→ (Q, Q') = OSCILLATION

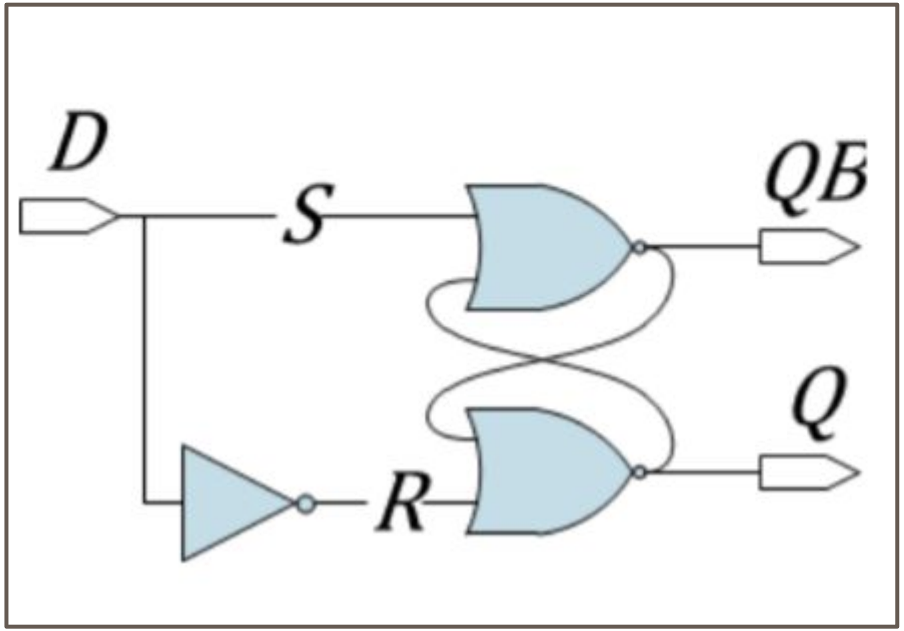


```
module SR_TestBench;
  reg set, reset;
  wire Q, Q_not;

  SR_Latch_Assign a(set, reset, Q, Q_not);

  initial begin
    set = 1'b0; reset = 1'b0; #1;
    set = 1'b1; #5;
    reset = 1'b1; #5;
    set = 1'b0; reset = 1'b0;
  end
endmodule
```



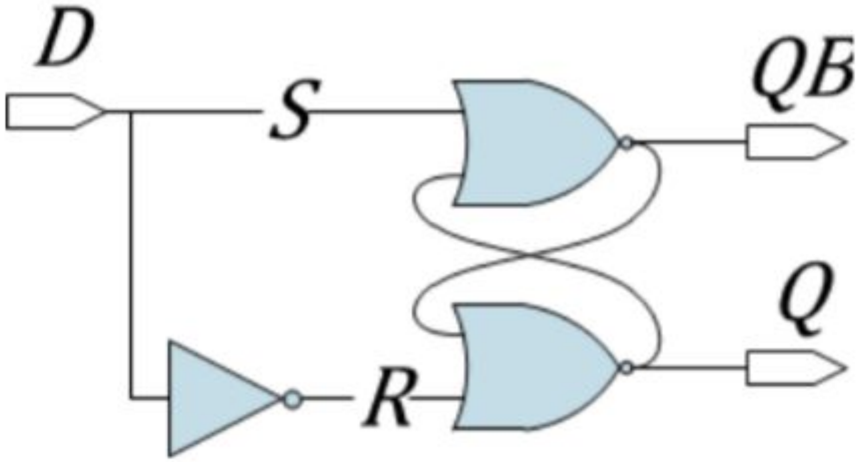
```
module SR_Latch_Assign  (
  input set, reset,
  output Q, Q_not
);

  assign #1 Q = ~(reset | Q_not);
  assign #1 Q_not = ~(set | Q);
endmodule
```

Moving on to Data (D) Latches



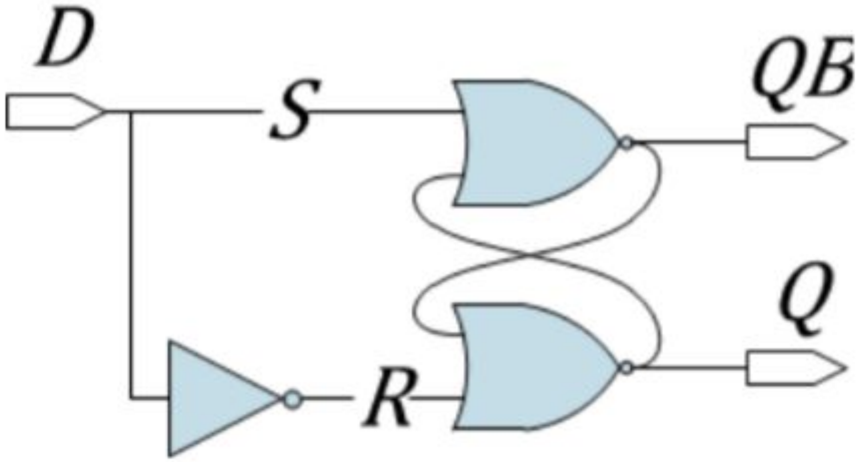
D	S	R	Q	Q’	Name
0					
1					

Moving on to Data (D) Latches



D	S	R	Q	Q’	Name
0	0	1	0	1	RESET
1					

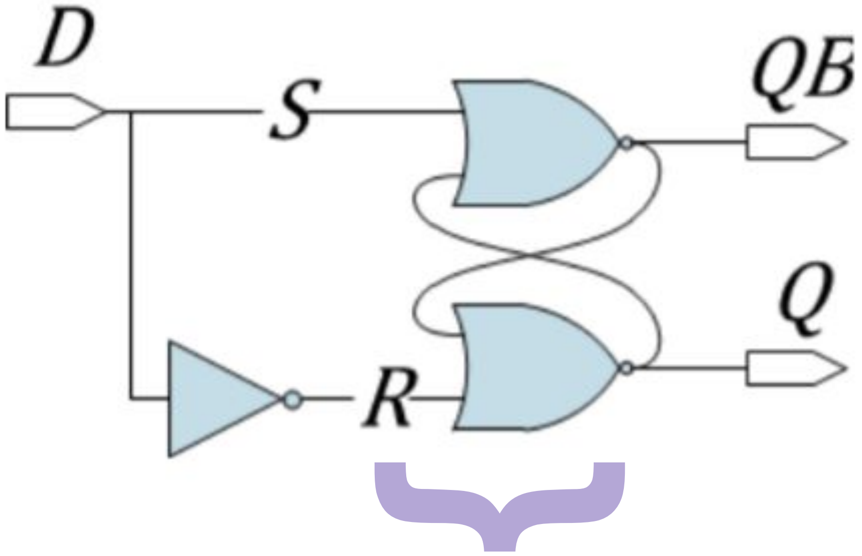
Moving on to Data (D) Latches



D	S	R	Q	Q’	Name
0	**0**	**1**	**0**	**1**	**RESET**
1	**1**	**0**	**1**	**0**	**SET**

What’s Good/Bad about D-Latches?

Moving on to Data (D) Latches

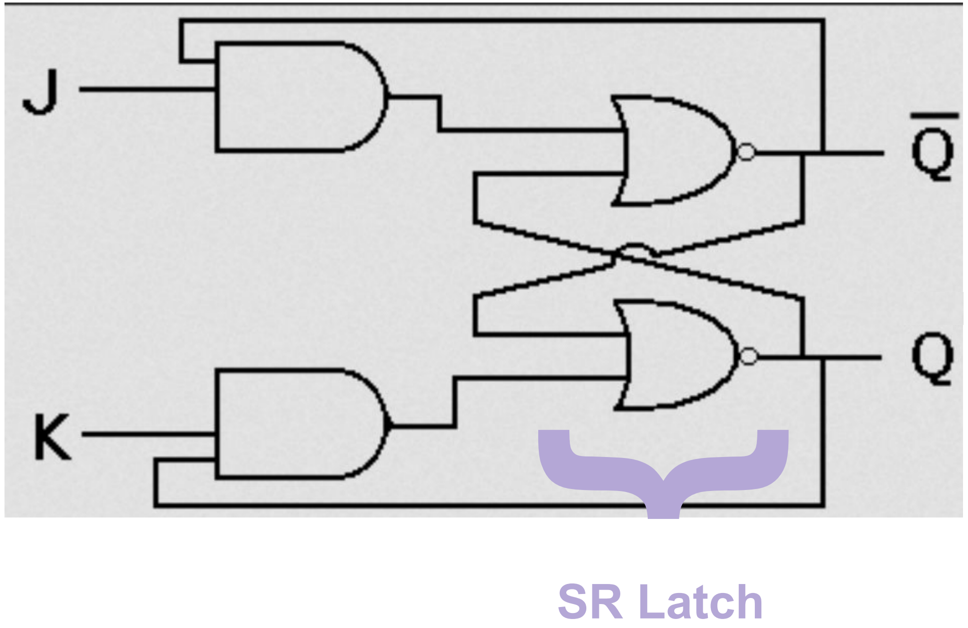


SR Latch

D	S	R	Q	Q’	Name
0	**0**	**1**	**0**	**1**	**RESET**
1	**1**	**0**	**1**	**0**	**SET**

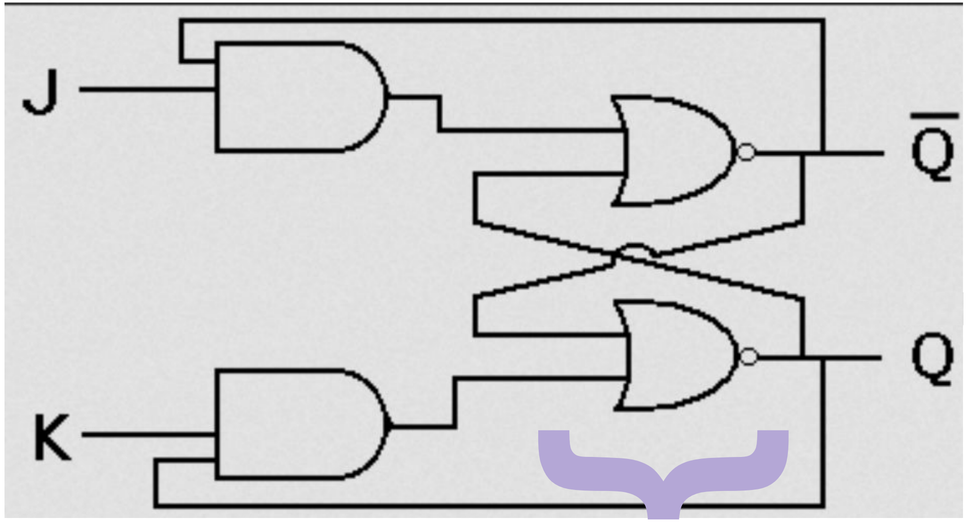
- + No more Invalid State :D
- - BUT NO MORE HOLD STATE :O
- - → Has no state (Not sequential)

BONUS: JK-Latches (There are many different latches)



J	K	Q	Q’	Name
0	0			
0	1			
1	0			
1	1			

BONUS: JK-Latches (There are many different latches)

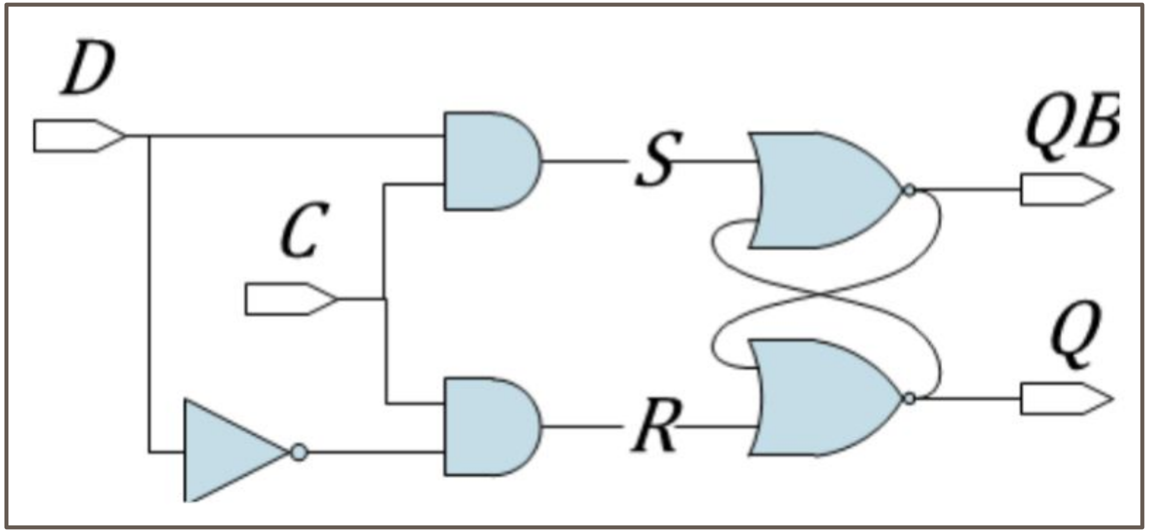


SR Latch

J	K	Q	Q’	Name
0	0	Q	Q’	HOLD
0	1	0	1	RESET
1	0	1	0	SET
1	1	Q’	Q	Toggle

(Positive) Enabled Level-Sensitive D-Latch

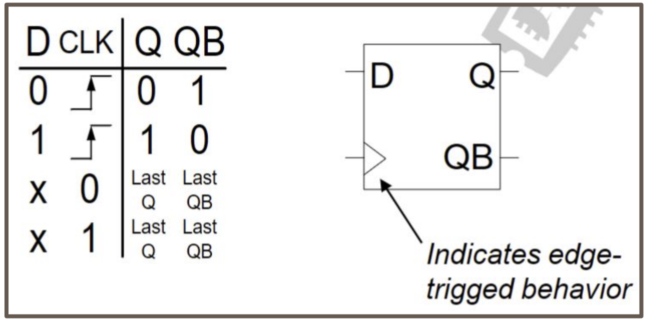
- Use of a Enable ("Clock") (BUT STILL NOT SYNCHRONOUS)
- Whenever C is high → Q = D (Level-Sensitive to D) (Set/Reset)
- Whenever C is low → Q holds (Hold)



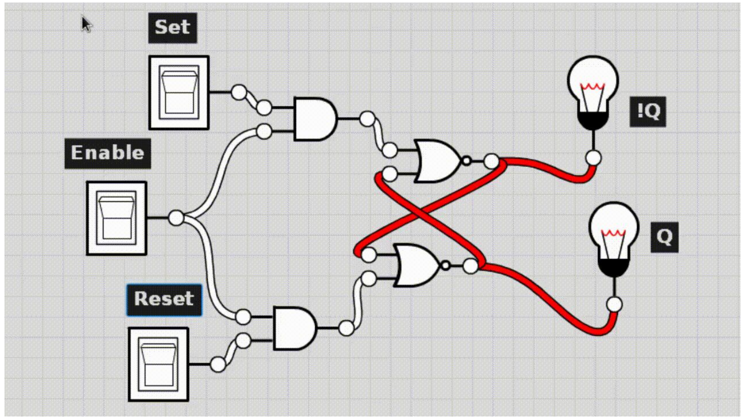
C	D	S	R	Q	QB	Name
0	0	0	0	Q	QB	HOLD
0	1	0	0	Q	QB	HOLD
1	0	0	1	0	1	RESET
1	1	1	0	1	0	SET

Asynchronous vs. Synchronous in Logic Design

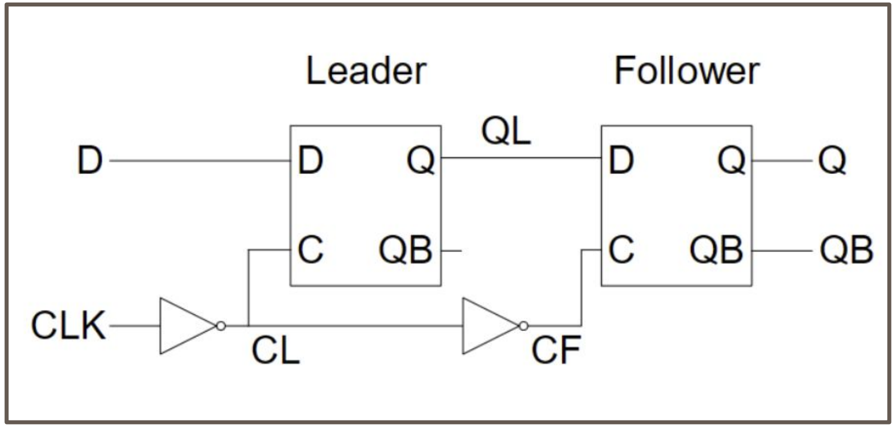
- Asynchronous Circuits
  - Circuit operates without a clock signal
- Synchronous Circuits
  - Samples inputs only once per clock cycle



D	CLK	Q	QB
0	↑	0	1
1	↑	1	0
x	0	Last Q	Last QB
x	1	Last Q	Last QB



Level-Sensitive Asynchronous D-Latch



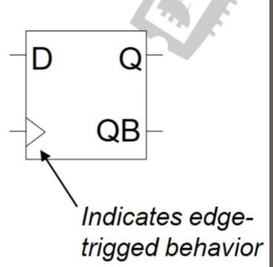
Edge-Triggered Synchronous D Flip-Flop

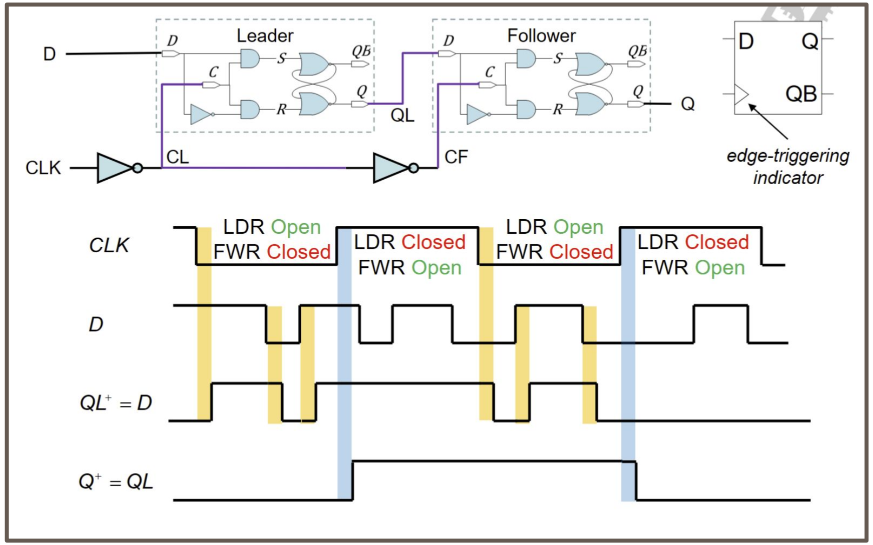
(Positive) Edge-Triggered D Flip-Flop Timing

- When CLK is Low → **Leader is open & Follower is closed**
  - **D will be able to affect QL**
  - **QL WILL NOT be able to affect Q**
- When CLK is High → **Leader is closed & Follower is open**
  - **D WILL NOT be able to affect QL**
  - **QL can now affect Q**

+ **Clocks D on positive edge of CLK**

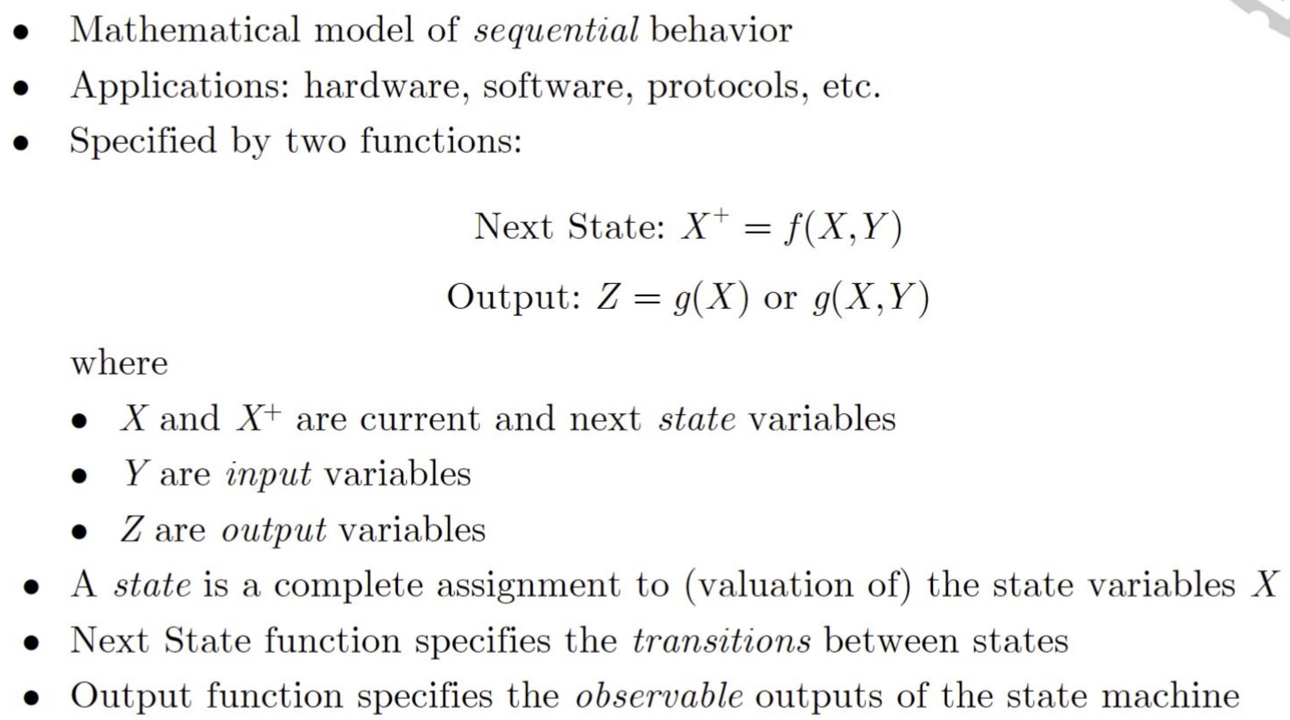
D	CLK	Q	QB
0	↑	0	1
1	↑	1	0
x	0	Last Q	Last QB
x	1	Last Q	Last QB





Designing Sequential Circuits (Finite State Machines)

Finite State Machines Overview



- Mathematical model of *sequential* behavior
- Applications: hardware, software, protocols, etc.
- Specified by two functions:

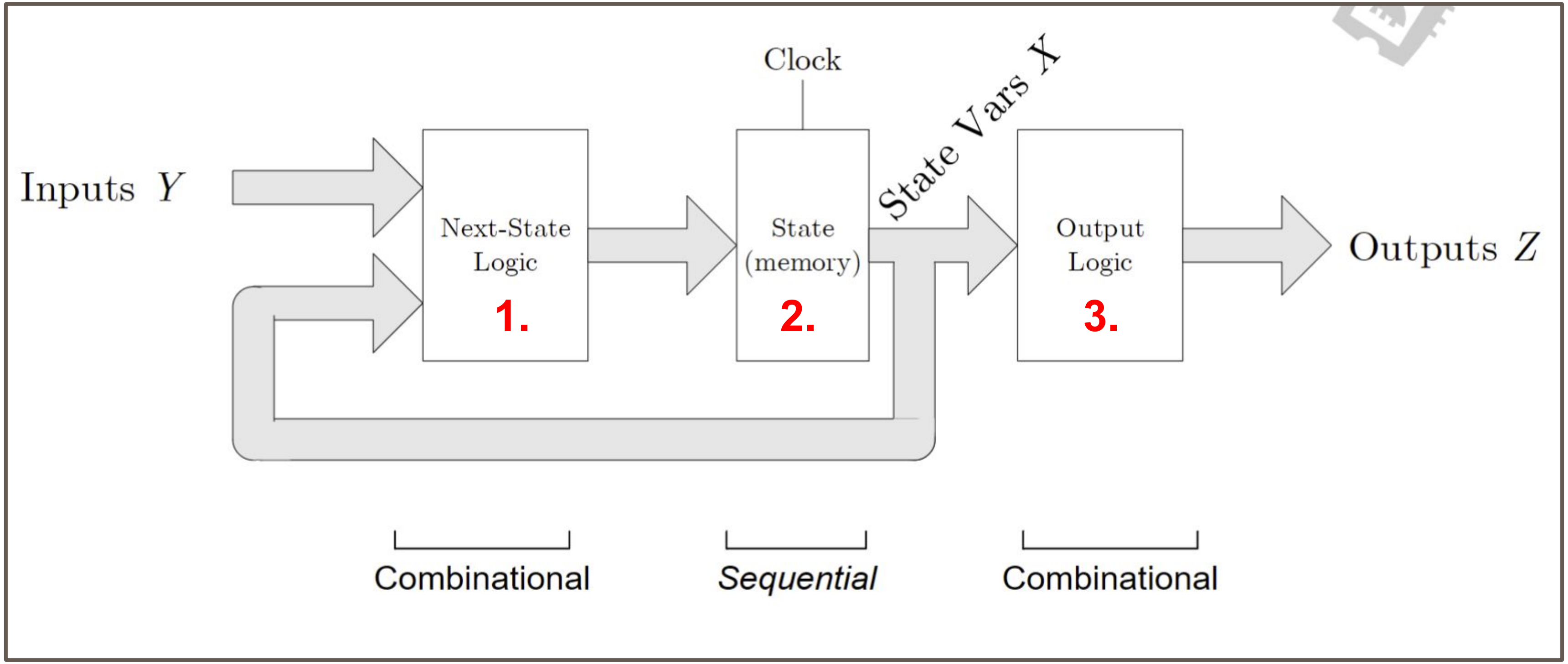
$$\text{Next State: } X^{+} = f(X, Y)$$

$$\text{Output: } Z = g(X) \text{ or } g(X, Y)$$

  where

  - X and X^{+} are current and next *state* variables
  - Y are *input* variables
  - Z are *output* variables
- A *state* is a complete assignment to (valuation of) the state variables X
- Next State function specifies the *transitions* between states
- Output function specifies the *observable* outputs of the state machine

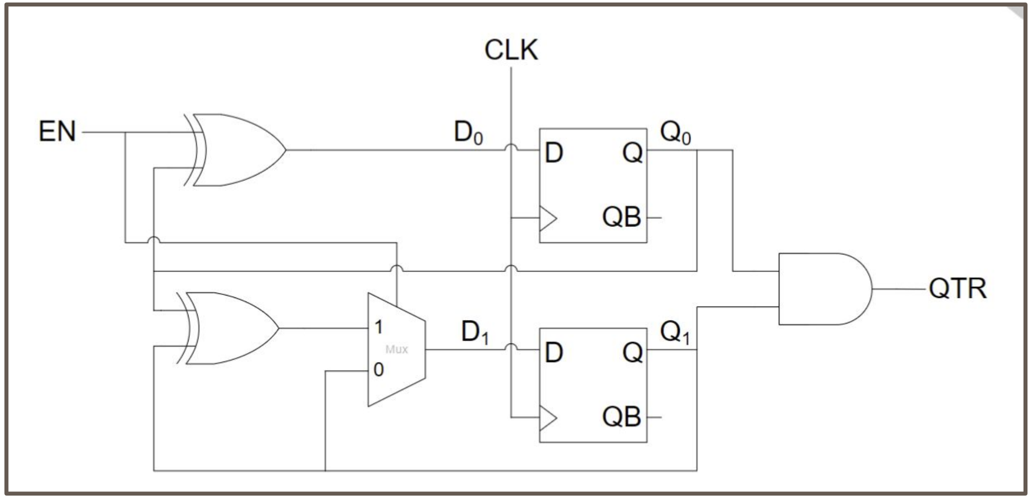
The 3 Main Components of Sequential Circuits



Component 1. Next State Logic (Combinational)

- Combinationally determining the **next state = f(current state, current inputs)**
- Generating excitation equation (Equations for next state D)

- Create excitation equations for the circuit below

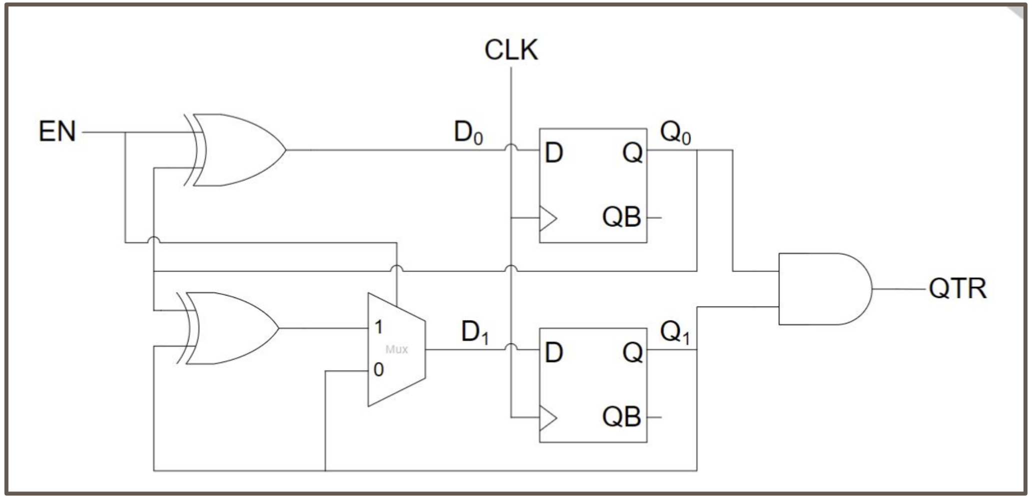


Component 1. Next State Logic (Combinational)

- **Combinationally** determining the next state = f(current state, current inputs)
- Generating excitation equation (Equations for next state D)

D_0 = EN XOR Q_0

D_1 = EN * (Q_0 XOR Q_1) + (EN' * Q_1)

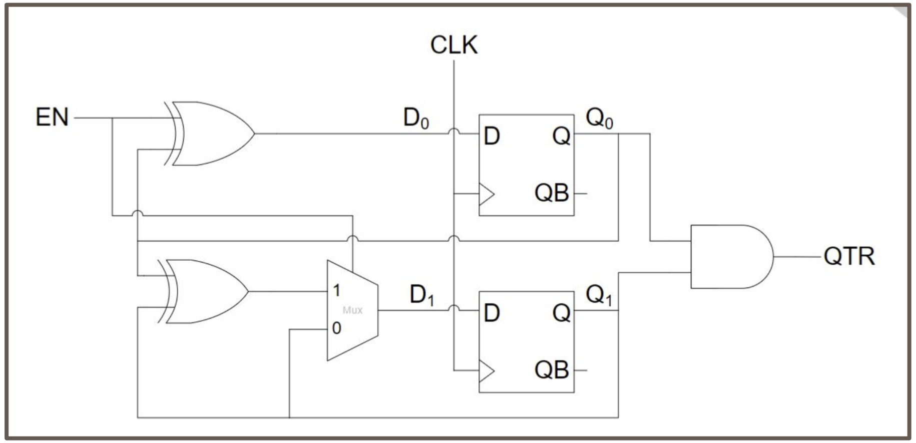


Component 2. Memory Logic (Sequential)

- **Sequentially** storing the new state (Generating Transition Equations)
- **NOTE:** This logic should be very minimal
    - In our designs in 270, we treat transition equations = excitation equations

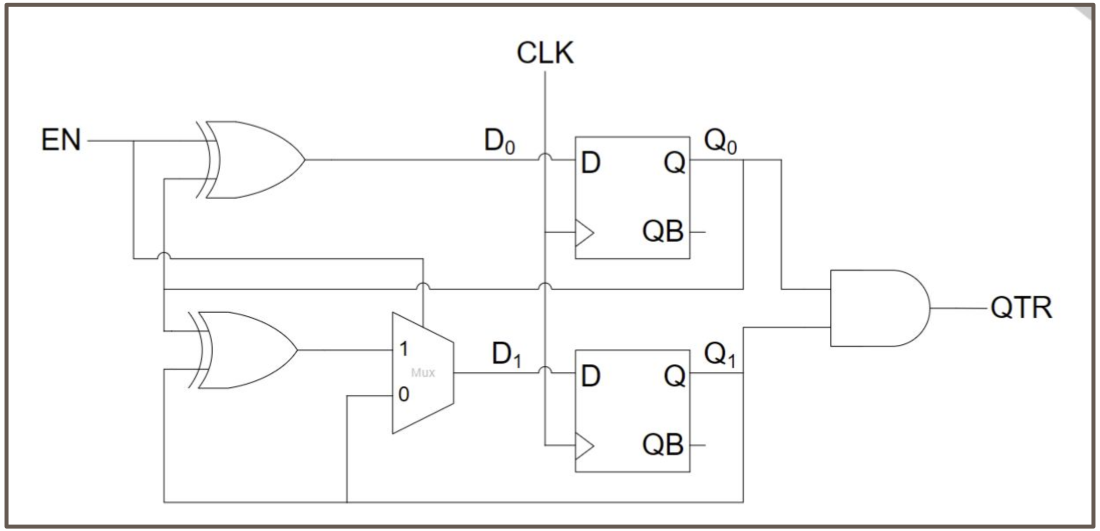
$Q_0^+ <= D_0 = EN \text{ XOR } Q_0$

$Q_1^+ <= D_1 = EN * (Q_0 \text{ XOR } Q_1) + (EN' * Q_1)$



Component 3. Output Logic (Combinational)

- **Combinationally** determining the outputs (Generating Output Equations)
- **Is our circuit a moore or a mealy machine?**



Moore vs. Mealy Machine??

- What do we mean when we call something a Moore/Mealy Machine?

Moore vs. Mealy Machine??

- Defining how the outputs are determined in our machine
- What’s the difference between a Moore and Mealy machine?

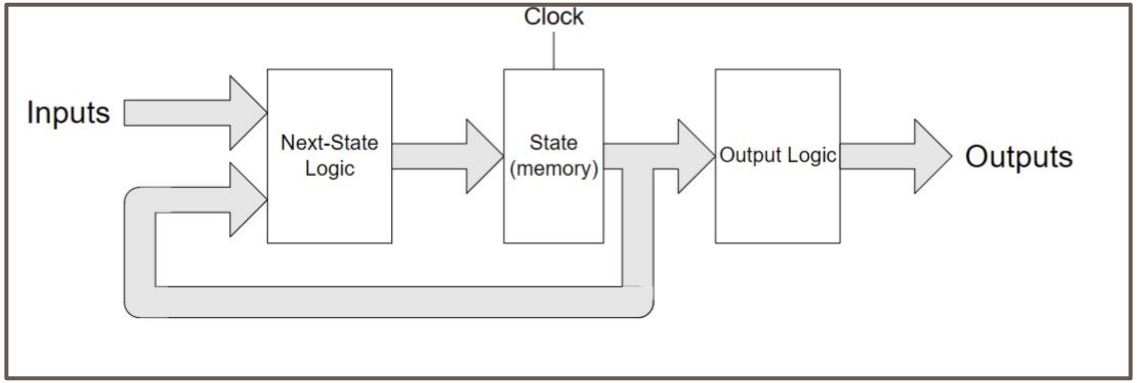
Moore vs. Mealy Machine??

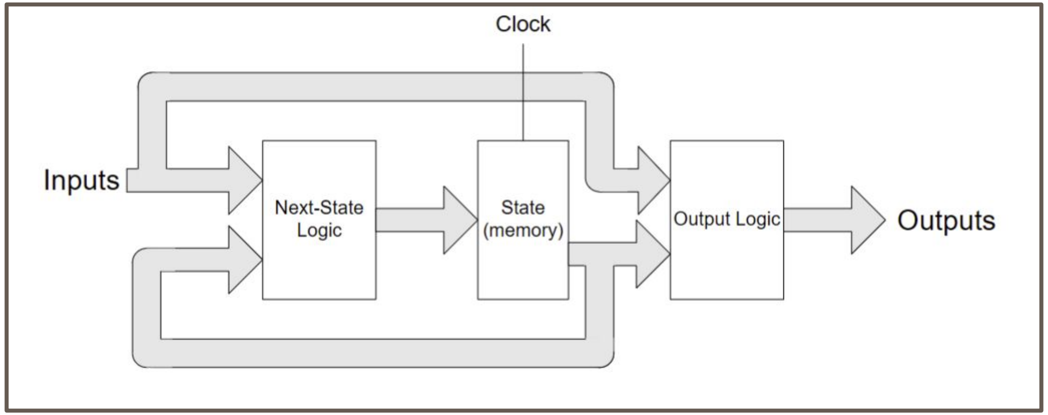
- Defining how the outputs are determined in our machine

- Moore Machine - Outputs can be determine from just the current state of our system

- Mealy Machine - Outputs are determined by both the current state **and current combination of inputs into the system**

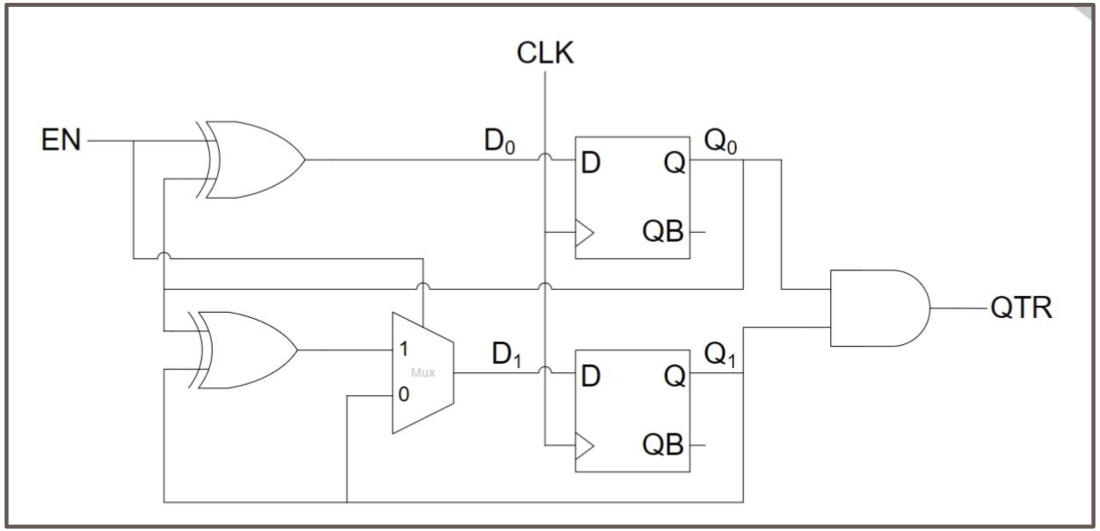
Moore vs. Mealy Machine??





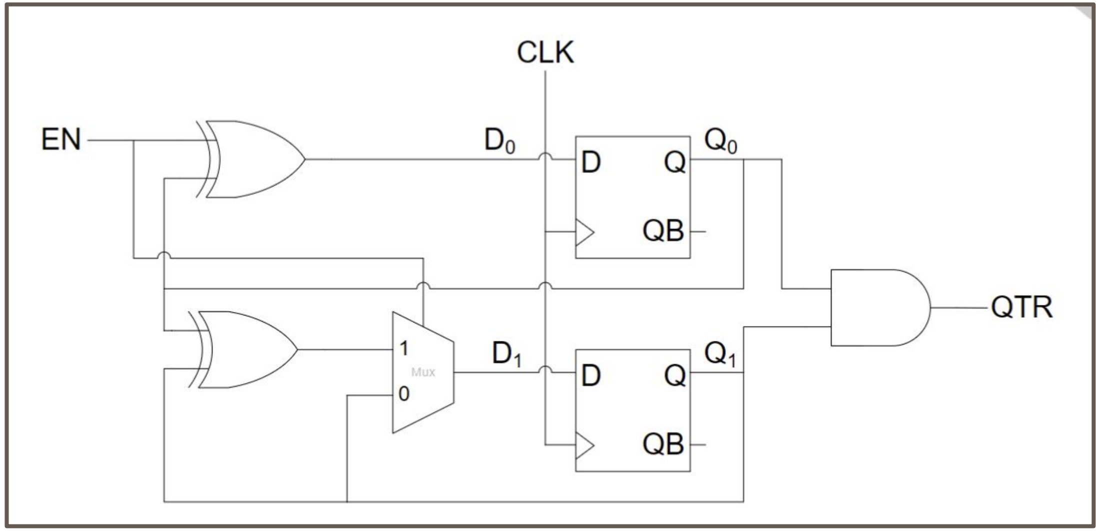
Component 3. Output Logic (Combinational)

- **Combinationally** determining the outputs (Generating Output Equations)
- **Is our circuit a moore or a mealy machine?**



Component 3. Output Logic (Combinational)

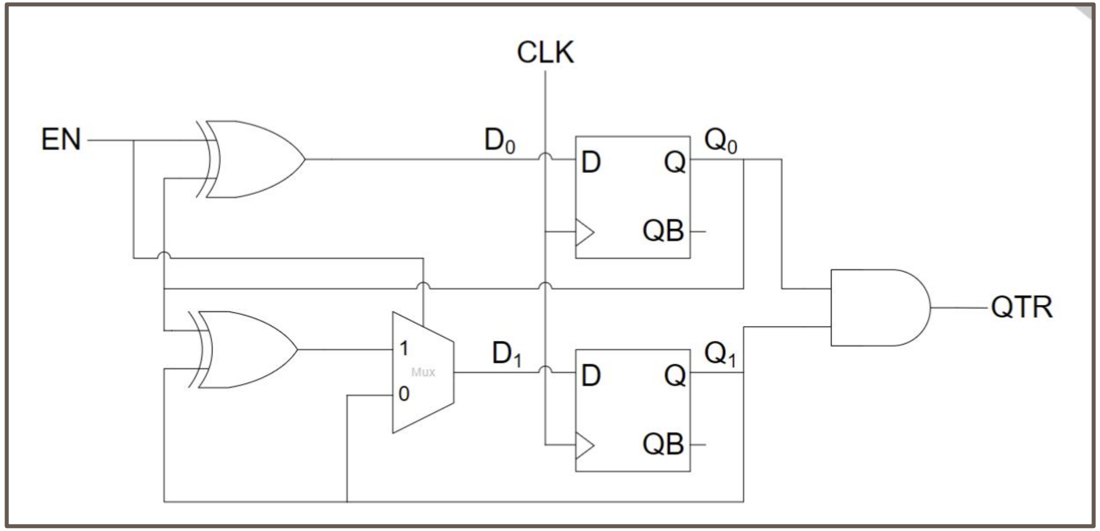
- **Combinationally** determining the outputs (Generating Output Equations)
- **Moore Machine - Output QTR only dependent on current state Q**



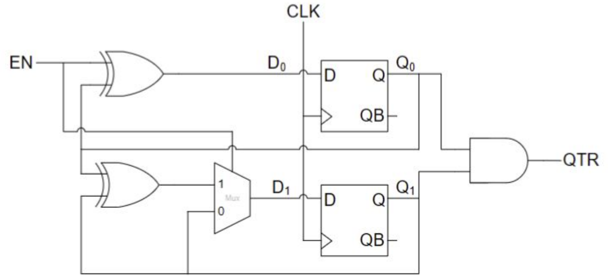
Component 3. Output Logic (Combinational)

- **Combinationally** determining the outputs (Generating Output Equations)
- **Moore Machine - Output QTR only dependent on current state Q**

$QTR = Q_0 \& Q_1$



Putting it all together (From Circuit → Tables/Equations)



Transition Equations:

$$Q_0^+ = D_0 = EN \oplus Q_0$$

$$Q_1^+ = D_1 = EN \cdot (Q_0 \oplus Q_1) + \overline{EN} \cdot Q_1$$

Output Equation:

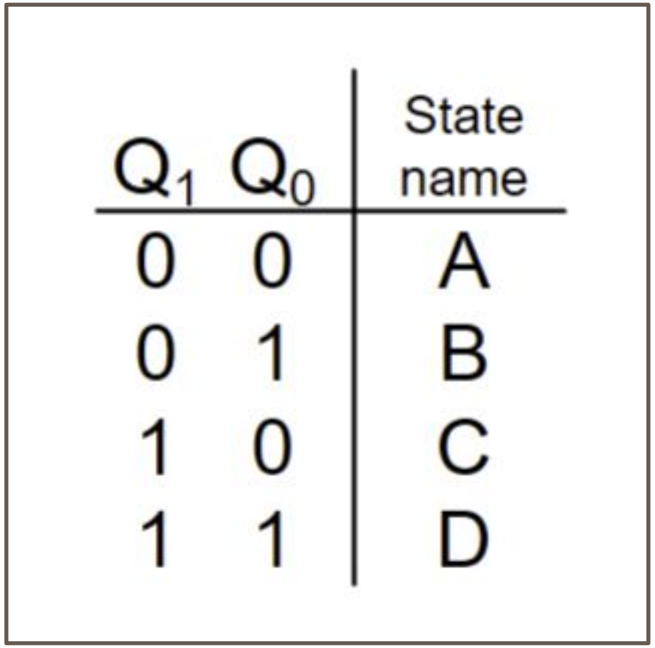
$$QTR = Q_0 \cdot Q_1$$

Transition/Output Table:

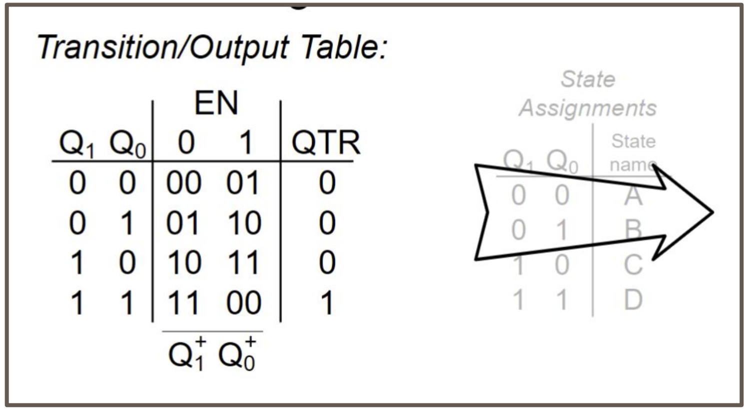
current state		*input* EN		*output*
Q_1	Q_0	0	1	QTR
0	0	00	01	0
0	1	01	10	0
1	0	10	11	0
1	1	11	00	1
		$Q_1^+ Q_0^+$ *next state*		

State Assignment

Creating a one-to-one mapping from state encoding to state names



Q_1	Q_0	State name
0	0	A
0	1	B
1	0	C
1	1	D



Transition/Output Table:

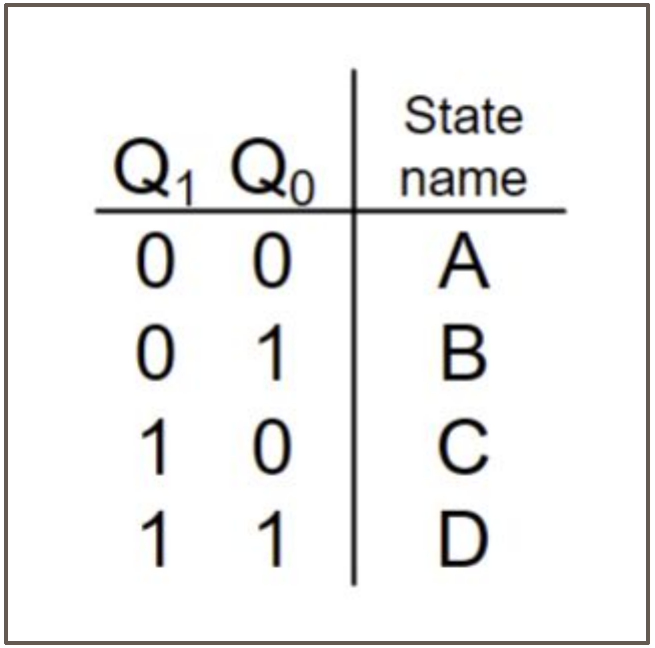
		EN		
Q_1	Q_0	0	1	QTR
0	0	00	01	0
0	1	01	10	0
1	0	10	11	0
1	1	11	00	1
		$Q_1^+ Q_0^+$		

State Assignments

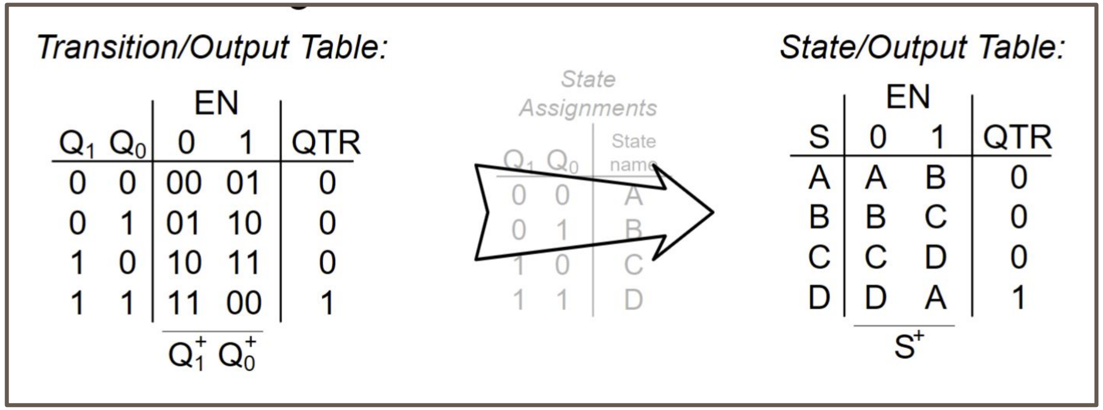
Q_1	Q_0	State name
0	0	A
0	1	B
1	0	C
1	1	D

?

Moving from V1 → V2 (State Labelling)



Q_1	Q_0	State name
0	0	A
0	1	B
1	0	C
1	1	D



Transition/Output Table:

		EN		
Q_1	Q_0	0	1	QTR
0	0	00	01	0
0	1	01	10	0
1	0	10	11	0
1	1	11	00	1
		$Q_1^+ Q_0^+$		

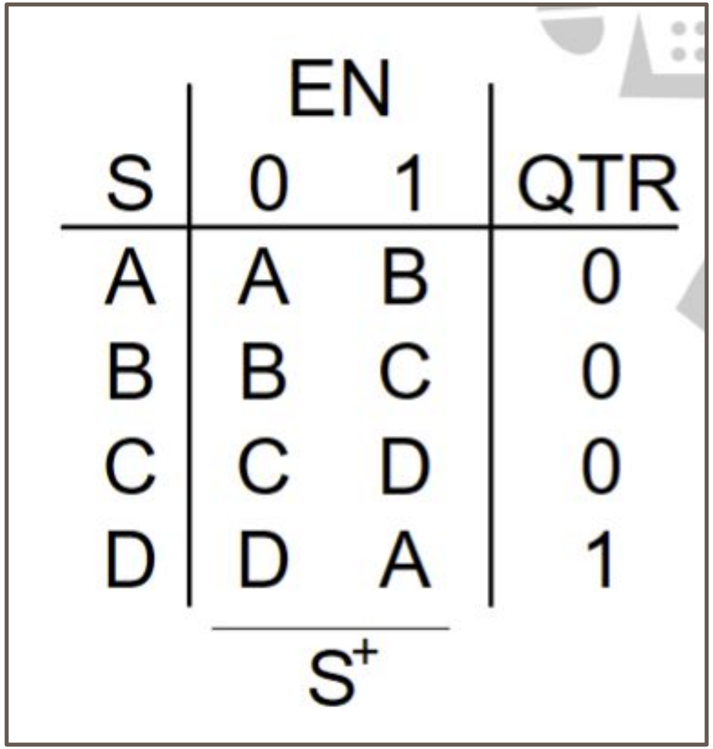
State Assignments

Q_1	Q_0	State name
0	0	A
0	1	B
1	0	C
1	1	D

State/Output Table:

	EN		
S	0	1	QTR
A	A	B	0
B	B	C	0
C	C	D	0
D	D	A	1
	S^+		

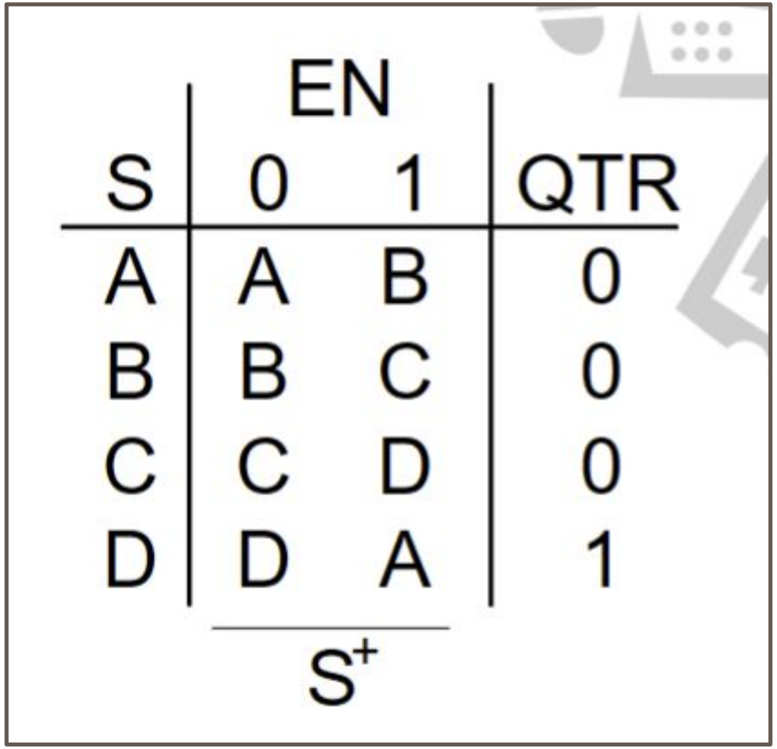
Creating State Diagram



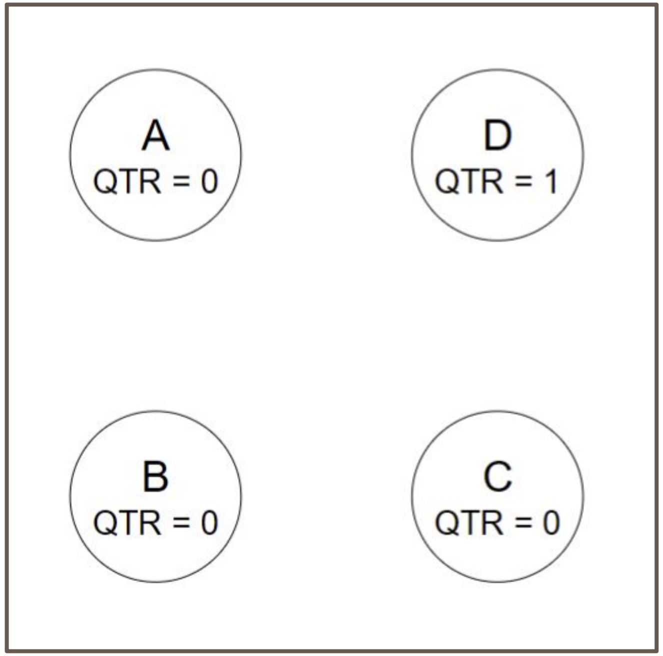
	EN		
S	0	1	QTR
A	A	B	0
B	B	C	0
C	C	D	0
D	D	A	1
	S^+		

What would the state diagram look like?

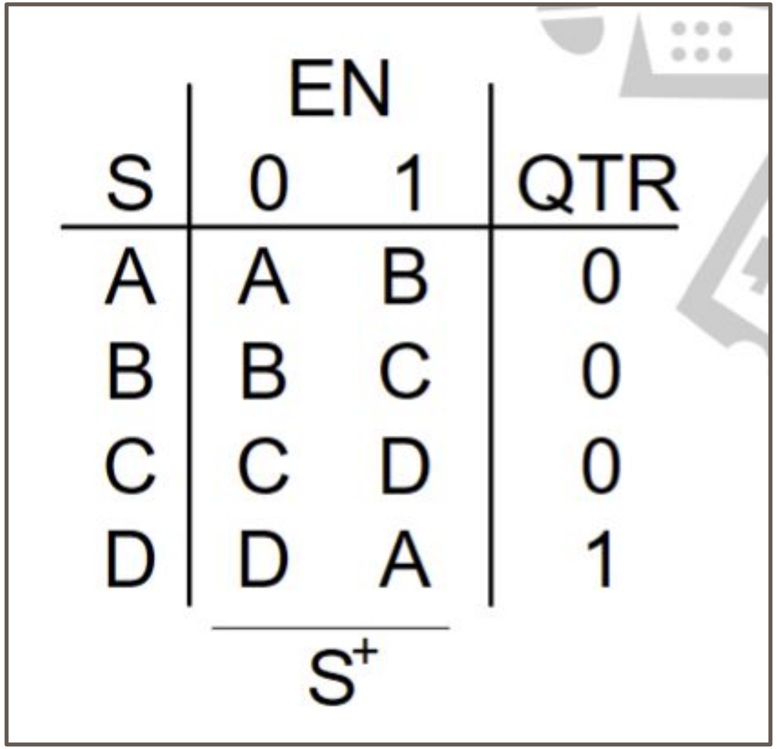
Creating State Diagram



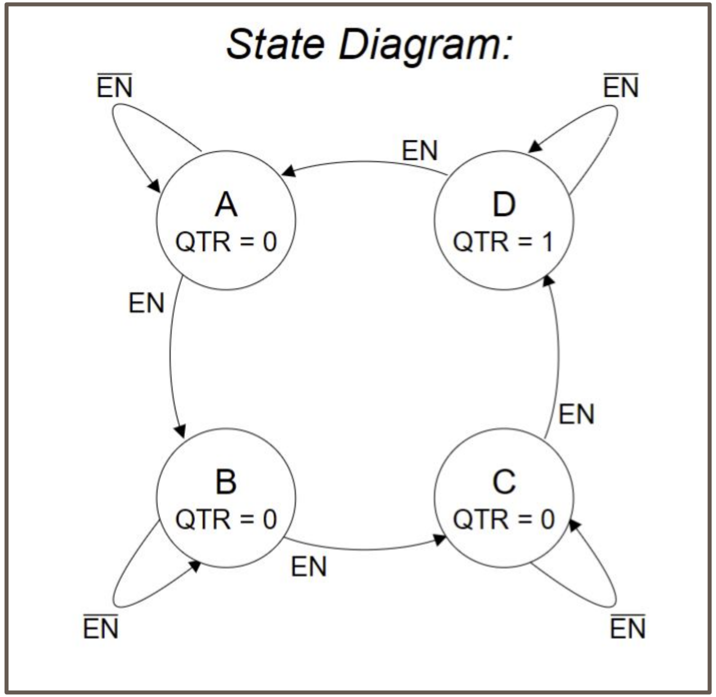
	EN		
S	0	1	QTR
A	A	B	0
B	B	C	0
C	C	D	0
D	D	A	1
	S^+		



Creating State Diagram



	EN		
S	0	1	QTR
A	A	B	0
B	B	C	0
C	C	D	0
D	D	A	1
	S^+		



Common Sequential Building Blocks

Common Sequential Building Blocks Overview

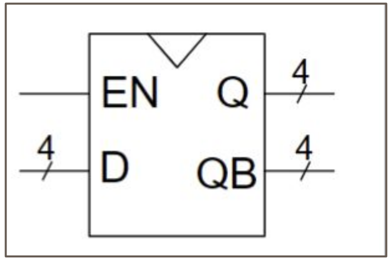
- Registers
- Counters
  - Ripple Counter
  - Parallel Count
- Shift Registers
  - Parallel Shift Registers
  - Universal Shift Registers
- Shift Register Counters
  - Ring Counter
  - Johnson Counter
  - Linear Feedback Shift Registers

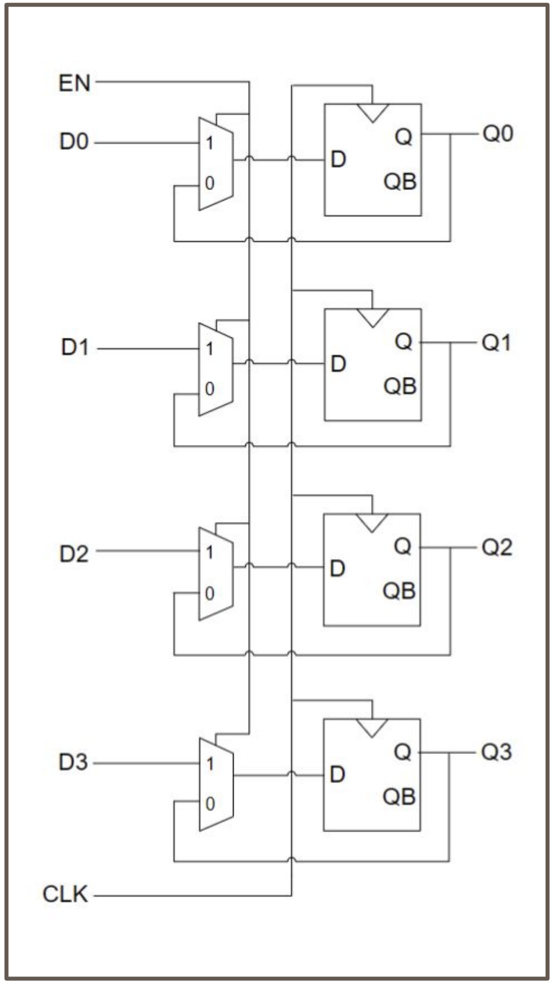
Registers

- Collection of D Flip-Flops
- All share a common clock

- **Used to store collection of bits**

```
always @(posedge clk) begin
   If (EN) Q <= D
   Else    Q <= Q
end
```





Modulo Counter

- No real physical counters are infinite
- Typically count through a series of finite states, then repeats

- Modulus - Number of states in counter’s sequence

Ex. Counter Sequence = {1, 2, 3, 4, 5, 6, 1, …} →

Ex. Counter Sequence = {5, 7, 2, 1, 6, 4, 5, …} →

Modulo Counter

- No real physical counters are infinite
- Typically count through a series of finite states, then repeats

- Modulus - Number of states in counter's sequence

Ex. Counter Sequence = {1, 2, 3, 4, 5, 6, 1, …} → **Modulus = 6**

Ex. Counter Sequence = {5, 7, 2, 1, 6, 4, 5, …} →

Modulo Counter

- No real physical counters are infinite
- Typically count through a series of finite states, then repeats

- Modulus - Number of states in counter’s sequence

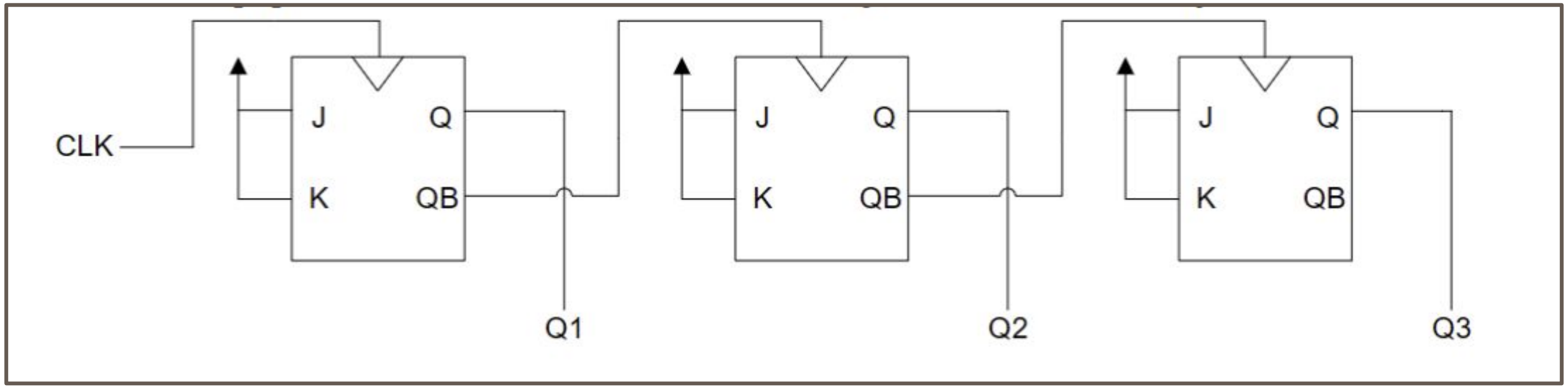
Ex. Counter Sequence = {1, 2, 3, 4, 5, 6, 1, …} → **Modulus = 6**

Ex. Counter Sequence = {5, 7, 2, 1, 6, 4, 5, …} → **Modulus = 6!!**

BOTH HAVE THE SAME AMOUNT OF UNIQUE STATES

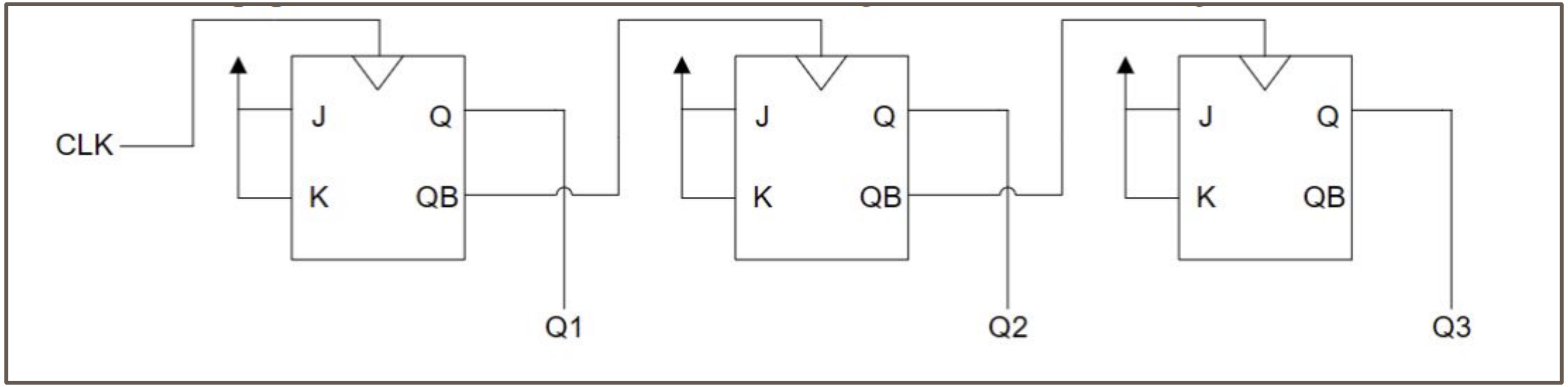
Ripple Counters (An Implementation of a Modulo Counter)

- What are the states of the counter below? What is the counter's modulus?
  - HINT: You can assume we start at state (Q3, Q2, Q1) = (0, 0, 0)
  - HINT: Q1 represents the LSD of the number {Q3, Q2, Q1}
  - HINT: When using JK Flip-Flops, **If (J, K) == (1, 1) → Q <--> QB (Toggle Q)**



Ripple Counters (An Implementation of a Modulo Counter)

- Unique States of Counter = {0, 1, 2, 3, 4, 5, 6, 7, 0, …} → **Modulus = 8**
- **What are the Pros/Cons of using this method?**



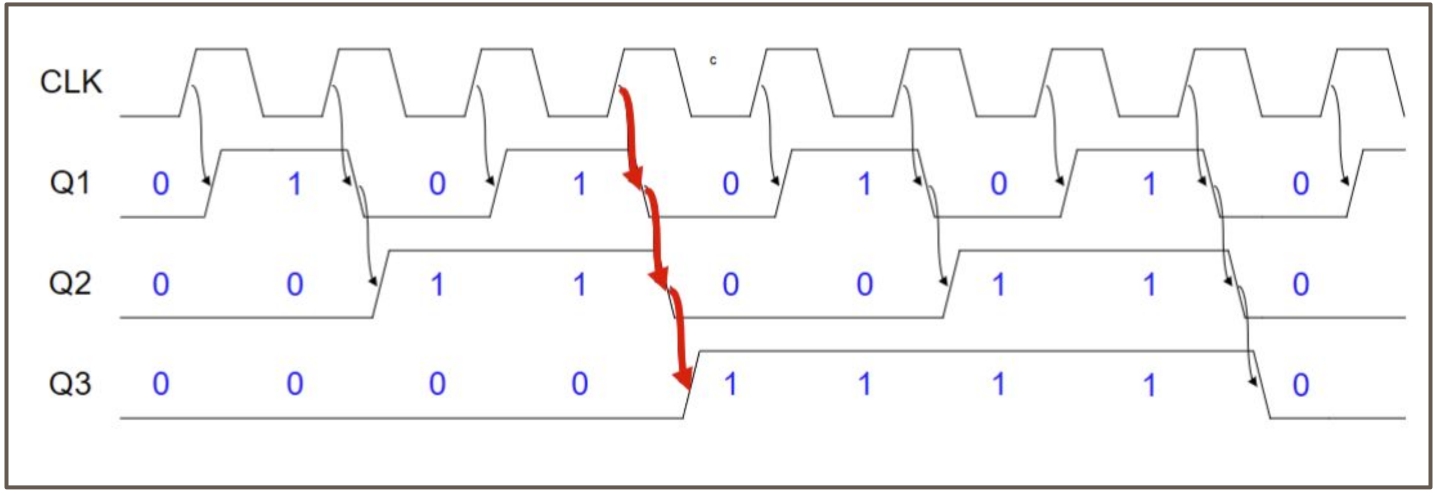
Ripple Counters (An Implementation of a Modulo Counter)

- Unique States of Counter = {0, 1, 2, 3, 4, 5, 6, 7, 0, …} → **Modulus = 8**

+ **Very straightforward approach of a modulo counter**

- **State needs to ripple through the circuit**

  - **Output doesn't change at the same time**

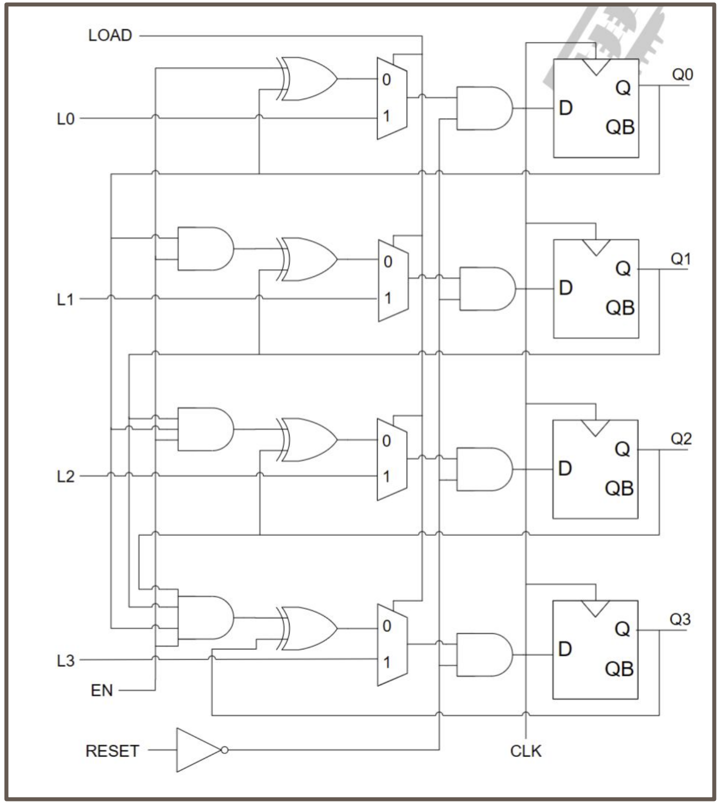


Parallel Counters (Better Modulo Counter Implementation)

- Outputs are seem right after posedge clk
  - No more ripple
- Able to do Enable, Load, and Reset
- **Can create unique modulus/states combos**

```
always @(posedge clk) begin
   If (RESET) Q <= 0
   If (LOAD)   Q <= {L3, L2, L1, L0}
   If (EN)       Q <= {Q+1}
   Else          Q <= Q
end
```

Note: If going through all states, will automatically handle the modulus

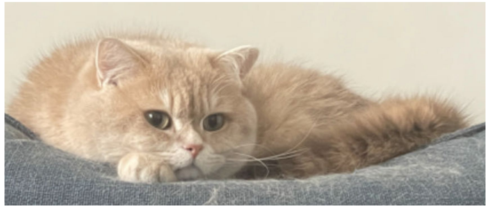


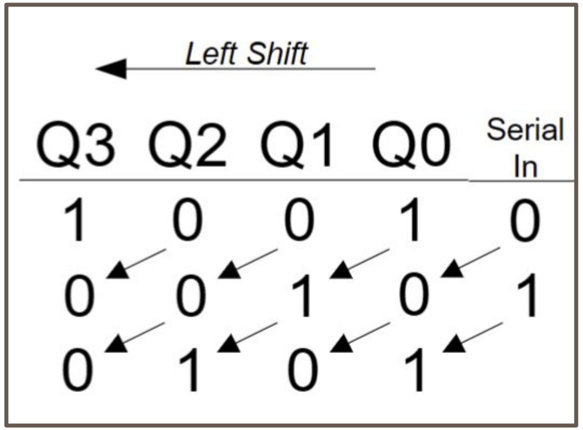
Shift Registers (Serial-In)

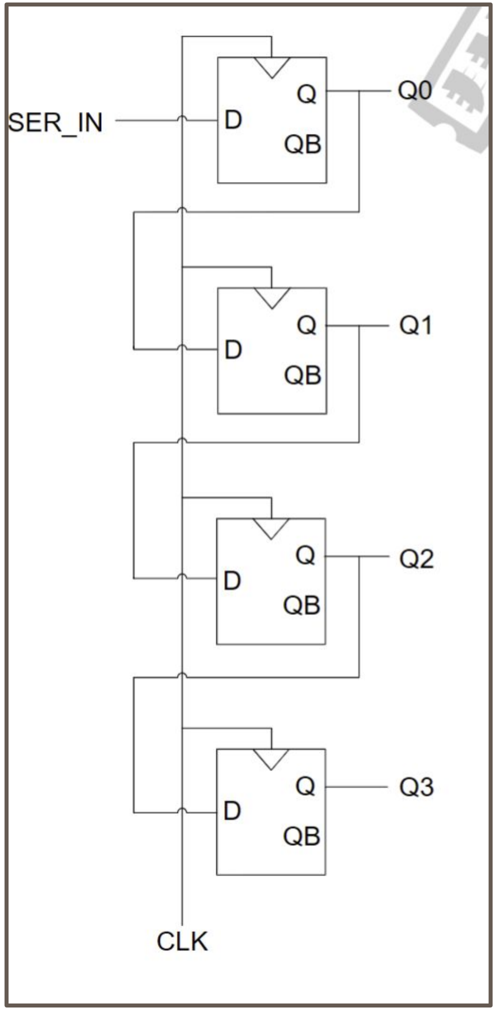
- Allowing the bits to be shifted left by one
- Specify "Serial In" bit to put in place for new Q0

+ NOTE: NOT A RIPPLE. Shifting is synchronous w/ CLK

```
always @(posedge clk) begin
   Q <= {Q2, Q1, Q0, Serial_In};
end
```



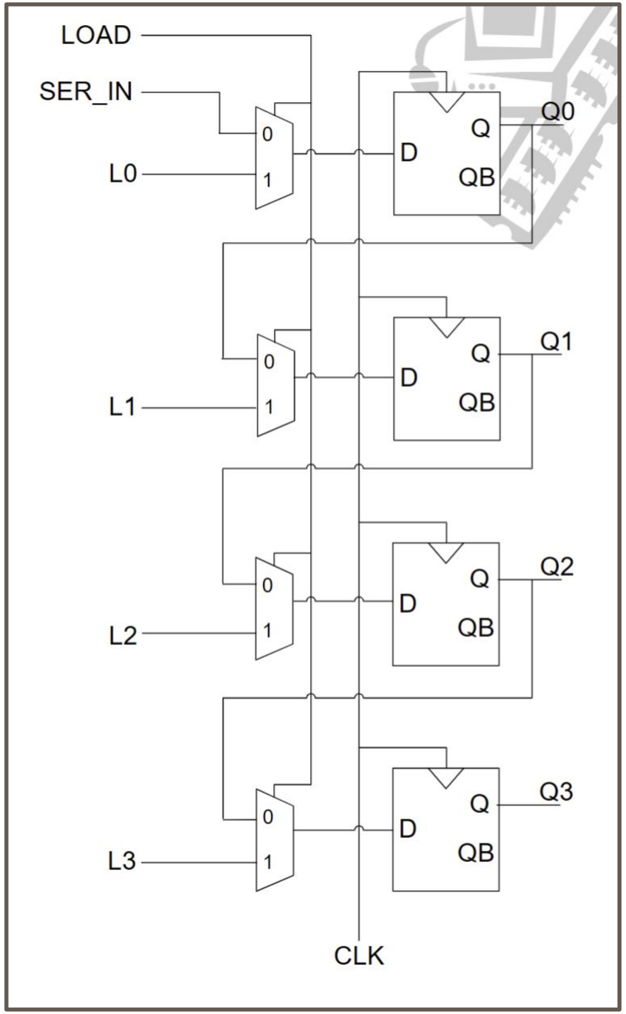




Shift Registers (Parallel-In)

- Giving the ability to load all register bits on CLK
- **Serial-In Shift Register + Load logic**

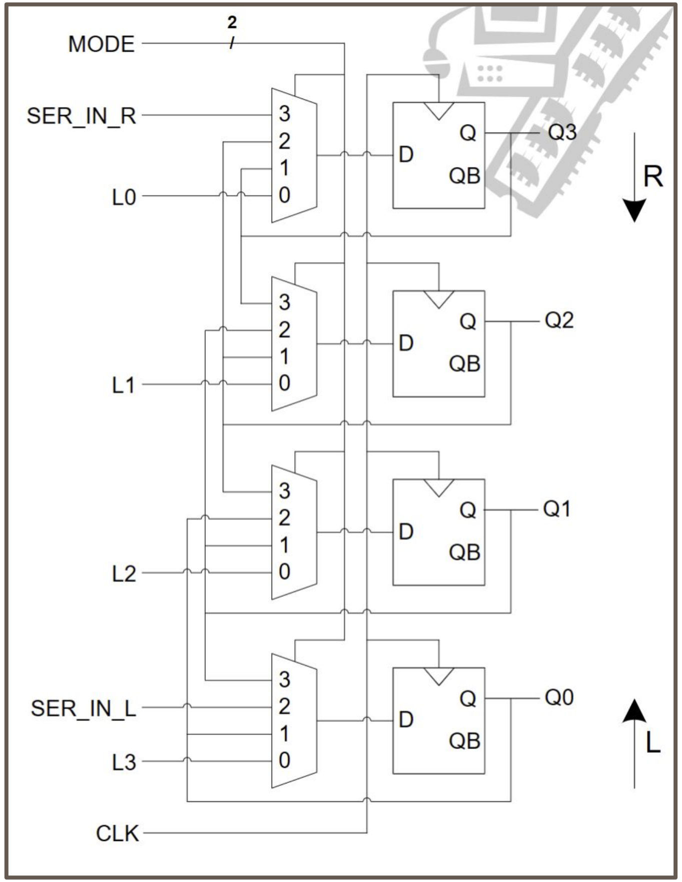
```
always @(posedge clk) begin
   If (LOAD)  Q <= {L3, L2, L1, L0};
   Else          Q <= {Q2, Q1, Q0, Serial_In};
end
```



Universal Shift Registers

- **Shift register that can**
  - **Parallel Load**
  - **Hold**
  - **Shift Left**
  - **Shift Right**

```
always @(posedge clk) begin
   If (MODE == LOAD)  Q <= {L3, L2, L1, L0};
   If (MODE == HOLD)  Q <= Q;
   If (MODE == SL)     Q <= {Q2, Q1, Q0, S_In_L};
   If (MODE == SR)     Q <= {S_In_R, Q3, Q2, Q1};
end
```

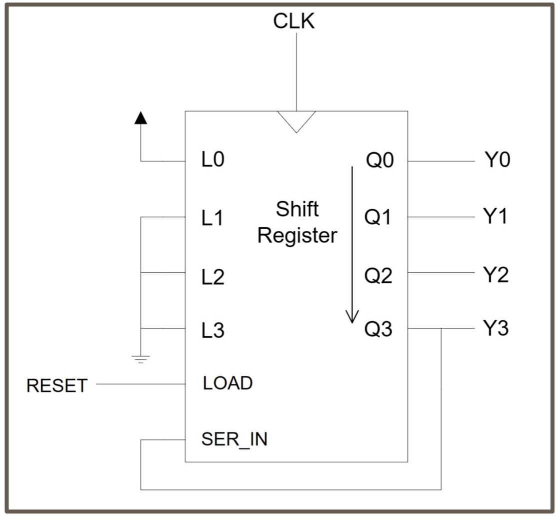


Ring Counter

- Each Flip Flop value propagates down the chain
- One-hot encoding scheme
- Check which Flip-Flop is high to find the count number

```
always @(posedge clk) begin
    If (RESET) Q <= 4'b0001;
    Else       Q <= {Q[2:0], Q3};
end
```

- **How many states exist in an n-bit ring counter?**



Straight ring counter

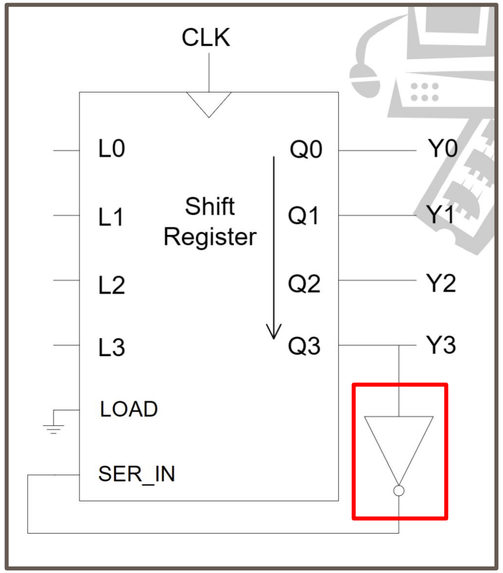
State	Q0	Q1	Q2	Q3
0	1	0	0	0
1	0	1	0	0
2	0	0	1	0
3	0	0	0	1
0	1	0	0	0
1	0	1	0	0
2	0	0	1	0
3	0	0	0	1
0	1	0	0	0

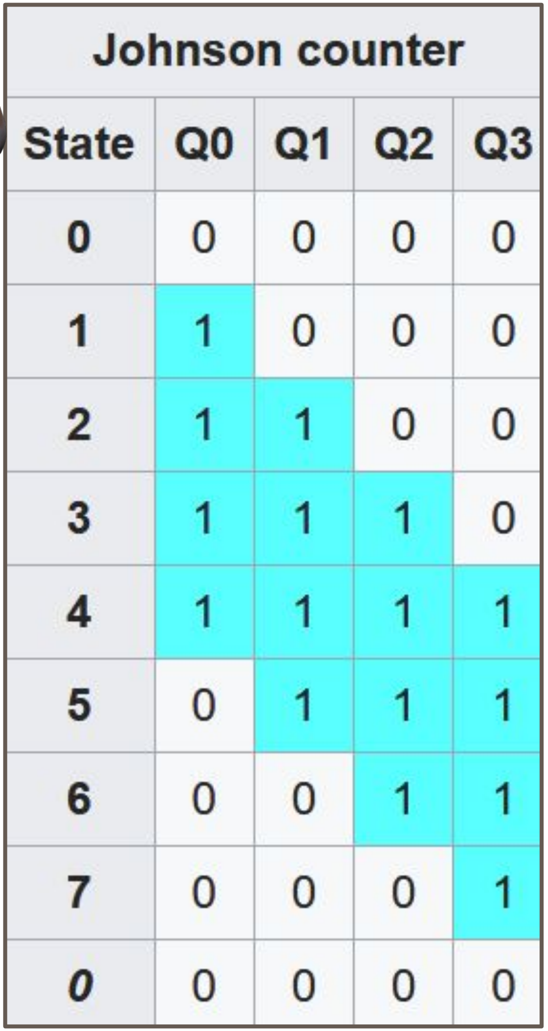
Johnson Counter

- Very similar to a ring counter
- **MSB is inverted and chained to the first input**

```
always @(posedge clk) begin
   If (RESET) Q <= 4’b0000;
   Else          Q <= {Q[2:0], ~Q3};
end
```

- **How many states exist in an n-bit Johnson counter?**



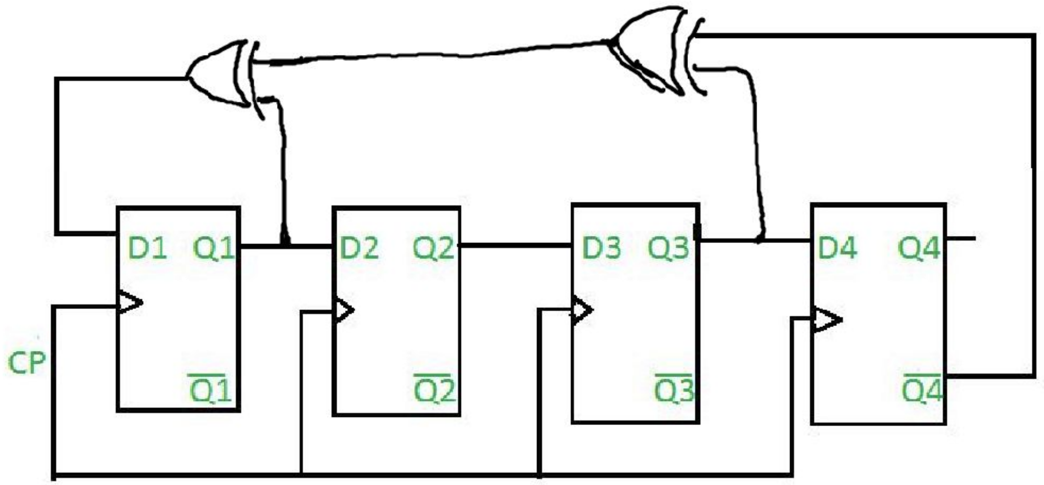


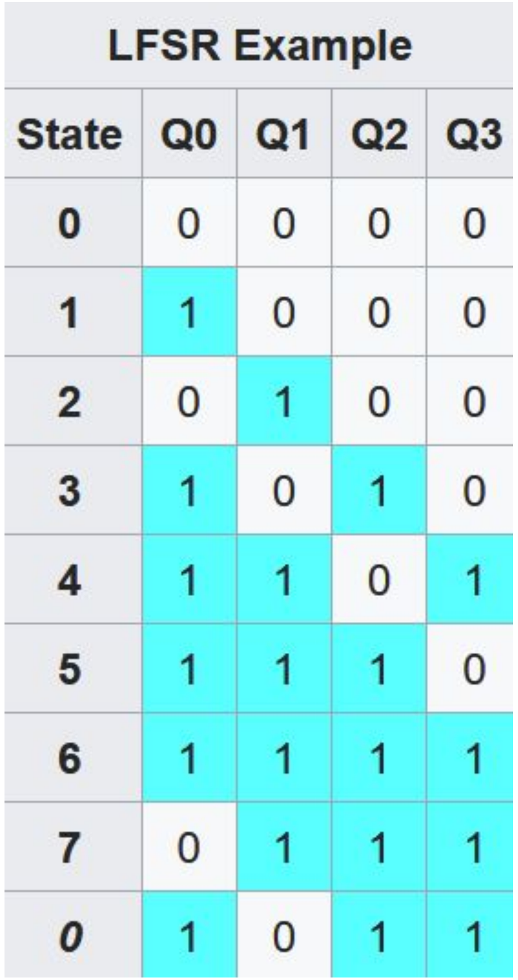
Johnson counter

State	Q0	Q1	Q2	Q3
0	0	0	0	0
1	1	0	0	0
2	1	1	0	0
3	1	1	1	0
4	1	1	1	1
5	0	1	1	1
6	0	0	1	1
7	0	0	0	1
0	0	0	0	0

Linear Feedback Shift Registers

- Modification of ring counter
- First input is a function of LFSR value
- **Can cover 2^n - 1 states!**





LFSR Example

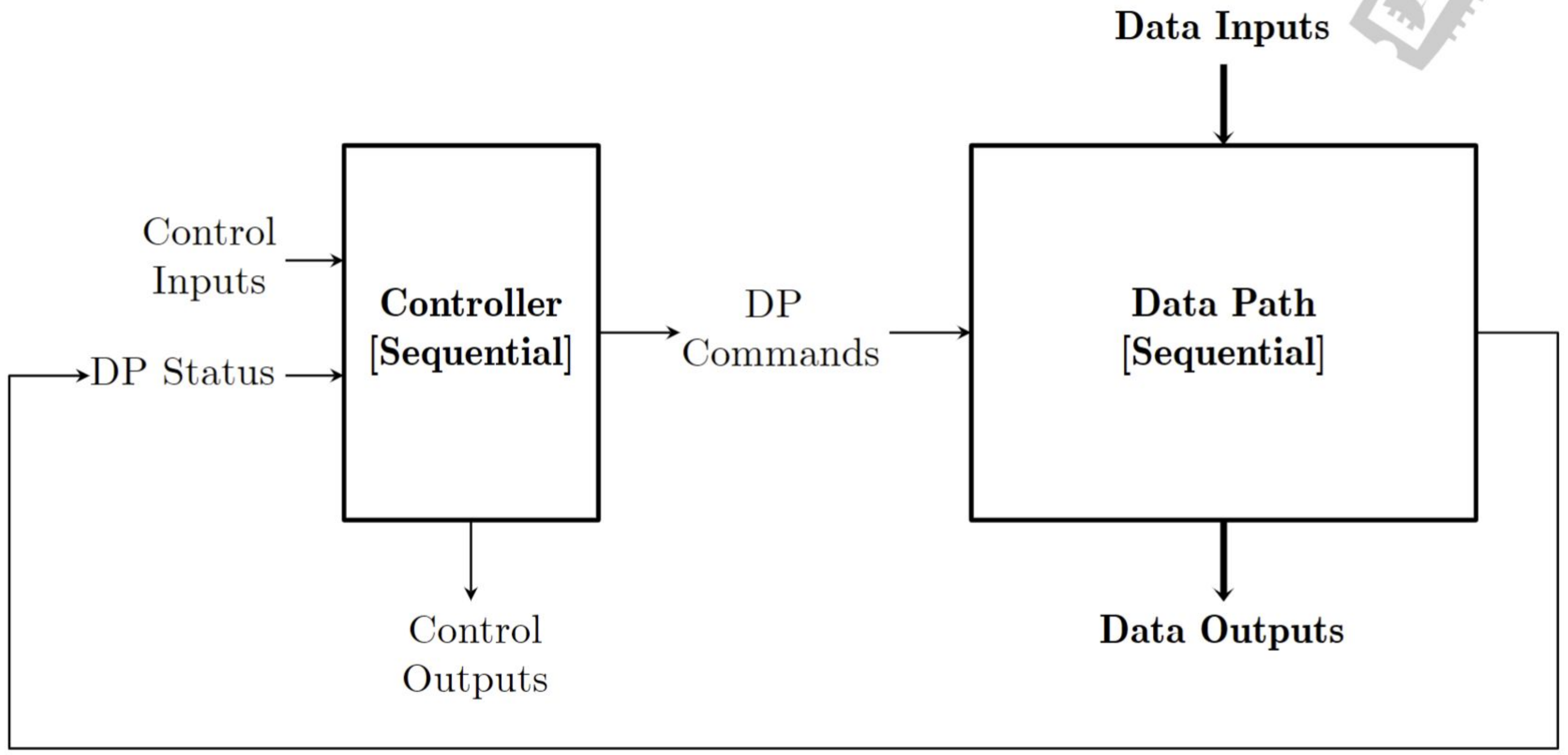
State	Q0	Q1	Q2	Q3
0	0	0	0	0
1	1	0	0	0
2	0	1	0	0
3	1	0	1	0
4	1	1	0	1
5	1	1	1	0
6	1	1	1	1
7	0	1	1	1
0	1	0	1	1

Register Transfer Level (RTL) Design

RTL Design Overview

- Splitting up design in Controller and Datapath
- **Controller** - Controls and orchestrates the Datapath operations
- **Datapath** - Computation on the signals
  - Arithmetic - Add/Subtract/Multiple/Count/…
  - Logical - Shift Left / Shield Right / Arithmetic Shift / bitwise Ops
  - Other - Clear / Load / Hold

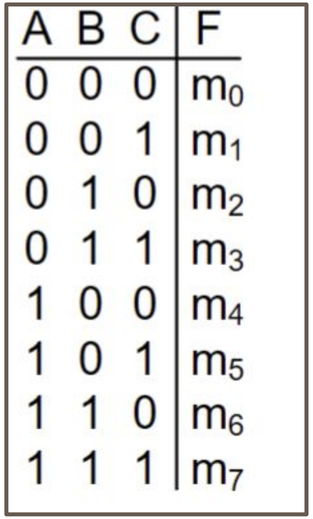
RTL Sequential Circuit



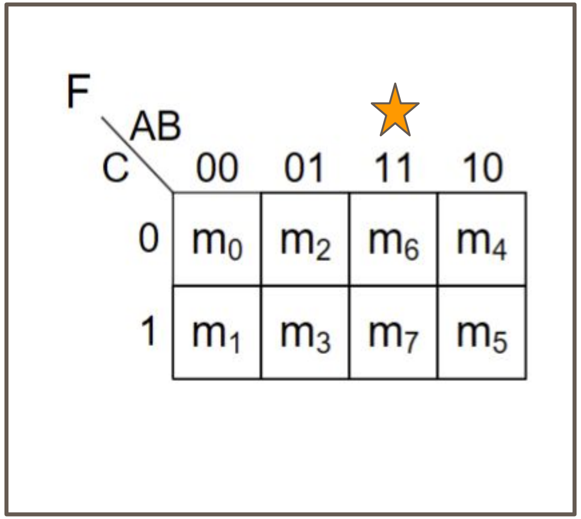
Karnaugh Maps (K-Maps)
(Logic Minimization)

K-Maps Overview

- Minterm: AND of every variable as itself or its complement(used to form SOP)
- Maxterm: OR of every variable as itself or its complement (used to form POS)
- K-Maps: A 2D representation of a Truth Table that visualizes which minterms/maxterms we can combine to make simpler expressions



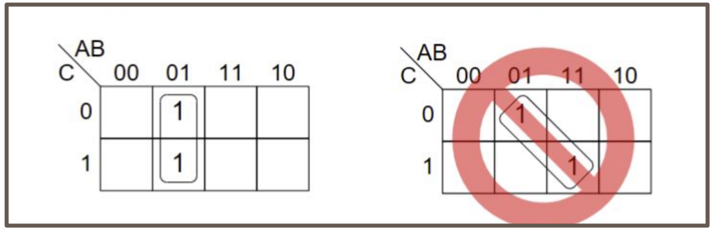
A	B	C	F
0	0	0	m_0
0	0	1	m_1
0	1	0	m_2
0	1	1	m_3
1	0	0	m_4
1	0	1	m_5
1	1	0	m_6
1	1	1	m_7



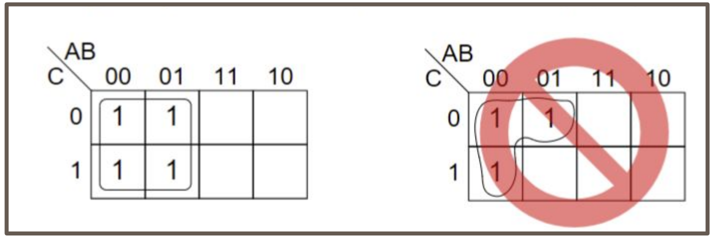
F: C \ AB	00	01	11	10
0	m_0	m_2	m_6	m_4
1	m_1	m_3	m_7	m_5

K-Map Rules

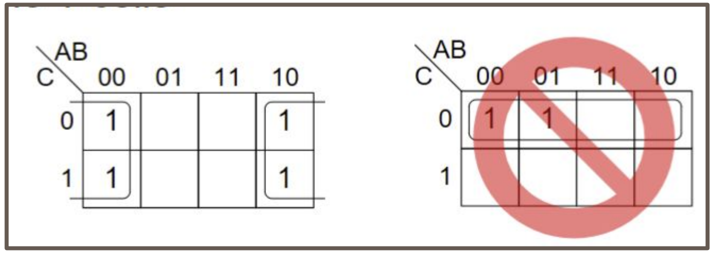
1. Only cover cells that have a 1
   (only include ‘don’t cares’ if it helps to enlarge a group)



2. Only circle groups that are powers of 2



3. Only circle adjacent cells



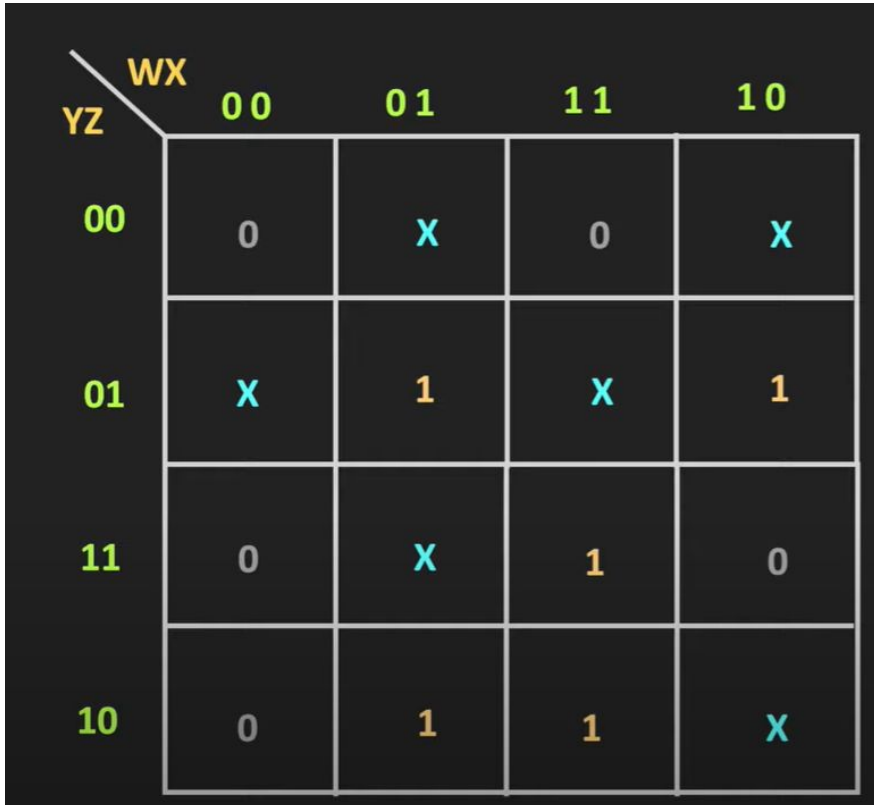
Important K-Map Terminology (Prime & Ess. Prime Impl.)

- **Implicant:** A rectangle that covers any high outputs
- **Prime implicant:** The largest single implicant that can fully cover a set of high outputs
- **Essential prime implicant:** A prime implicant that cannot be completely covered by any combination of other prime implicants

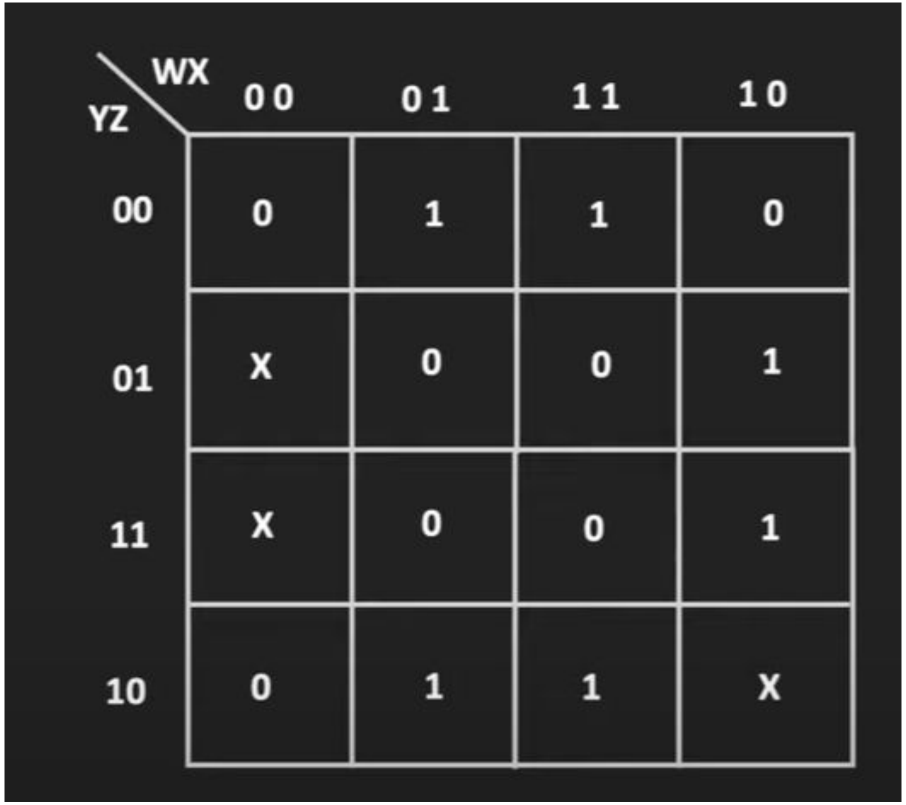
K-Map Minimization Procedure

1. Identify all of your prime implicants
2. Determine which of the prime implicants are essential
   a. Include those EPIS into your minimal SOP
3. Remove any “dominated” prime implicants
4. Find secondary Essential Prime Implicants (Repeat)

Let’s Try Out a Few



YZ \ WX	00	01	11	10
00	0	X	0	X
01	X	1	X	1
11	0	X	1	0
10	0	1	1	X



YZ \ WX	00	01	11	10
00	0	1	1	0
01	X	0	0	1
11	X	0	0	1
10	0	1	1	X

Have a cute pet? See it featured on the next exam review!

https://forms.gle/ZS4YfU8RH4iJaa626