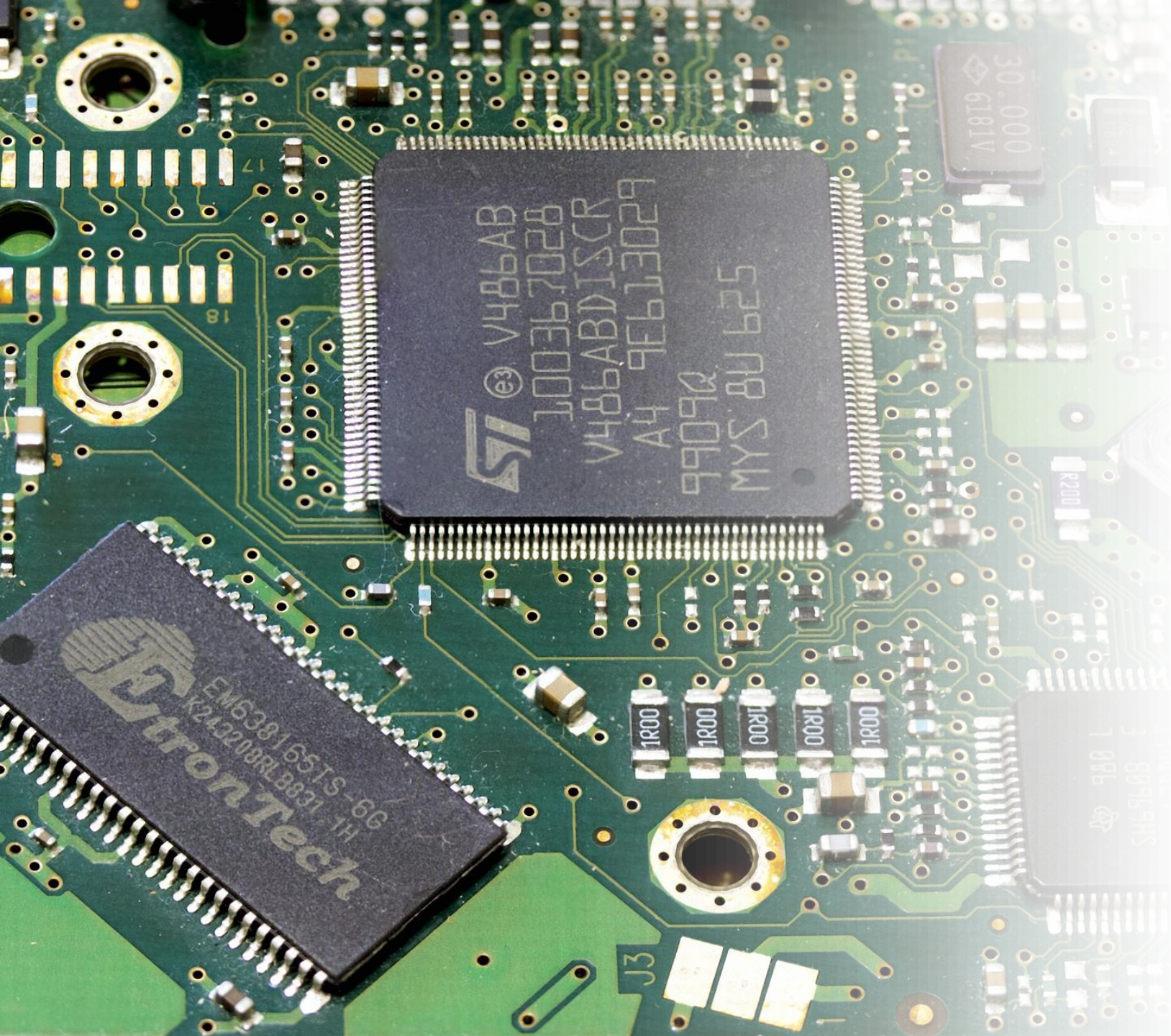


Digital Logic and System Design

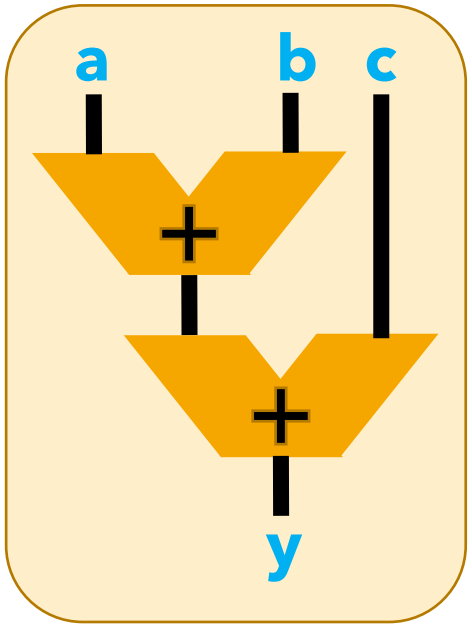
7. Sequential Logic

COL215, I Semester 2024-2025
Venue: LHC 408
'E' Slot: Tue, Wed, Fri 10:00-11:00

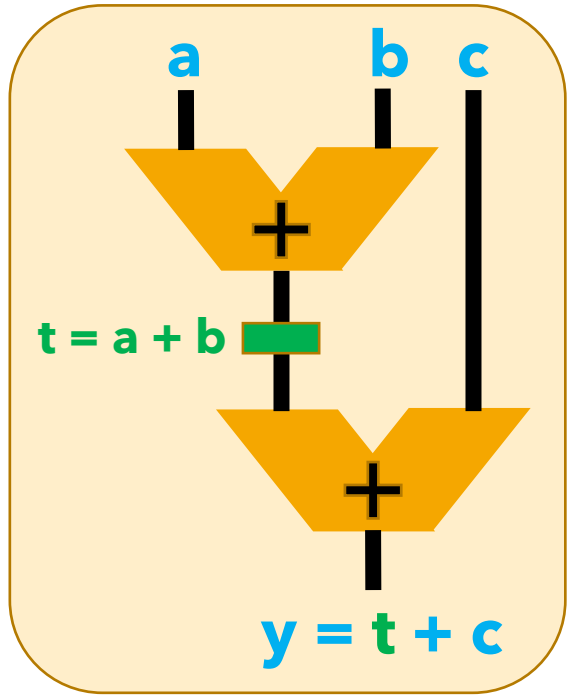
Instructor: Preeti Ranjan Panda
panda@cse.iitd.ac.in
www.cse.iitd.ac.in/~panda/
Dept. of Computer Science & Engg., IIT Delhi

Sharing Resources

- Implement: **y = a + b + c**
- ...with only **ONE adder**
- **REUSE** the adder
- Need to **STORE** intermediate value
  - **t = a + b**
  - **y = t + c**



2 Adders



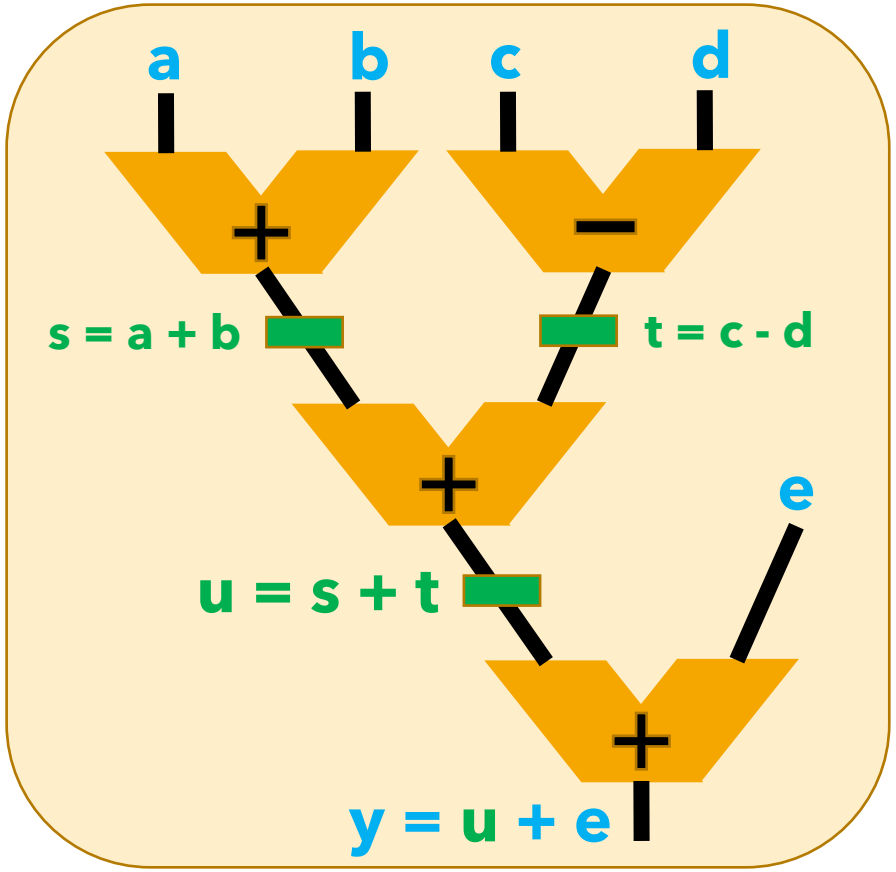
Store intermediate value **t**
Use later for 2nd addition

Complex Computation

- Break up complex function/computation
  - intermediate values
  - similar to software
- Efficient resource utilization
  - same adder **reused** for several additions

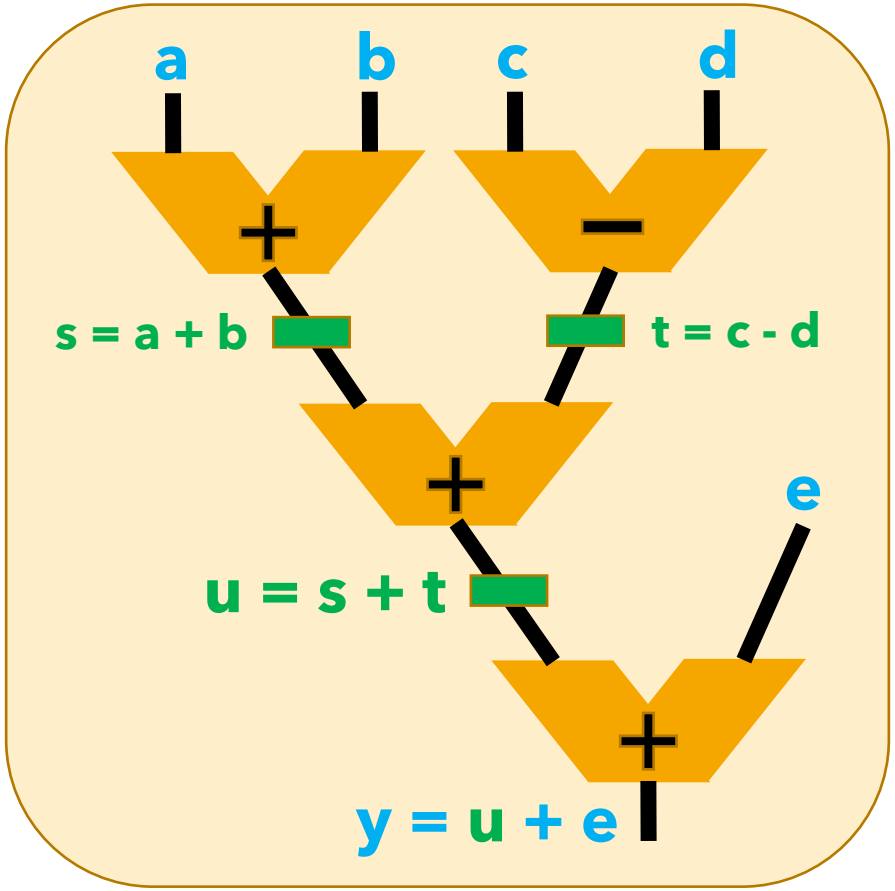
$$y = a + b + c - d + e$$

$$s = a + b$$
$$t = c - d$$
$$u = s + t$$
$$y = u + e$$



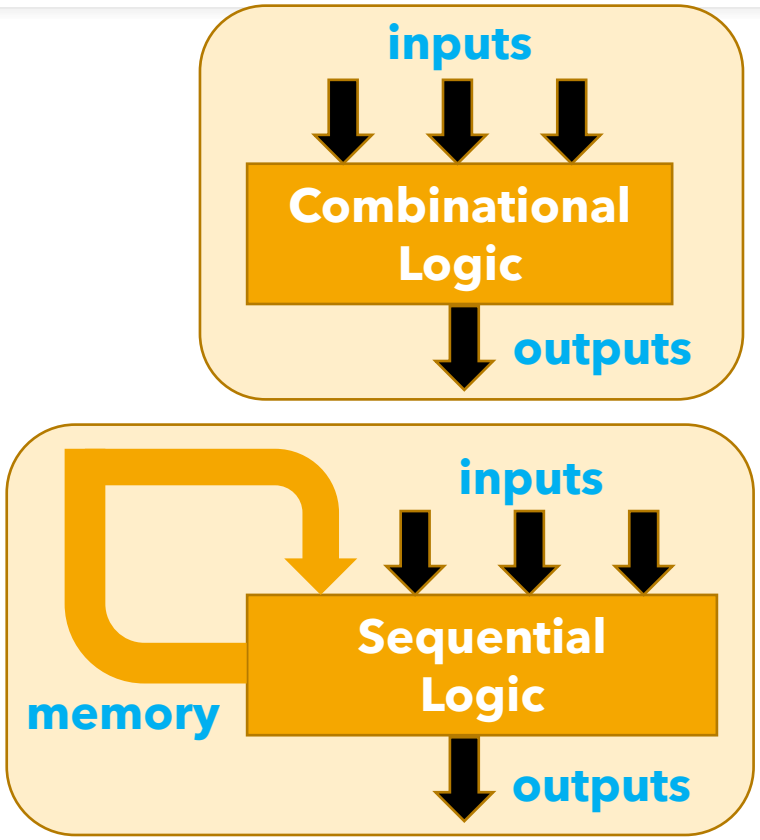
New Hardware Components

- Elementary operation:
  - **Store** a value
  - use it later
- **Control** and **Sequencing**
  - Manage **order** of operations
  - Decide what action is performed by unit (ADD vs SUB)
- Provide alternative **paths** for operands



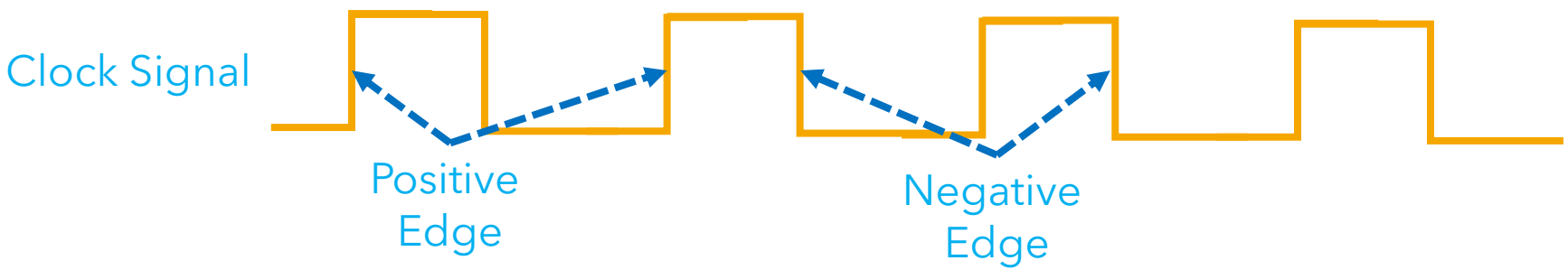
Combinational vs. Sequential Logic

- Combinational:
  - **Output** is a function only of **present inputs**
  - Independent of **earlier** input values
- Sequential:
  - Output is a function of **present input AND earlier inputs**
    - **memory** is stored



Synchronous Design

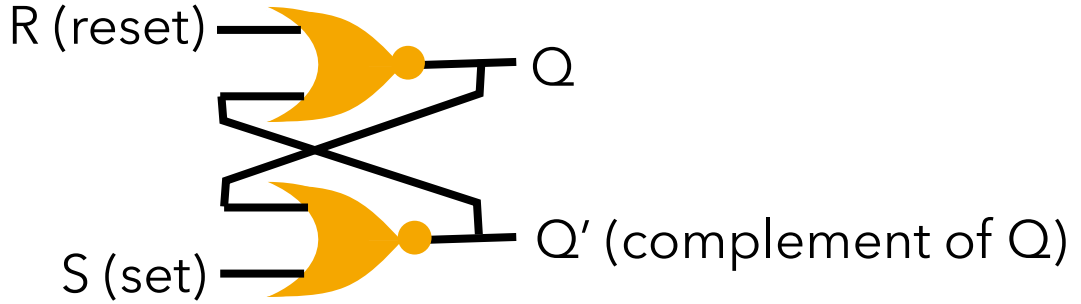
- Computation and Storage are associated with **CLOCKs**
  - Periodic signals
  - Events triggered at **Clock Edges**
    - **positive edge: (0 to 1) transition**
    - **negative edge: (1 to 0) transition**

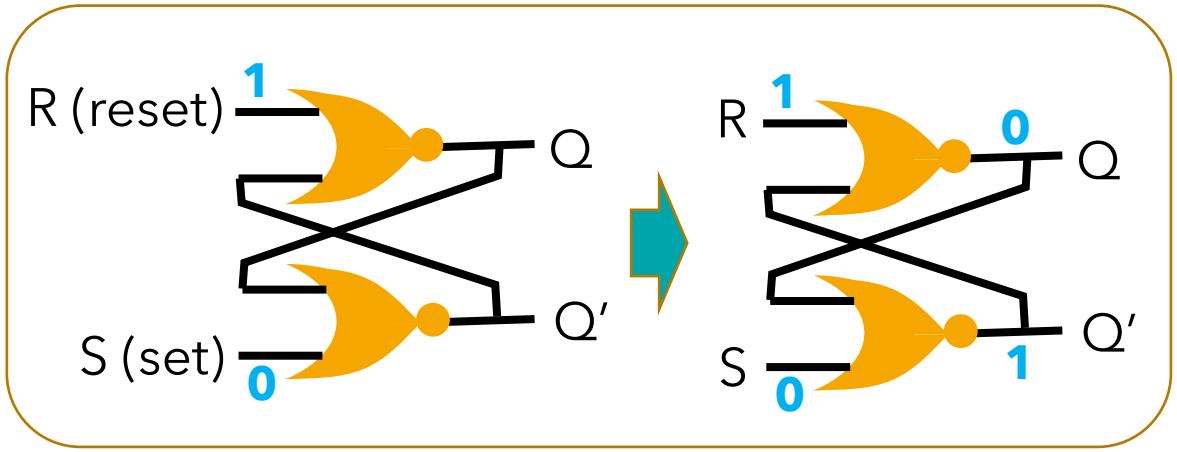


Alternative: Asynchronous Design

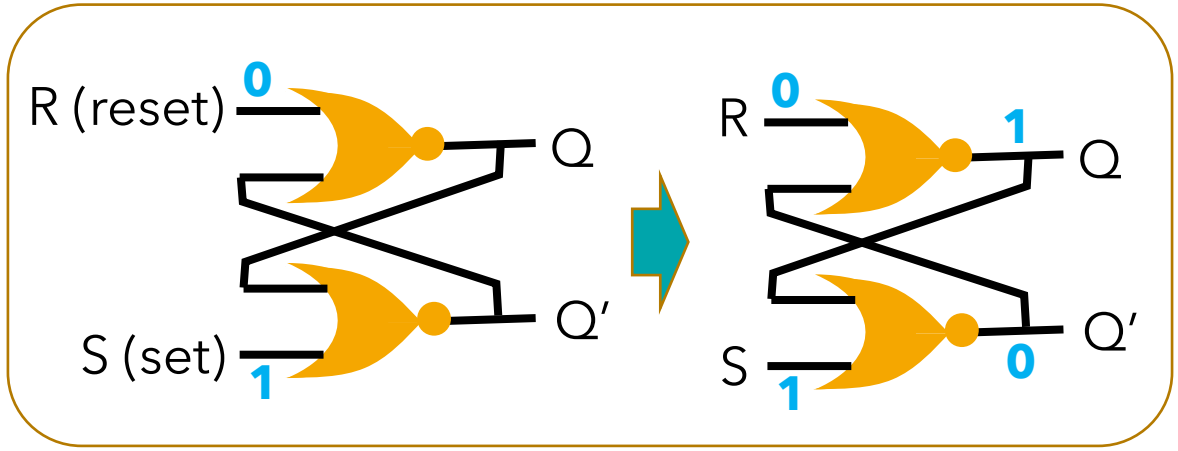
- No common clock
- Completion signalled by individual components
  - Contrast: In Synchronous Design completion is checked only at clock edges

Storage Element: SR Latch



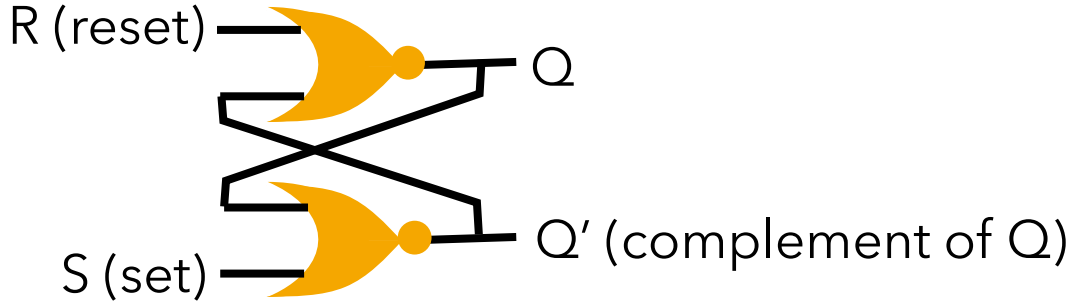


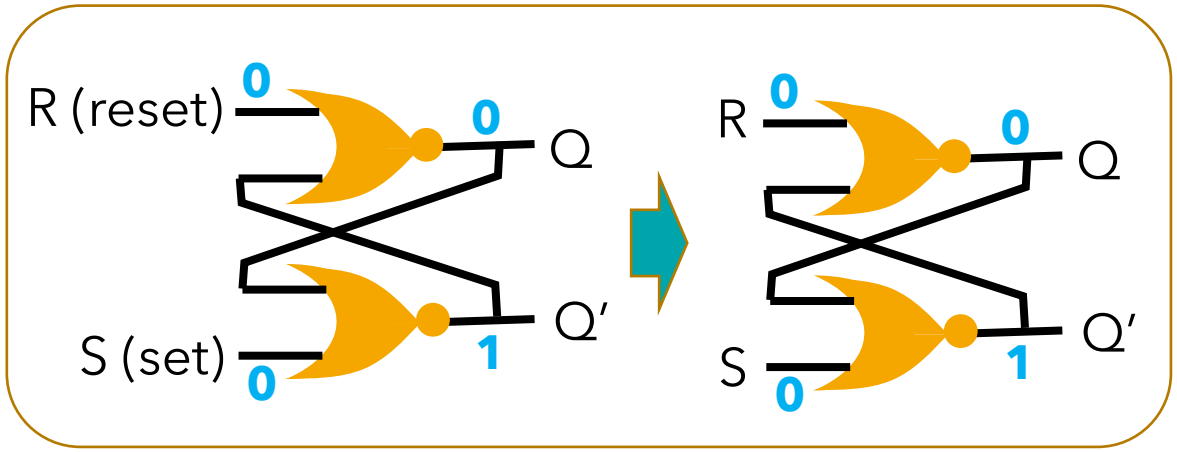
RS = 10: Outputs **QQ′=01** (Q **reset** to **0**)



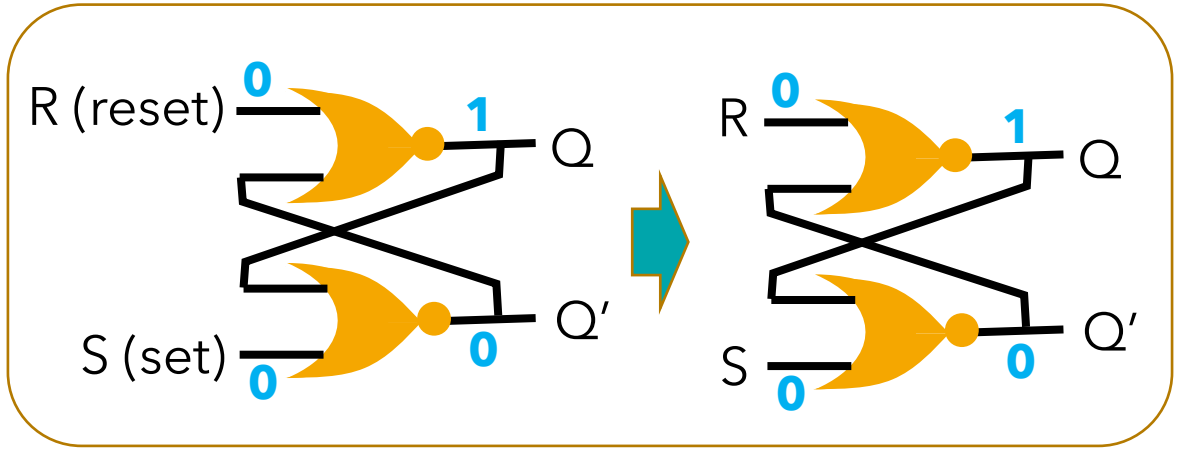
RS = 01: Outputs **QQ′=10** (Q **set** to **1**)

Storing the Q value



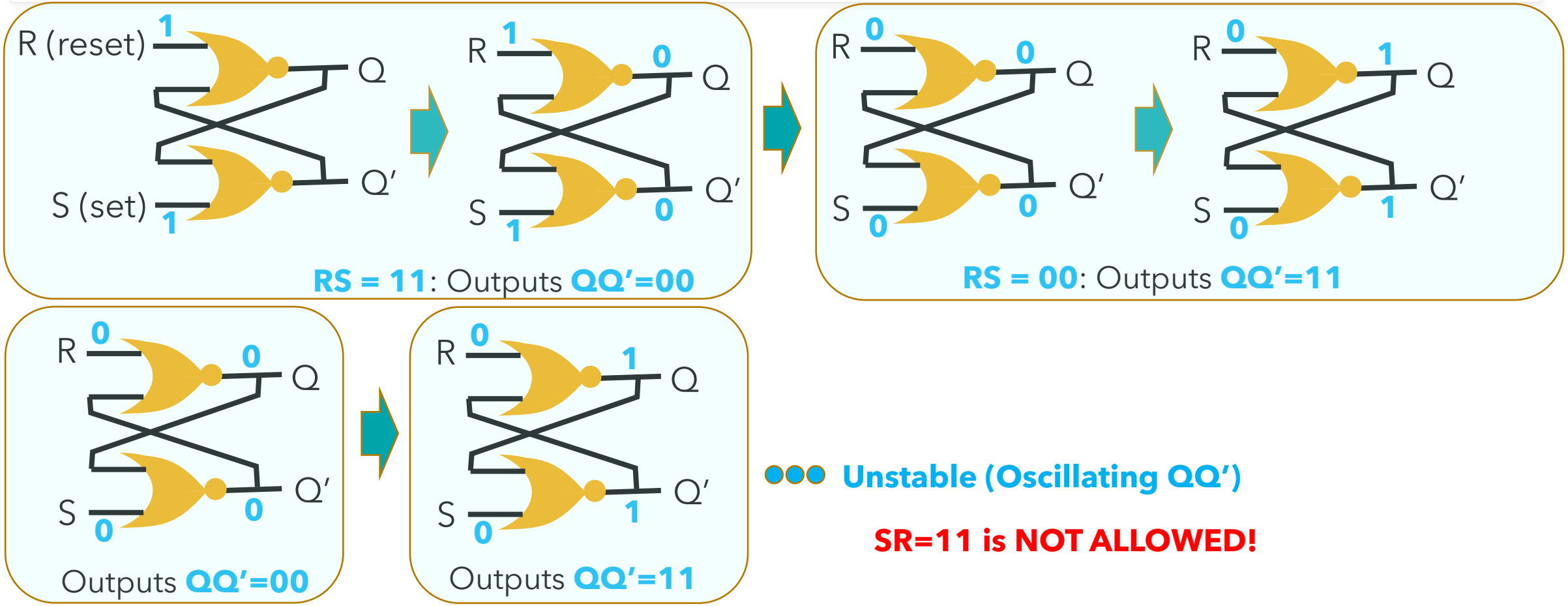


RS = 00: Outputs **QQ'=01 maintained (no change)**

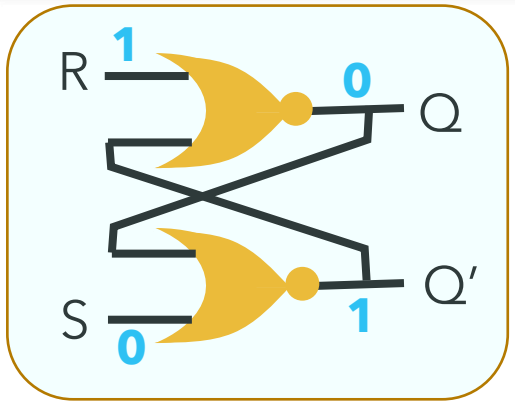


RS = 00: Outputs **QQ'=10 maintained (no change)**

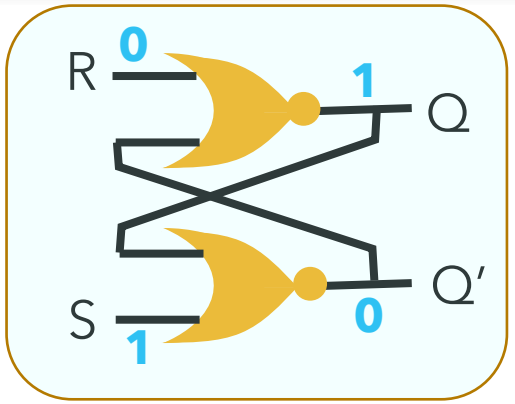
Unstable conditions in SR Latch



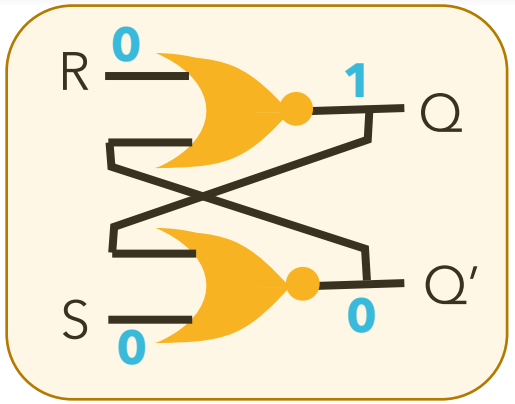
We have a **storage/memory device**!

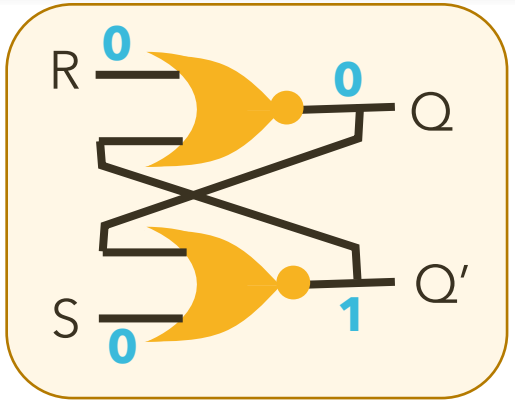


RS = 10: Output **Q=0**



RS = 01: Output **Q=1**

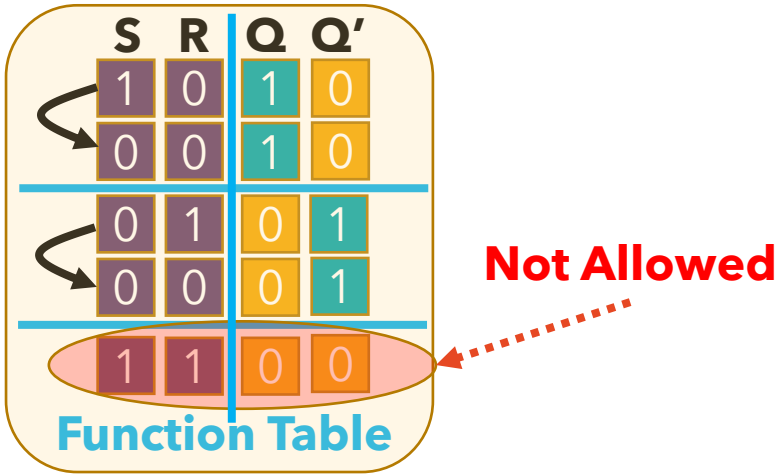




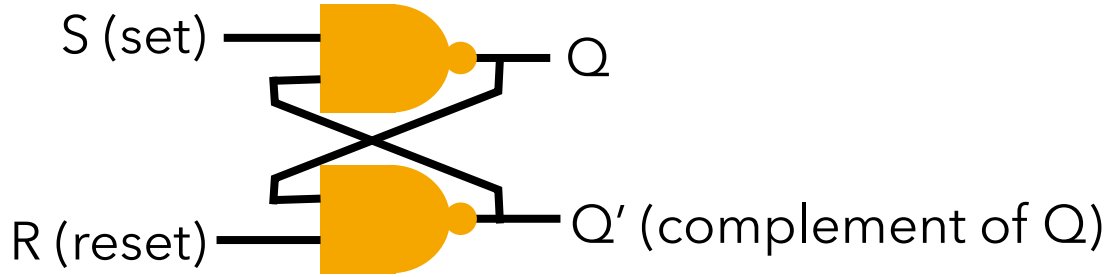
RS = 00: Output **Q unchanged**

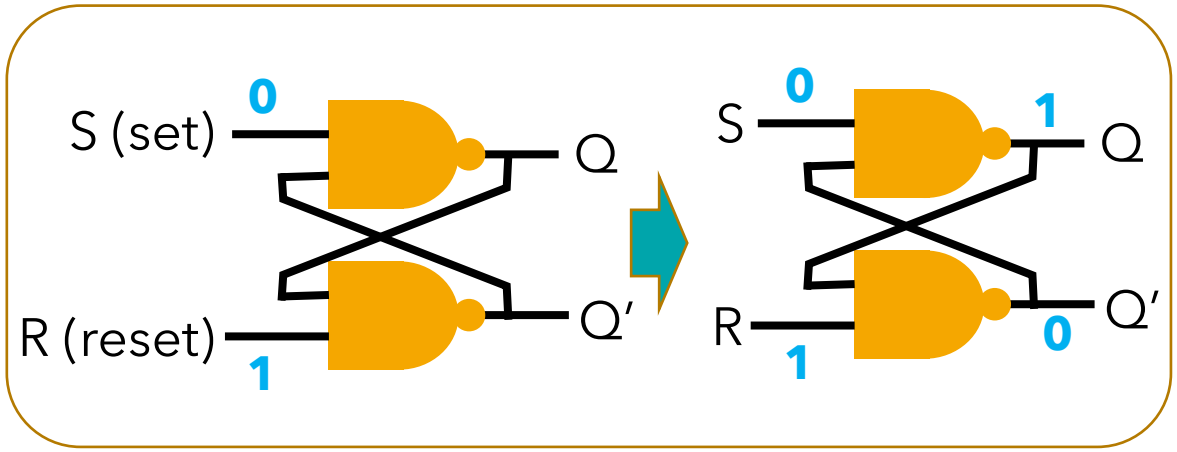
Working of the SR Latch

- Normally, SR=00
- If Q=0 needed, make SR=01. Return to SR=00.
- If Q=1 needed, make SR=10. Return to SR=00.
- Don't set SR=11
- Don't go directly from SR=01 to SR=10

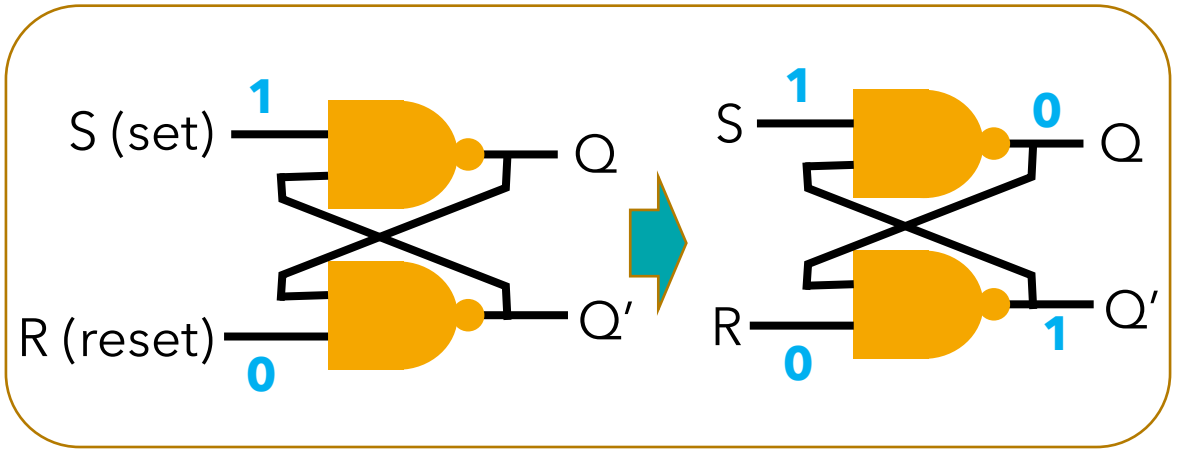


SR Latch with NAND gates



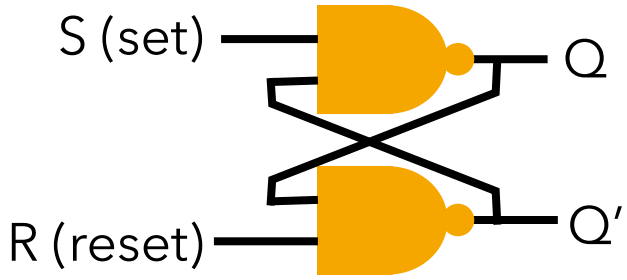


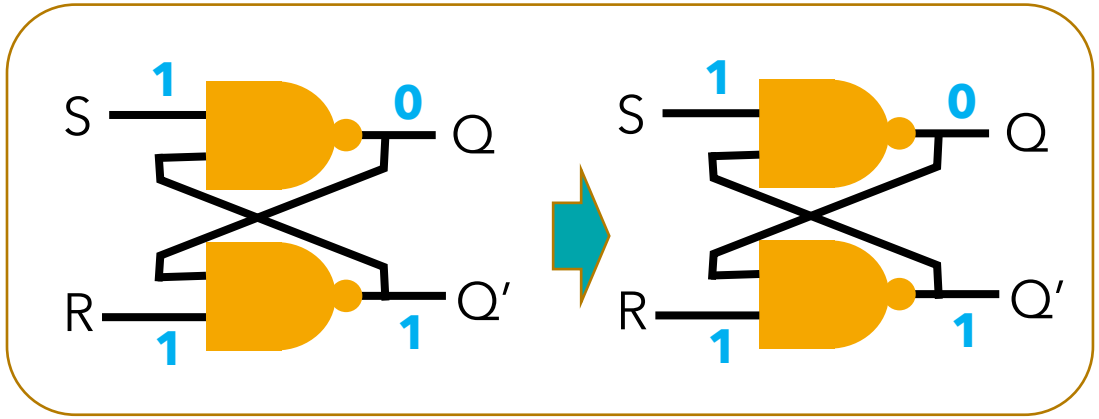
SR = 01: Outputs **QQ′=10** (Q **set** to **1**)



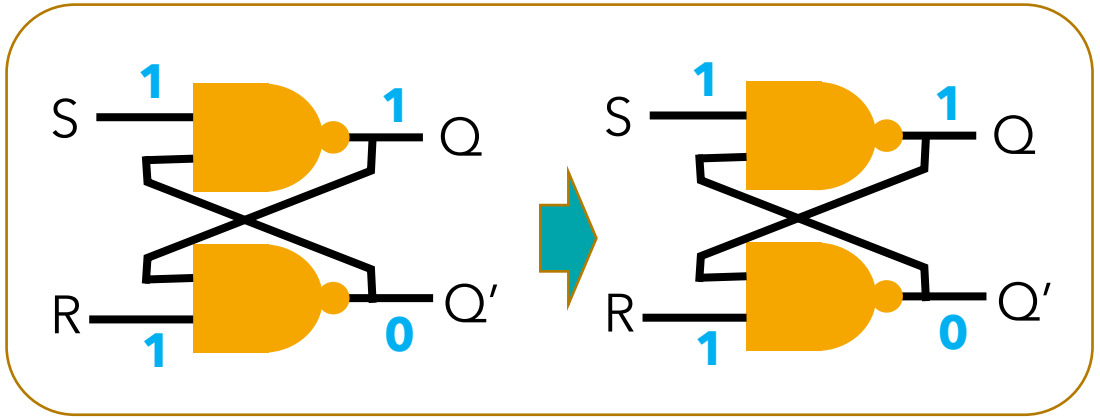
SR = 10: Outputs **QQ′=01** (Q **reset** to **0**)

Storing the Q value



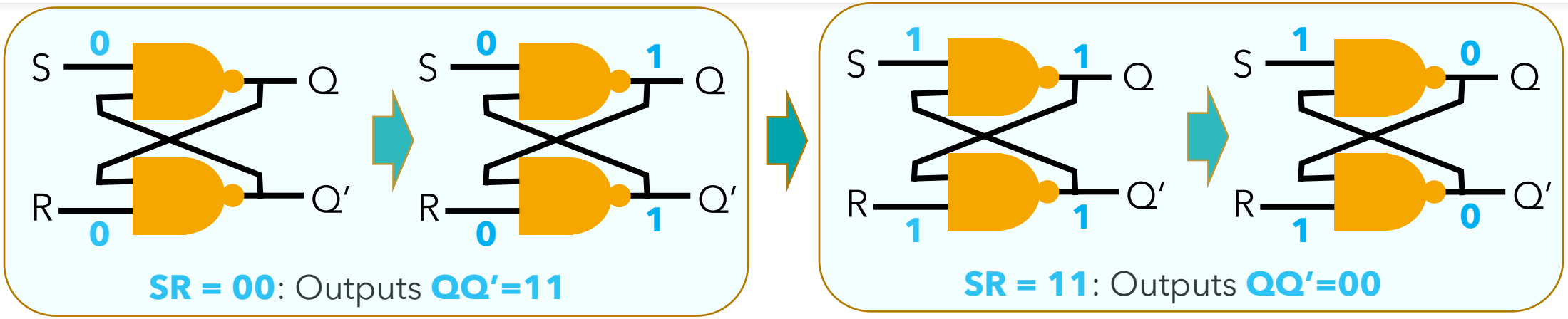


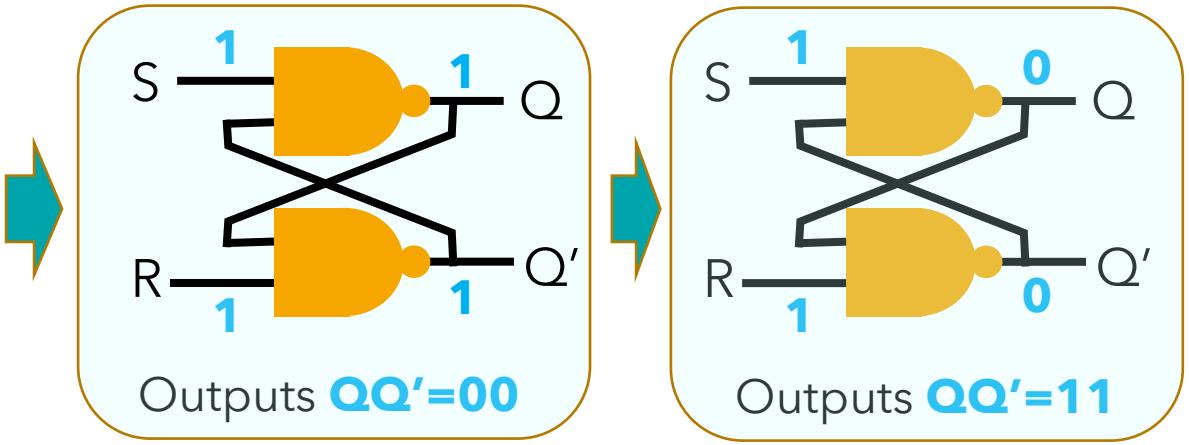
SR = 11: Outputs **QQ'=01 maintained (no change)**



SR = 11: Outputs **QQ'=10 maintained (no change)**

Unstable conditions in NAND Latch

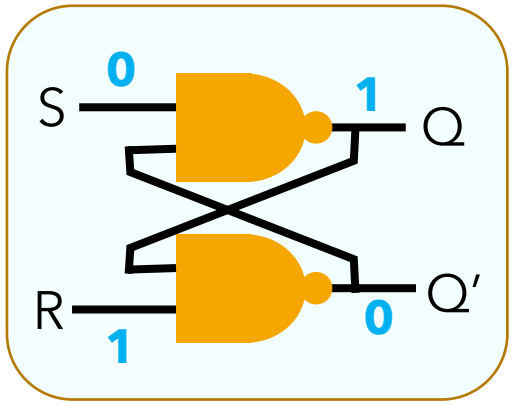




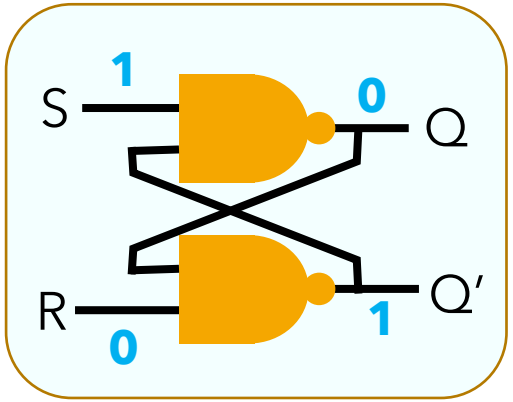
Unstable (Oscillating QQ')

SR=00 is NOT ALLOWED!

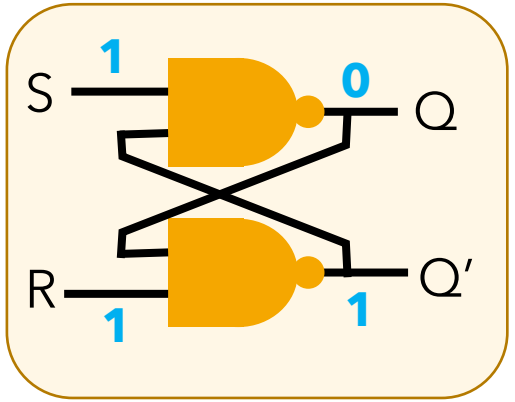
NAND latch is also a **storage/memory device**

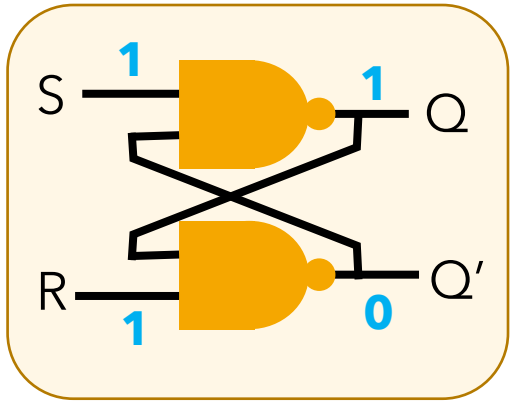


SR = 01: Output **Q=1**



SR = 10: Output **Q=0**

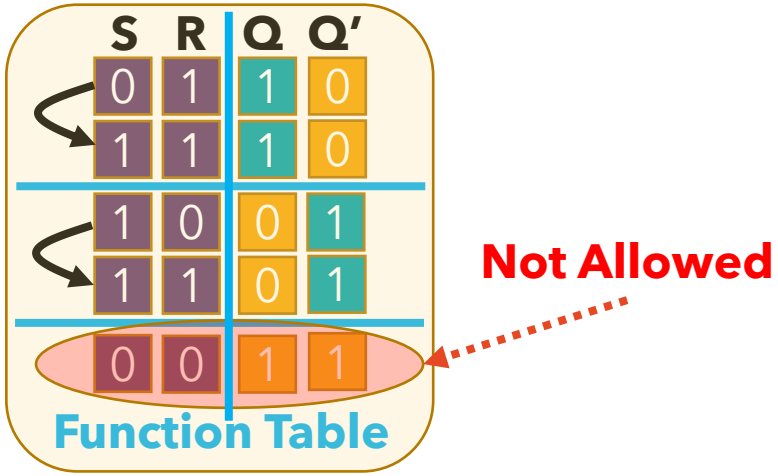




SR = 11: Output **Q unchanged**

Working of the NAND SR Latch

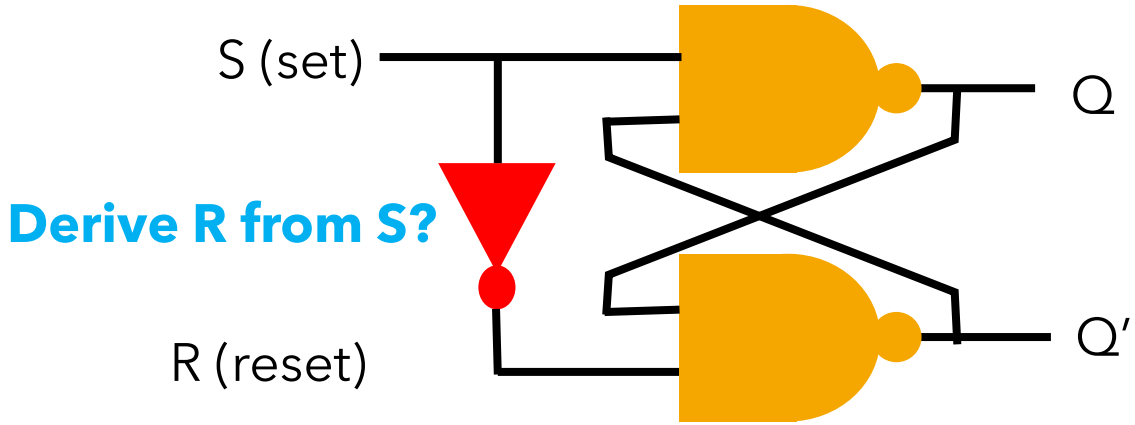
- Normally, SR=11
- If Q=0 needed, make SR=10. Return to SR=11.
- If Q=1 needed, make SR=01. Return to SR=11.
- Don't set SR=00
- Don't go directly from SR=01 to SR=10



NAND/NOR Latch needs enhancement

- Signal to **enable writing**
- When enabled, **S** and **R** are complements (**01** or **10**)

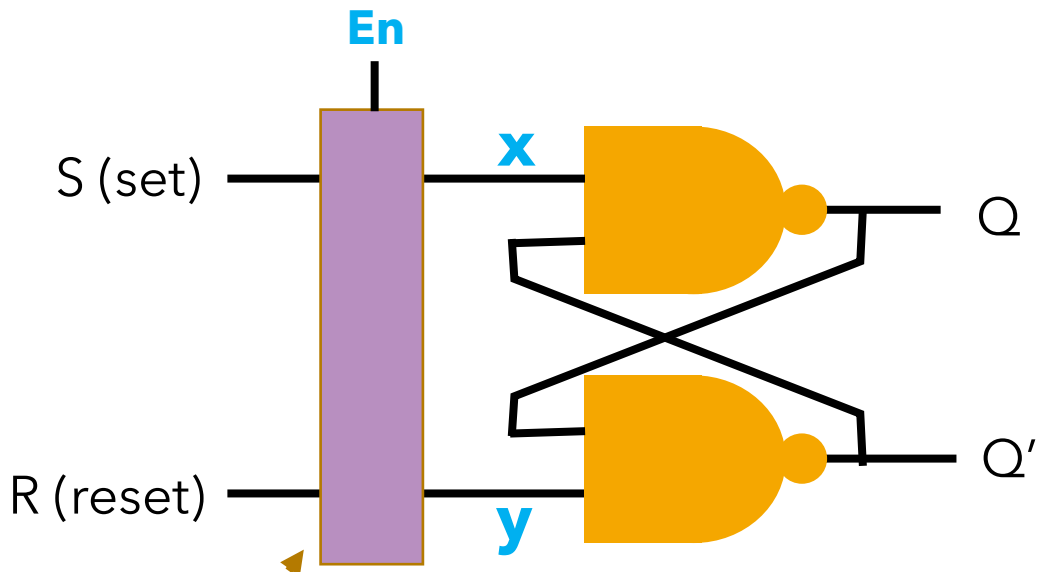
Ensuring complement



Unsafe. May transition through **SR=00**

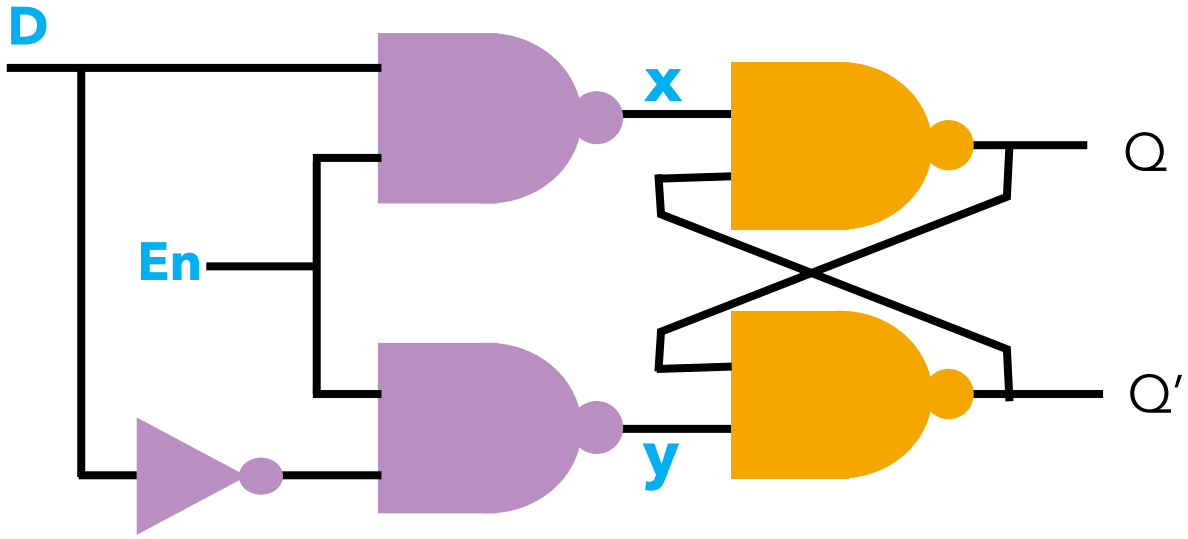
Enabling writing

- **En** (enable) signal
  - When **En=0**: **x = y = 1**
    - save state
  - When **En = 1**:
    - **x = S'**
    - **y = R'**
- What should go here?



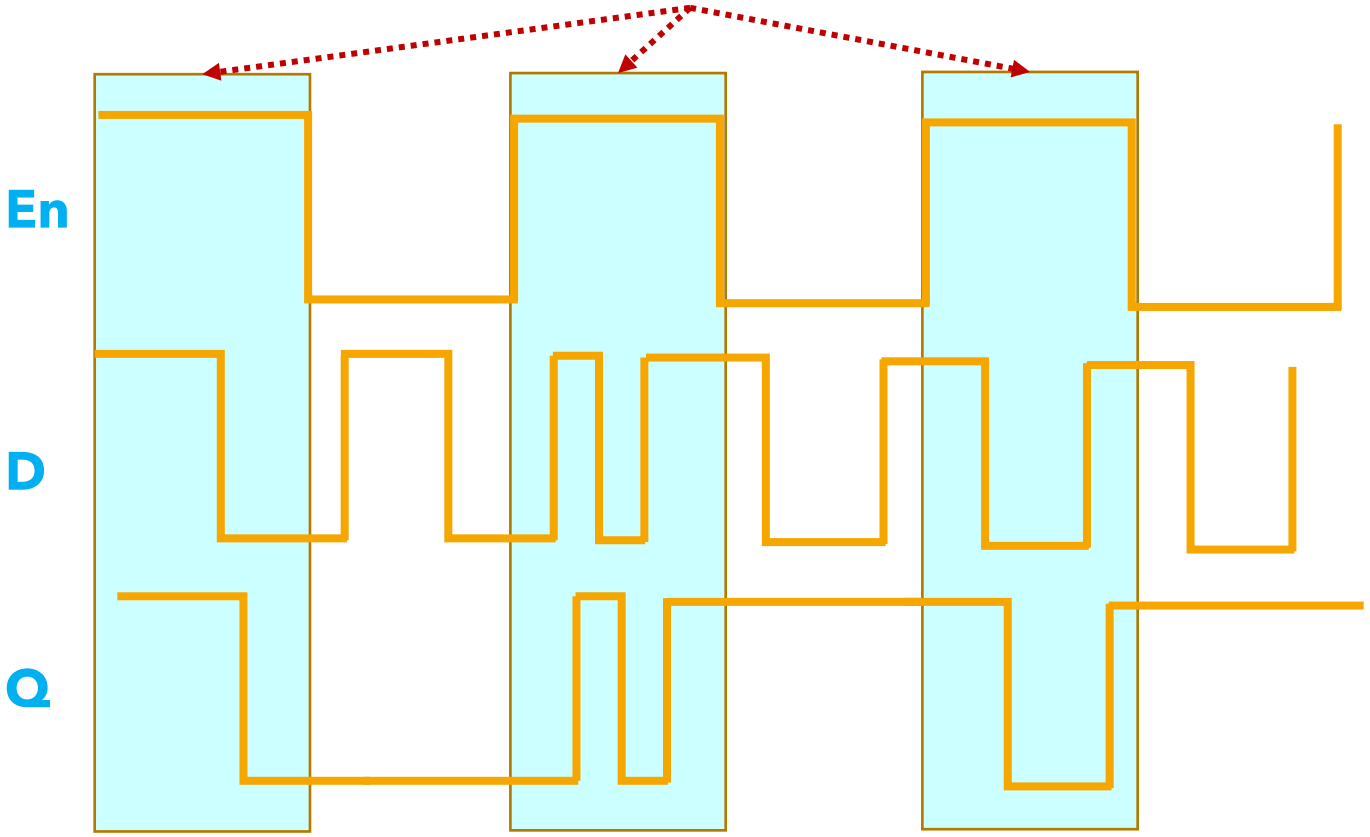
The D (data) Latch

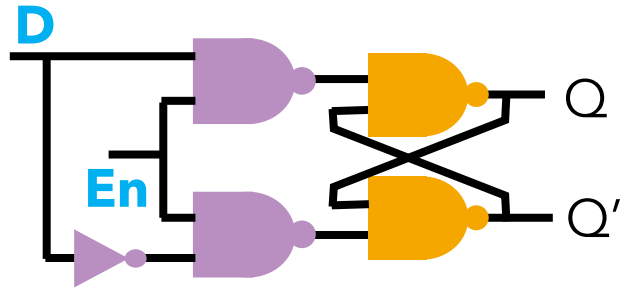
- **En** (enable) signal
  - When **En=0**: **x = y = 1**
    - save state
  - When **En = 1**:
    - **x = D'**
    - **y = D**
- **D** input **ignored** when **En = 0**
- When **En=1**: **Q = D**
- Q follows D when **En enabled**
- Last D value saved when **En disabled**



The D Latch Functionality

Q follows **D** when **En** is ON





previous Q value

En	D	Q
0	0	q
0	1	q
1	0	0
1	1	1

Function Table

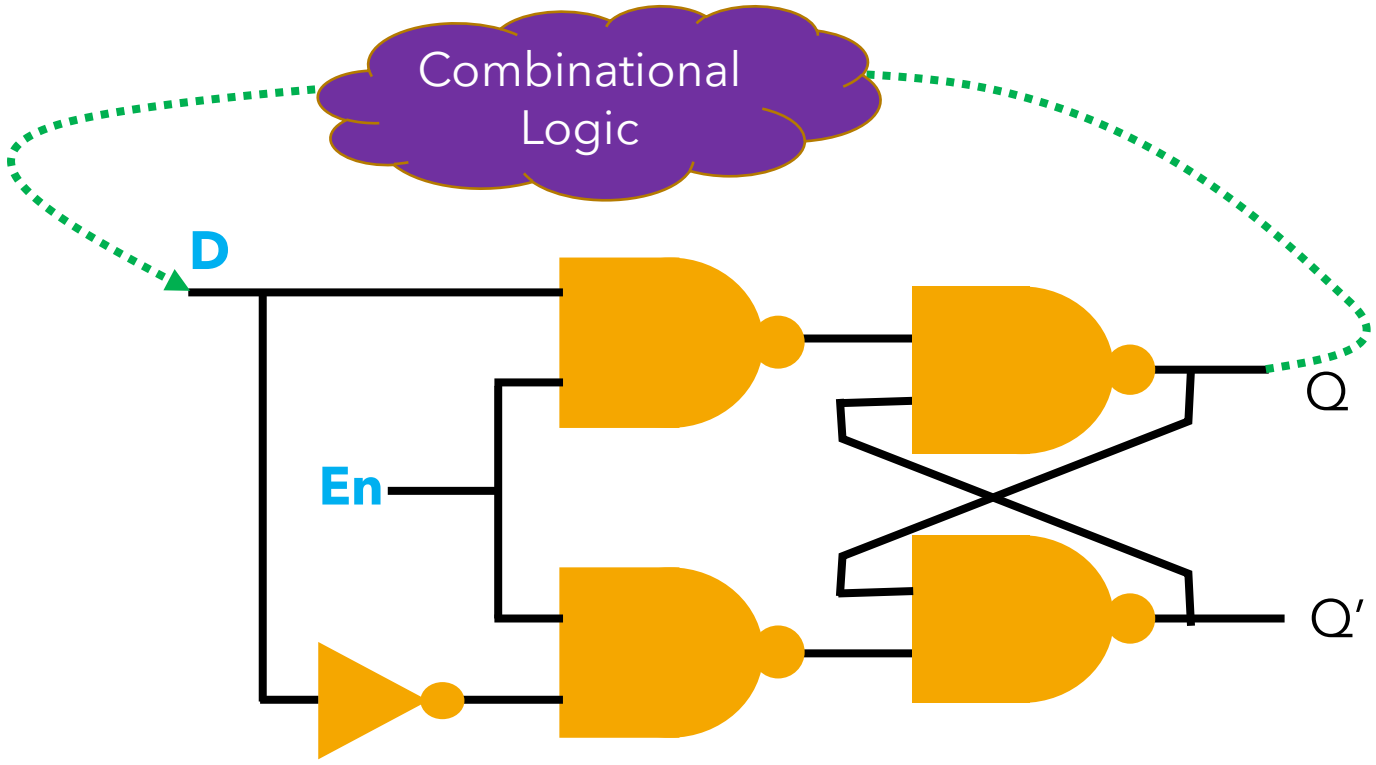
previous Q value

En	D	Q
0	x	q
1	x	D

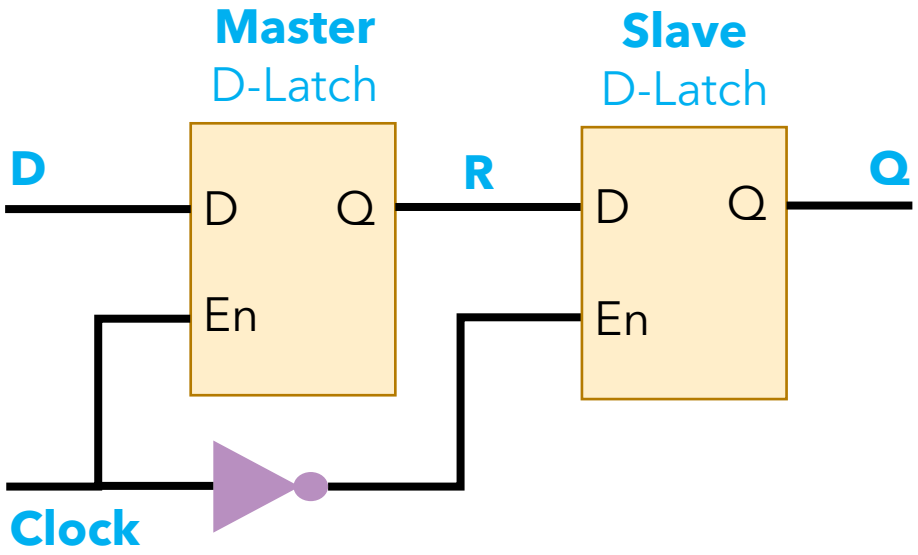
Function Table

Concerns about Latches

- **D** input might be function of output Q of same latch
  - **loop** in signal path
  - OK in sequential designs
- **En** might be **ON** for entire duration
  - ...causing **oscillations**
- **Safety:** in Latch-based designs, need to be careful of feedback/loop paths
  - e.g., use different En signals in different latch stages

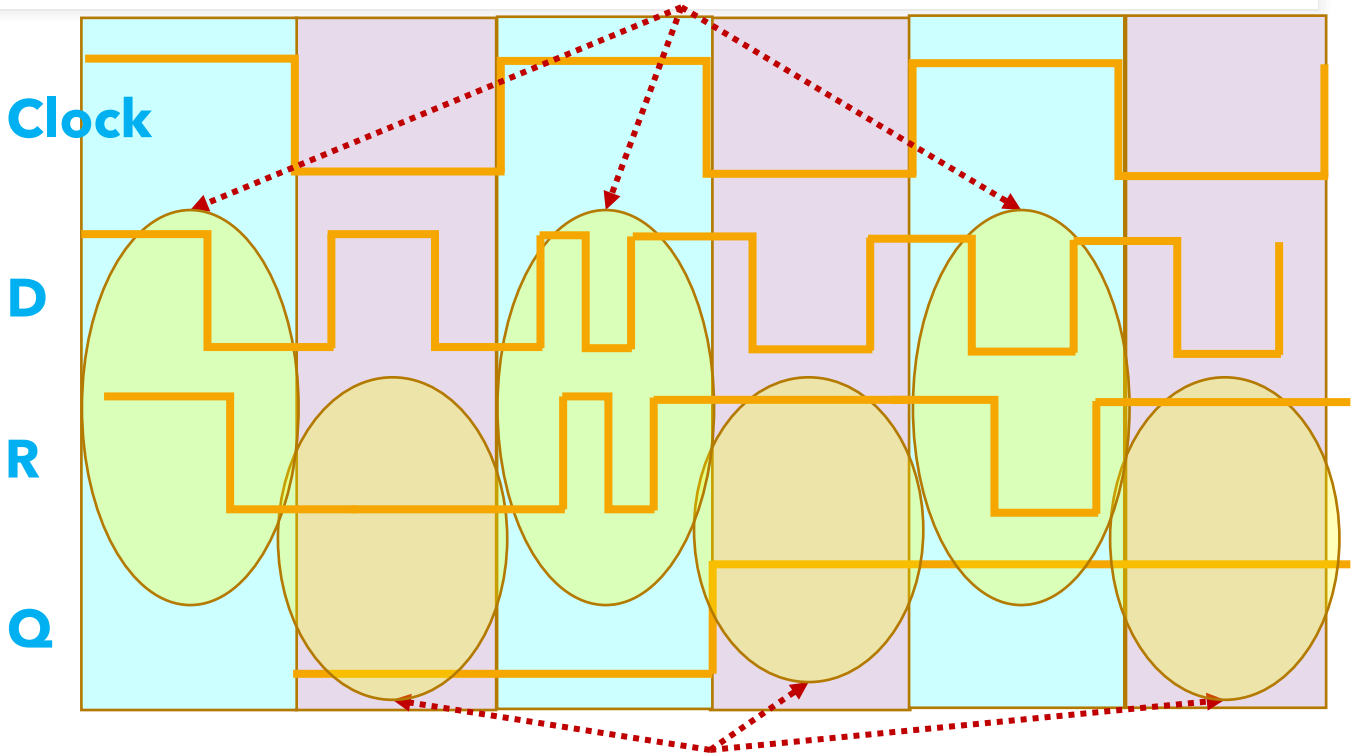


Master-Slave Flip-Flop



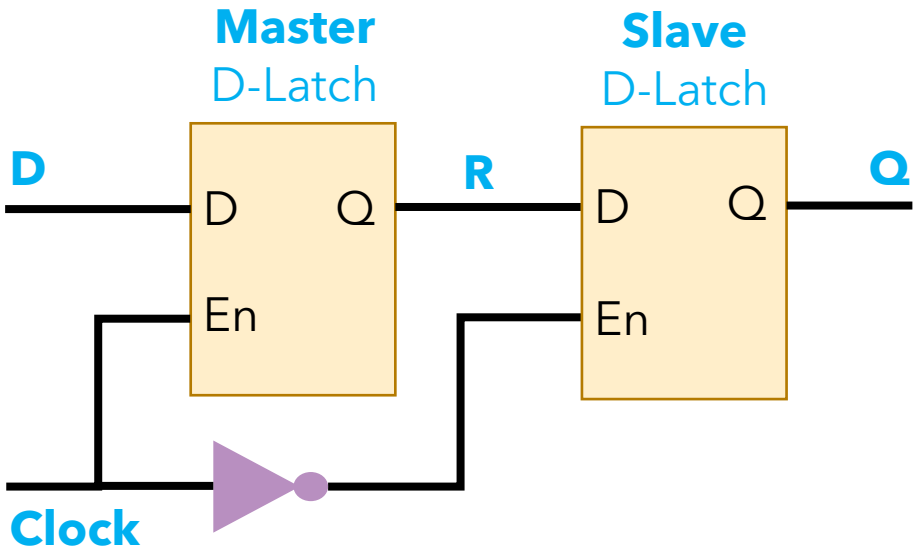
Clock=1: Master Enabled. Slave disabled.
Clock=0: Slave Enabled. Master disabled.

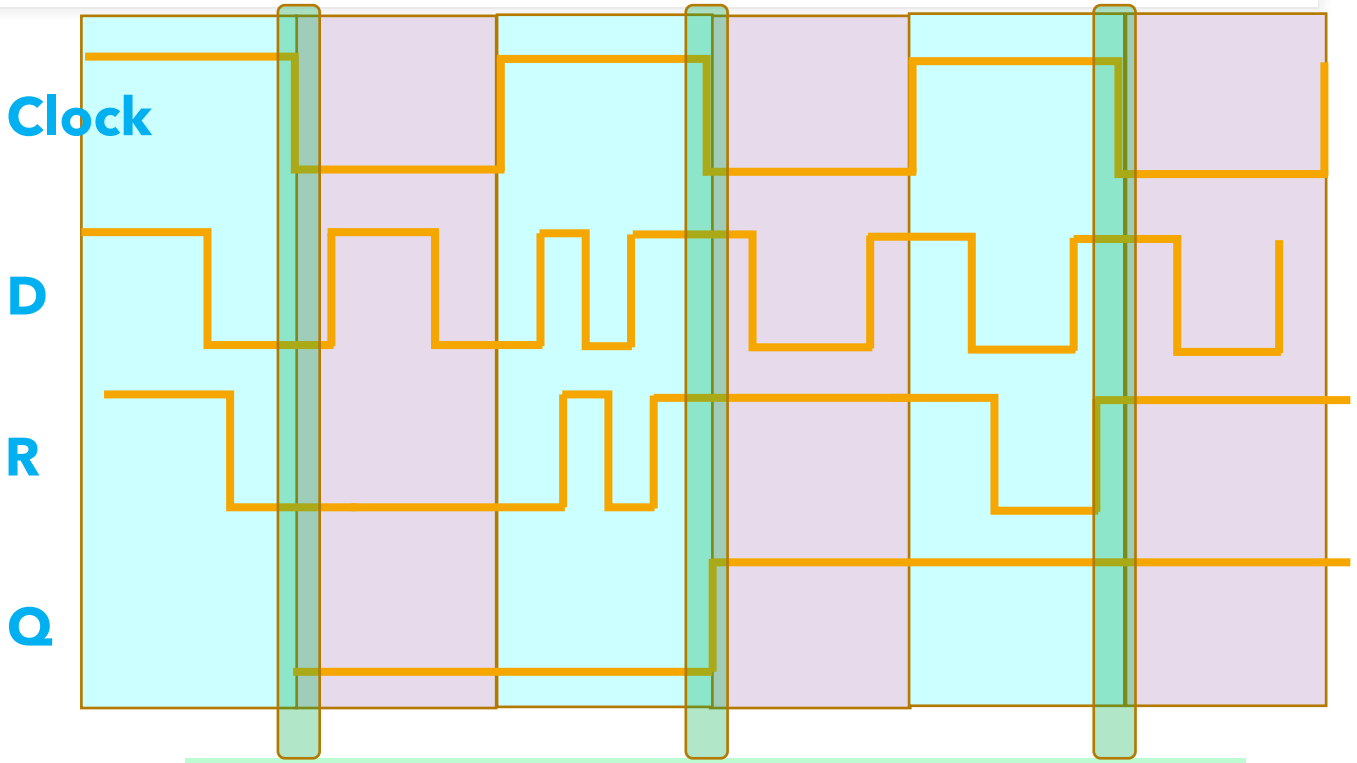
R follows **D** when **Clock=1 (Master Enabled)**



Q follows **R** when **Clock=0 (Slave Enabled)**

Master-Slave Flip-Flop

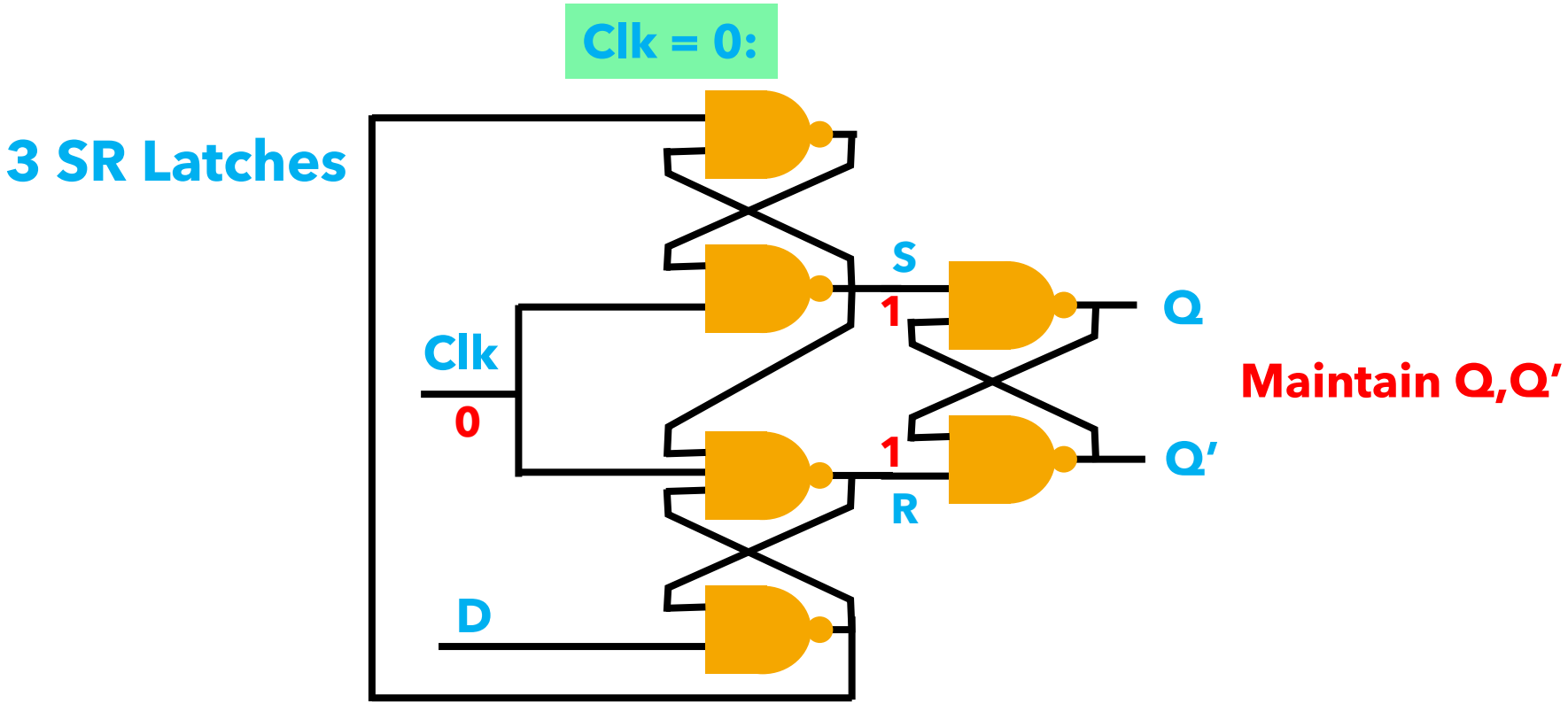




Clock=1: Master Enabled. Slave disabled.
Clock=0: Slave Enabled. Master disabled.

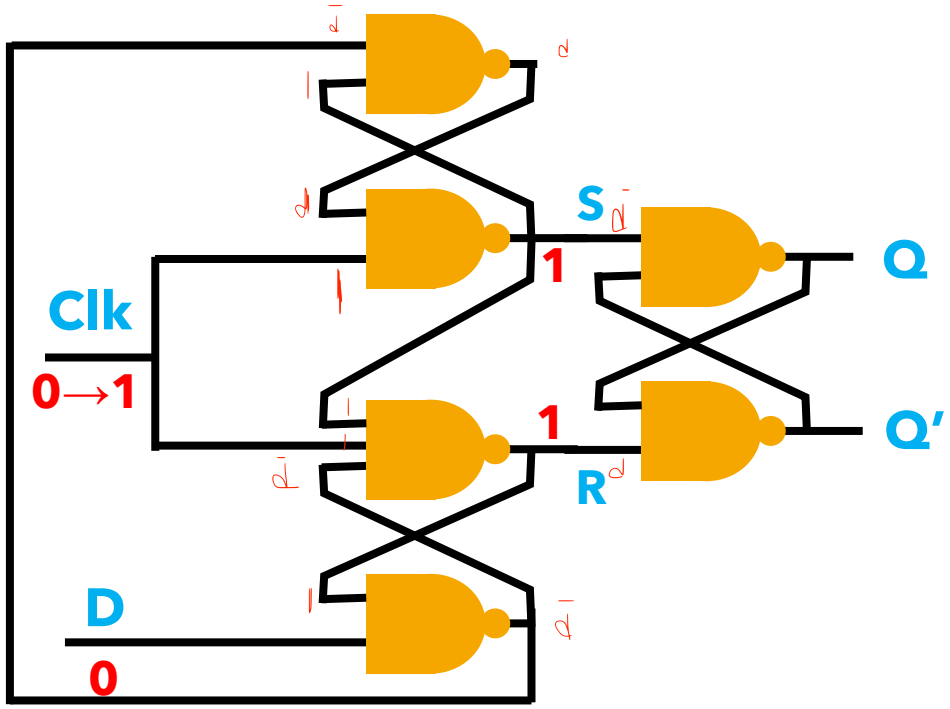
D→Q on Negative Edge of Clock

D Flip-flop: Alternative Design



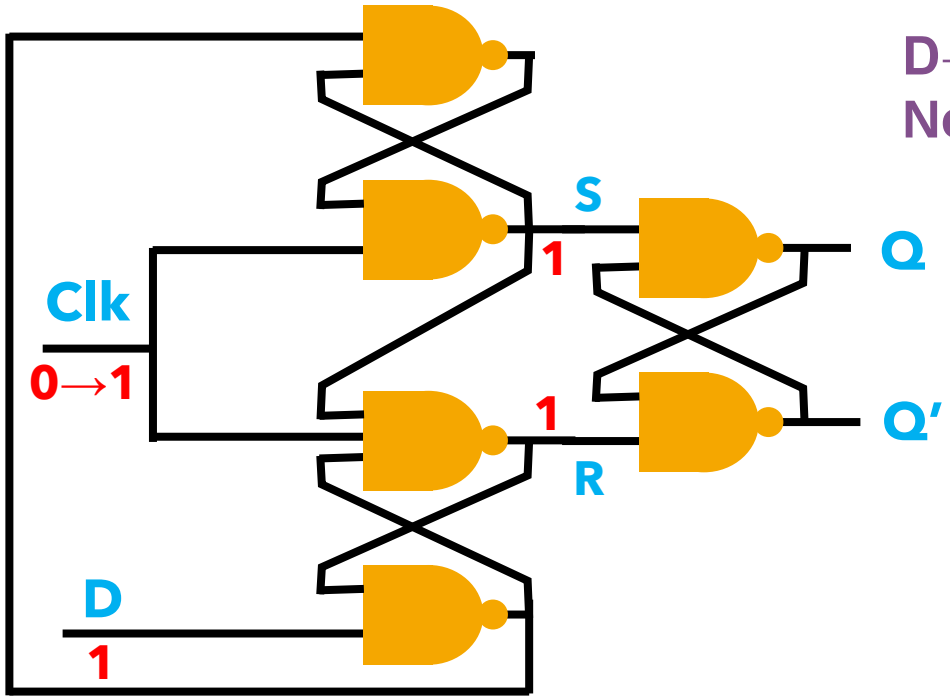
D Flip-flop: Alternative Design

Clk = 0→1, D = 0



Clk=0→1 causes R=1→0, Q=0
Further changes in D?

Clk = 0→1, D = 1



Clk=0→1 causes S=1→0, Q=1
Further changes in D?

D→Q on Clk edge
No changes later

Positive Edge-triggered Flip-flop

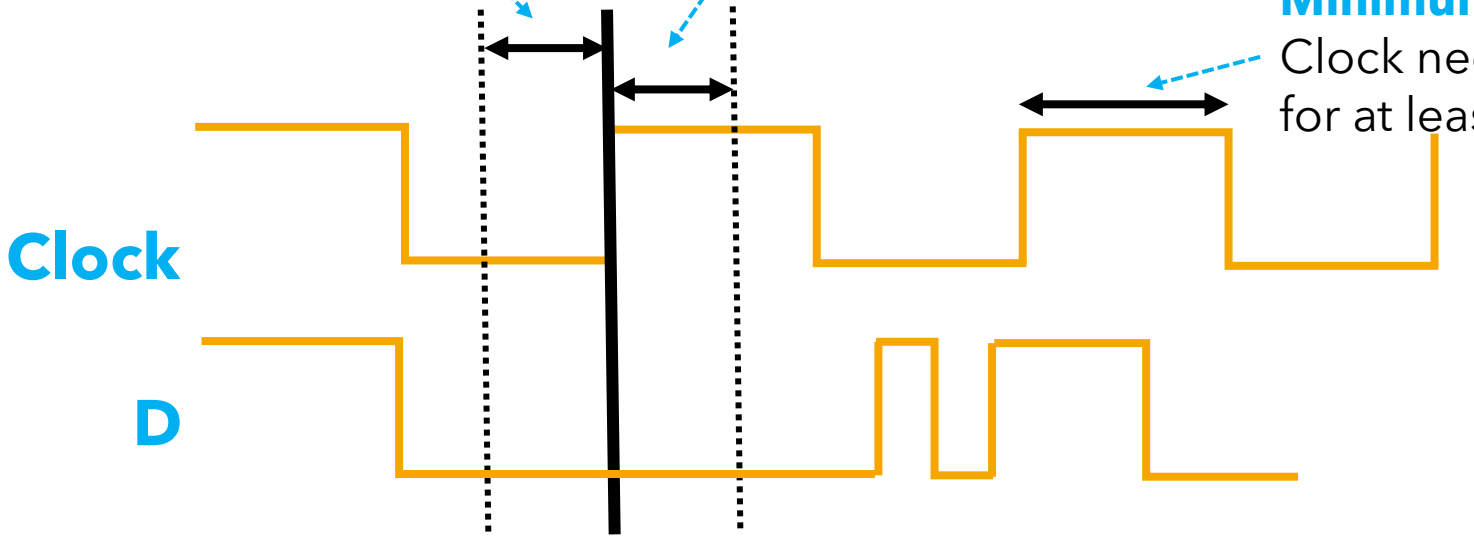
- D is transferred to Q at rising Clk edge
- No changes on falling Clk edge
- No changes in rest of Clk period

Flip-flop Timing Considerations

Setup Time
D needs to be constant for this duration **before clock edge**

Hold Time
D needs to be constant for this duration **after clock edge**

Minimum Pulse Width
Clock needs to be constant for at least this duration



Other flip-flop designs

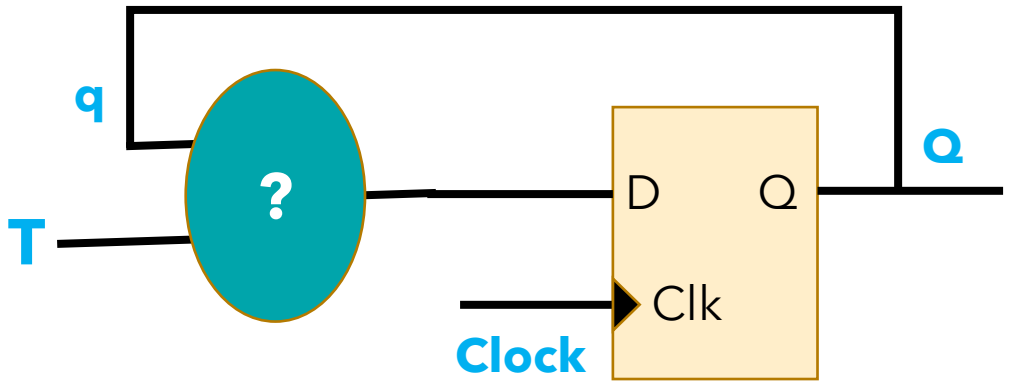
- T flip-flop
  - T=0: Q=q
  - T=1: Q=NOT q
- JK flip-flop
  - J=1,K=0: Q=1
  - J=0,K=1: Q=0
  - J=1,K=1: Q=NOT q
  - J=0,K=0: Q=q

T (Toggle) Flip-Flop

T Flip-flop Specification

On clock edge:

- T=0: Q=q — maintain state
- T=1: Q=NOT q — invert



T Flip-Flop implemented using D Flip-flop

T Flip-flop Specification

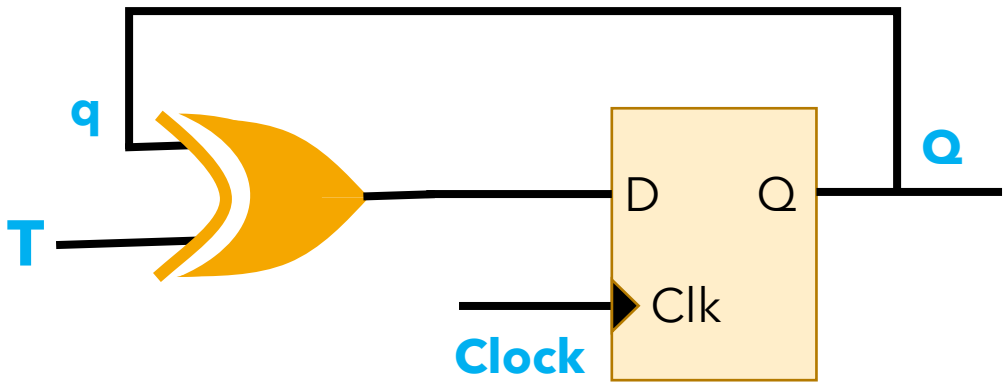
- T=0: Q=q
- T=1: Q=NOT q

T	q	Q
0	0	0
0	1	1
1	0	1
1	1	0

Q=q

Q= NOT q

Function Table



Q = q ⊕ T

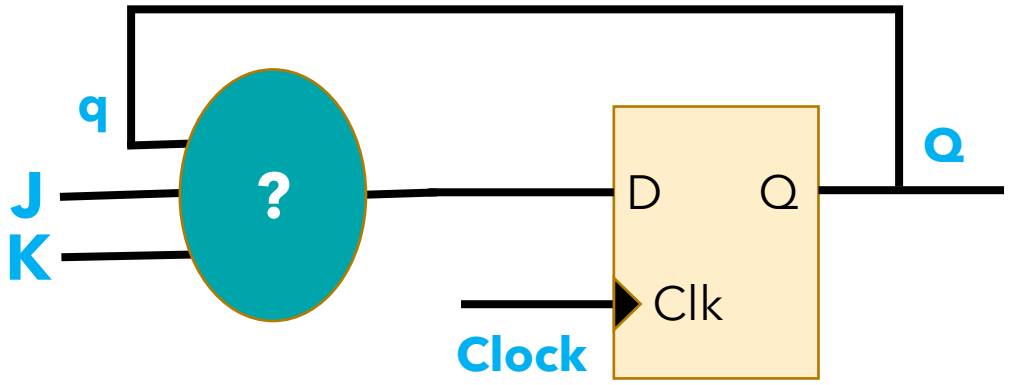
On clock edge: **Q** takes the value of **q ⊕ T**
(q ⊕ T) is sent to D input

JK Flip-Flop

Specification

On clock edge:

- J=1,K=0: Q=1
- J=0,K=1: Q=0
- J=1,K=1: Q=NOT q
- J=0,K=0: Q=q



JK Flip-Flop implemented using D Flip-flop

Function Table

J	K	q	Q	
0	0	0	0	Q=q
0	0	1	1	
0	1	0	0	Q= 0
0	1	1	0	
1	0	0	1	Q=1
1	0	1	1	
1	1	0	1	Q=NOT q
1	1	1	0	

K-map for Q

q \ JK	00	01	11	10
0	0	0	1	1
1	1	0	0	1

Jq'

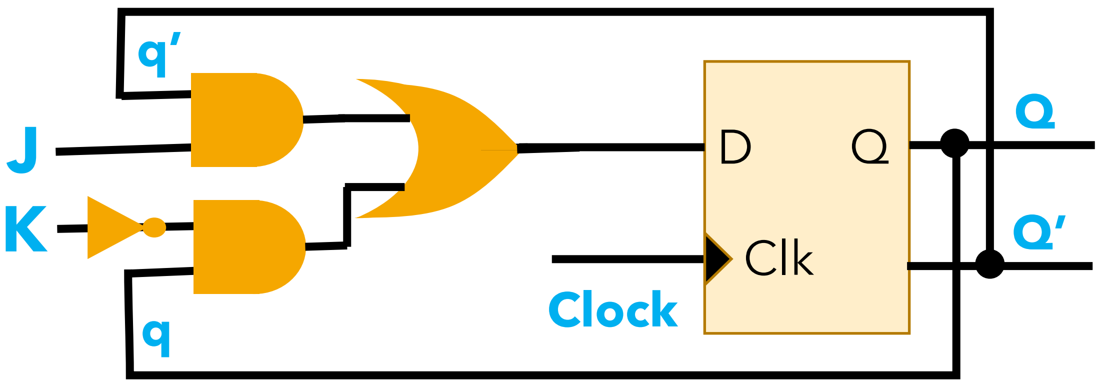
K'q

Q = Jq' + K'q

Specification

On clock edge:

- J=1,K=0: Q=1
- J=0,K=1: Q=0
- J=1,K=1: Q=NOT q
- J=0,K=0: Q=q



On clock edge: **Q** takes the value of **Jq' + K'q**

This is sent to D input

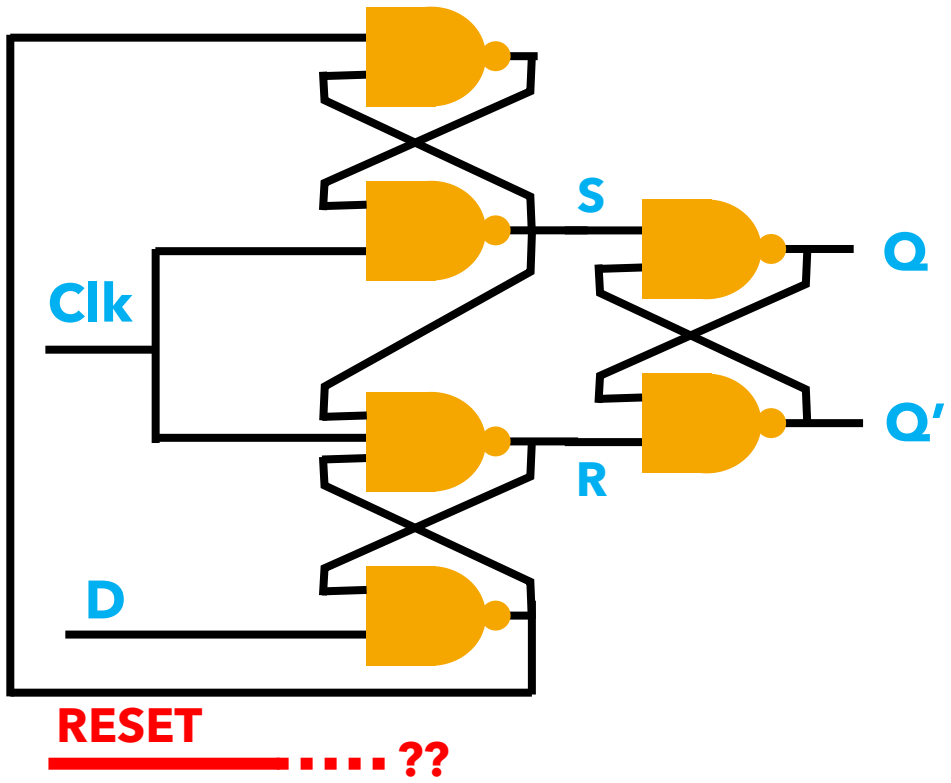
Latches vs. Flip-flops

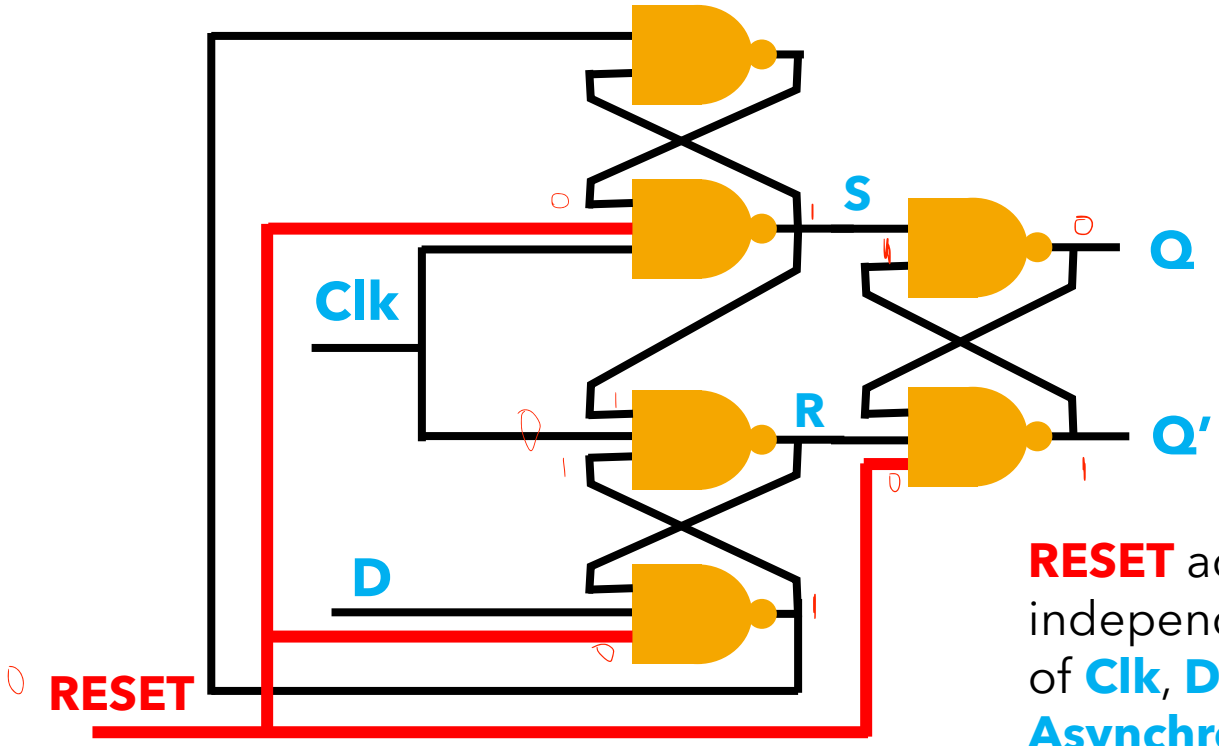
- Latches are **Level-triggered**
  - Q follows D when **En=1 (level = 1)**
- Flip-flops are **Edge-triggered**
  - D value captured **on clock edge**
  - Feedback/loop is eliminated

D Flip-flop: Asynchronous Inputs

New Input RESET:

RESET=0 causes Q=0





RESET acts independently of **Clk**, **D**

Asynchronous Reset

Characteristic Tables: Flip-flop Properties

D	Q (t+1)	
0	0	Reset
1	1	Set

D Flip-flop Characteristic Table

T	Q (t+1)	
0	Q(t)	No change
1	Q′(t)	Complement

T Flip-flop Characteristic Table

J	K	Q (t+1)	
0	0	Q(t)	No change
0	1	0	Reset
1	0	1	Set
1	1	Q′(t)	Complement

JK Flip-flop Characteristic Table

Characteristic Equations

D	Q (t+1)	
0	0	Reset
1	1	Set

D Flip-flop
Q (t+1) = D

T	Q (t+1)	
0	Q(t)	No change
1	Q′(t)	Complement

T Flip-flop
Q (t+1) = Q ⊕ T

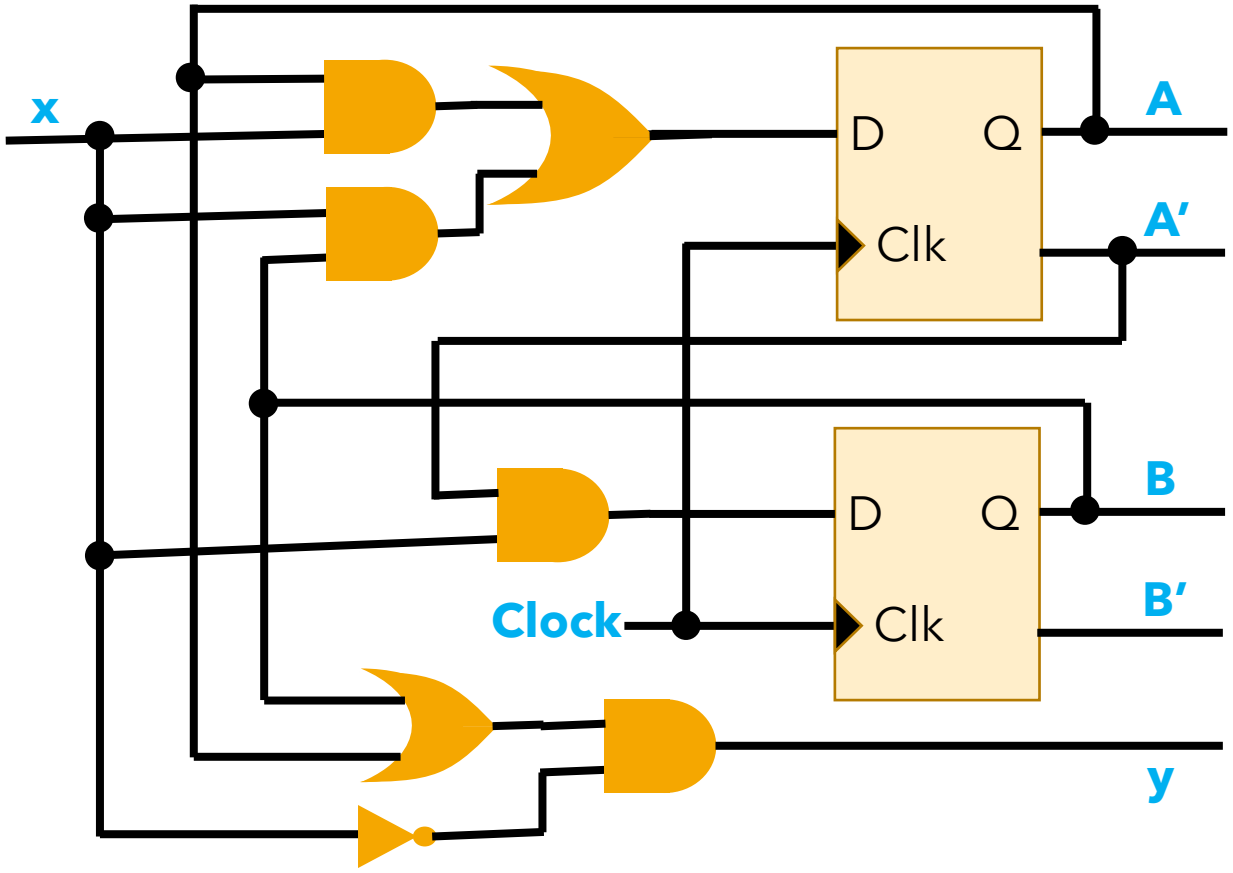
J	K	Q (t+1)	
0	0	Q(t)	No change
0	1	0	Reset
1	0	1	Set
1	1	Q′(t)	Complement

JK Flip-flop
Q (t+1) = Q′J + QK′

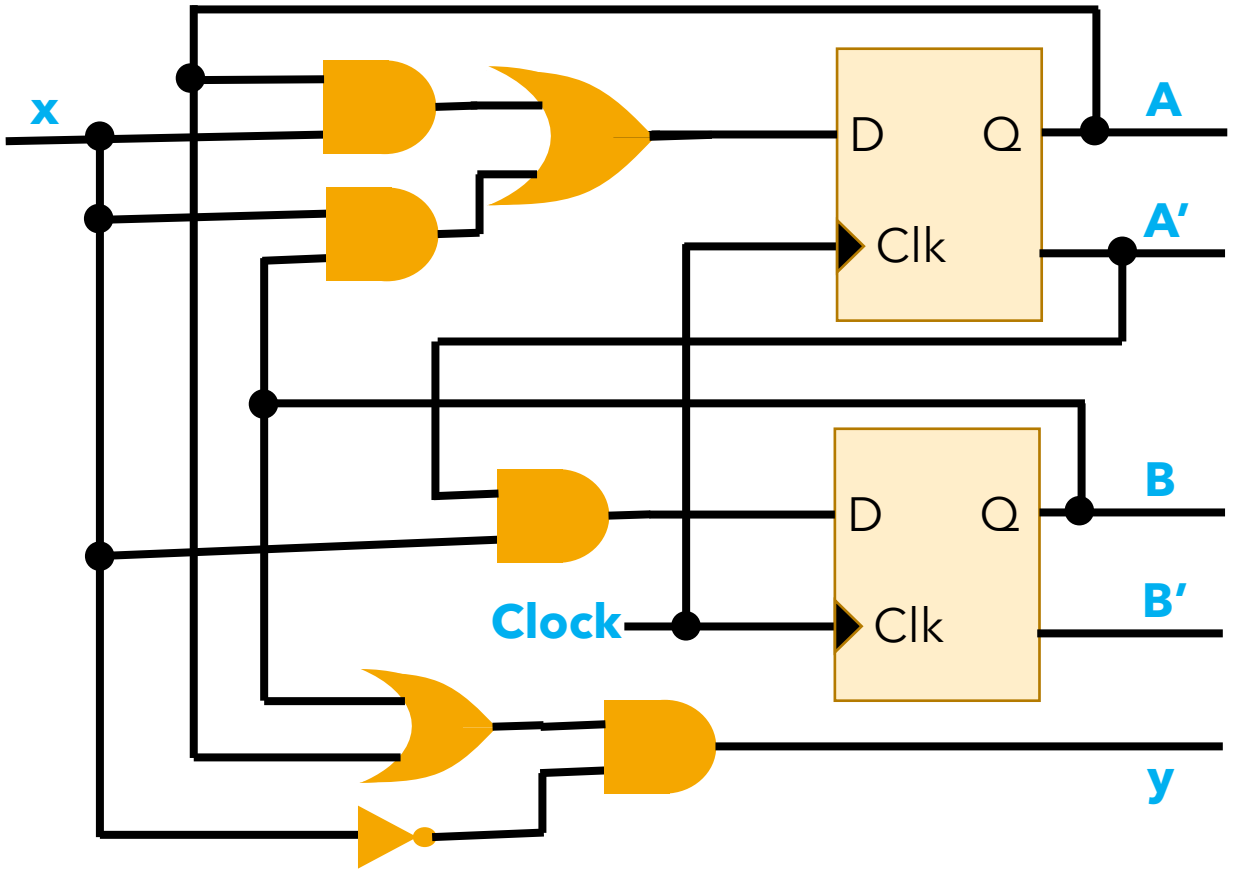
Q (= Q(t)): Value of Q just before clock edge
Clock is implicit (not written in equation)

Analysing Clocked Sequential Circuits

- All clocked sequential circuits have:
  - **Inputs**
  - **Outputs**
  - Combinational **logic**
    - **functions of inputs and Qs**
  - A set of **flip-flops** (any type)
  - A **clock** (common to all flip-flops)



Deriving State Equations



Equations for new values of A, B, y:

A(t+1) = A(t) x (t) + B (t) x (t)

B(t+1) = A' (t) x (t)

y = x' (t)(A (t) +B (t))

Simplify: Drop the (t)

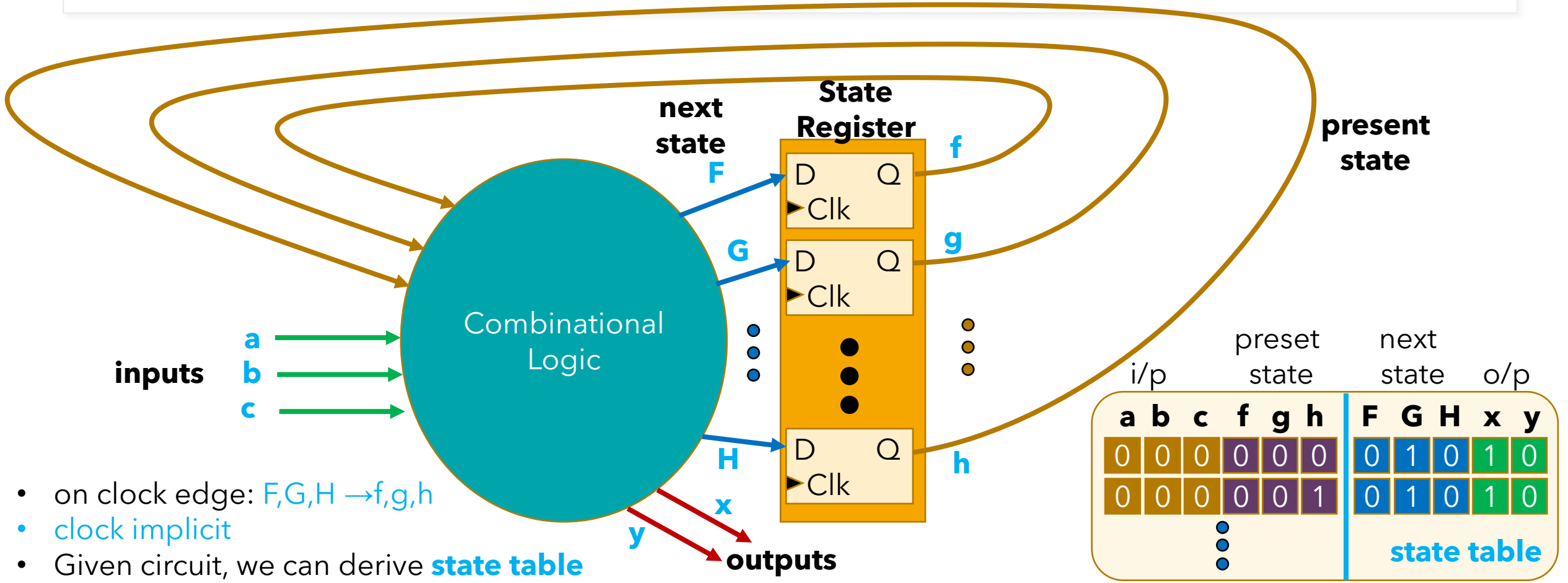
A(t+1) = Ax + Bx

B(t+1) = A'x

y = x'(A+B)

Generates A, B, y values in next clock cycle in terms of values in current clock cycle

Sequential Designs

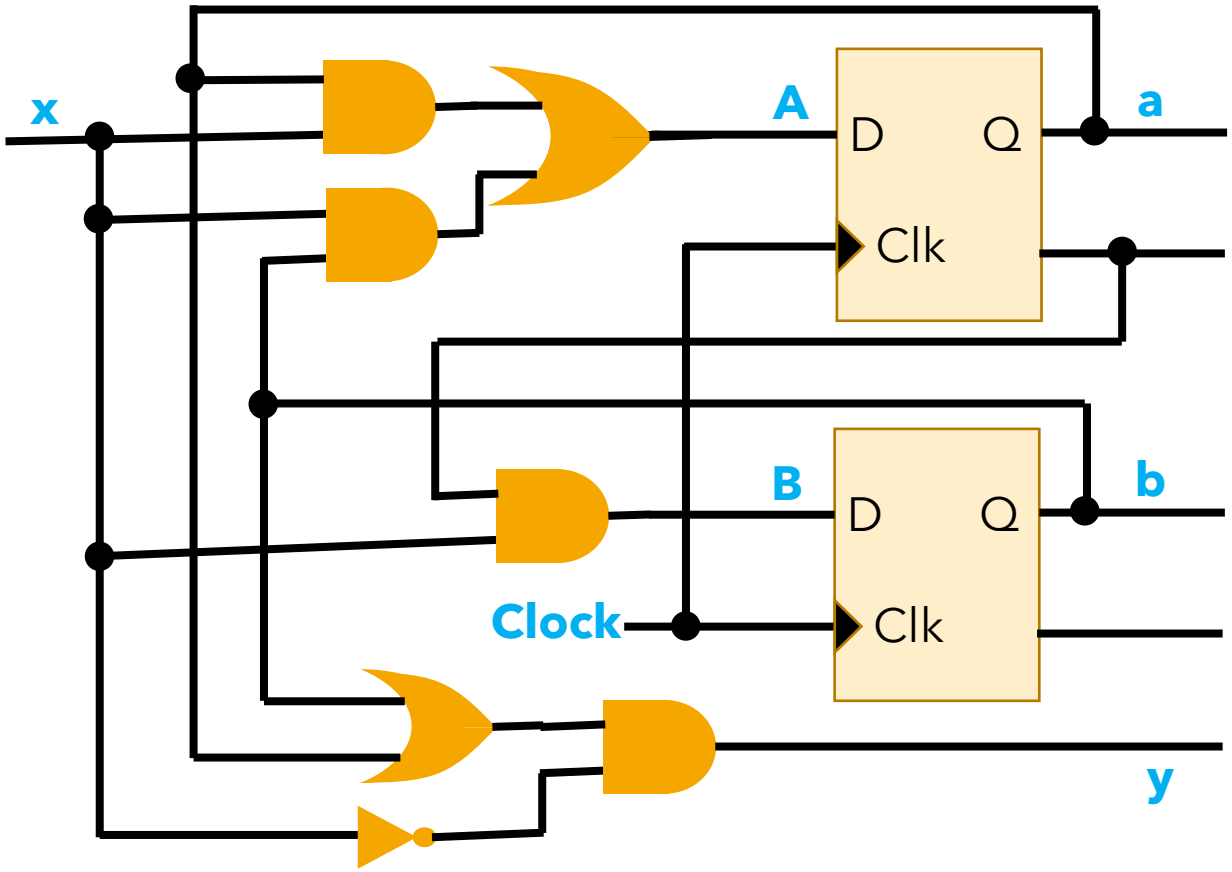


i/p			preset state			next state			o/p	
a	b	c	f	g	h	F	G	H	x	y
0	0	0	0	0	0	0	1	0	1	0
0	0	0	0	0	1	0	1	0	1	0

state table

- on clock edge: F,G,H →f,g,h
- clock implicit
- Given circuit, we can derive **state table**
- Given state table, can derive **circuit**

Deriving State Table



State Table

a	b	x	A	B	y
0	0	0	0	0	0
0	0	1	0	1	0
0	1	0	0	0	1
0	1	1	1	1	0
1	0	0	0	0	1
1	0	1	1	0	0
1	1	0	0	0	1
1	1	1	1	0	0

State Diagram (Finite State Machine - FSM)

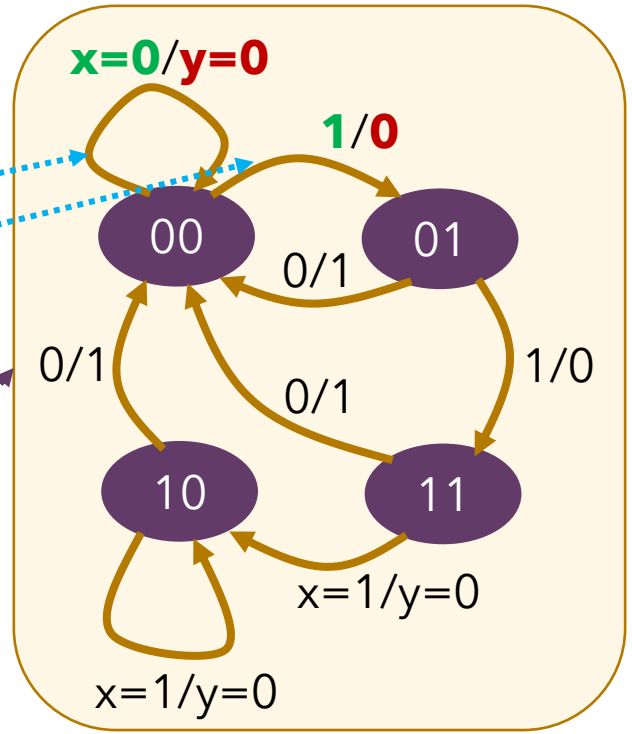
Q values of Flip-flops together constitute the **State** of the system
Our design has 4 **states**: **00**, **01**, **10**, **11** **(ab/AB values)**

State Table

present state a	present state b	x	next state A	next state B	y
0	0	0	0	0	0
0	0	1	0	1	0
0	1	0	0	0	1
0	1	1	1	1	0
1	0	0	0	0	1
1	0	1	1	0	0
1	1	0	0	0	1
1	1	1	1	0	0

Same Information

State Diagram

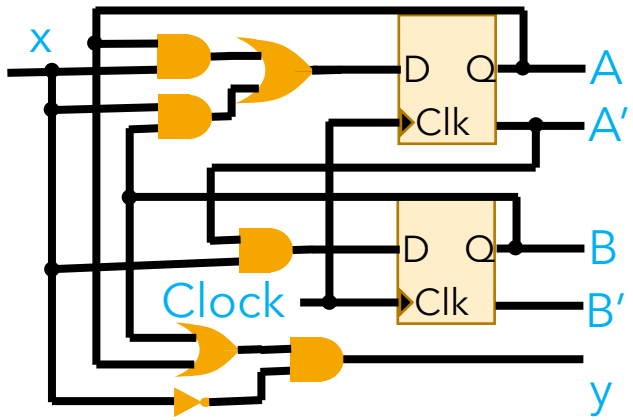


State Transition

Each **Row** in State Table becomes: one **Edge/State Transition** in FSM

Sequential Analysis

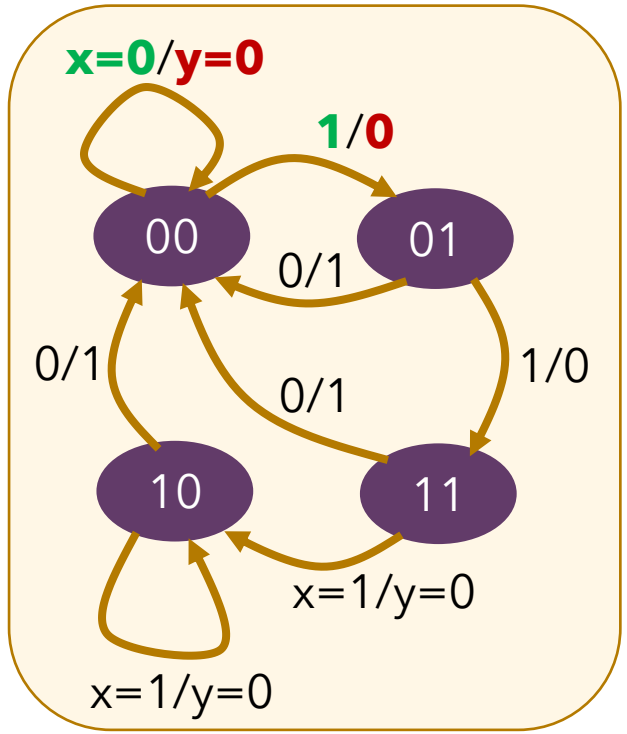
Sequential Design



State Table

present state a	b	x	next state A	B	y
0	0	0	0	0	0
0	0	1	0	1	0
0	1	0	0	0	1
0	1	1	1	1	0
1	0	0	0	0	1
1	0	1	1	0	0
1	1	0	0	0	1
1	1	1	1	0	0

State Diagram



Synthesis

Example Finite State Machine

Specification: Counter

Output P = 1 every 3rd clock cycle
Output P = 0 in other cycles
If input R = 0, reset count to zero
If input R = 1, resume counting

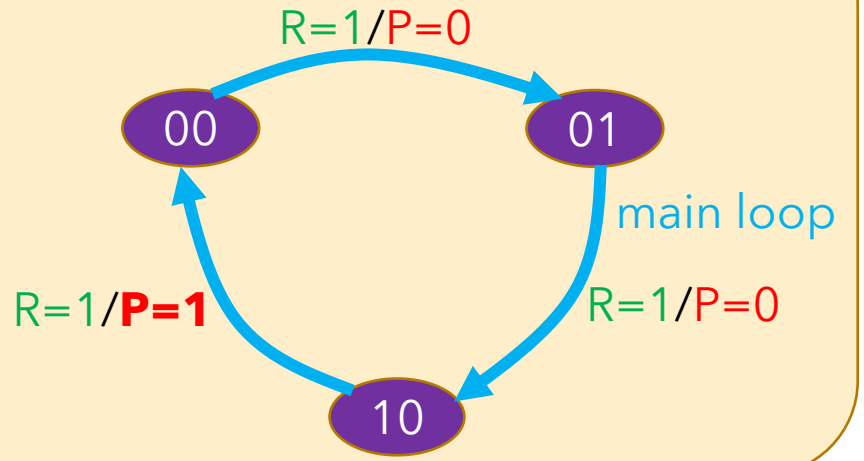
Strategy

- Let us count: 0, 1, 2, 0, 1, 2, 0, 1, 2....
- Every time we hit 0, set P=1

States

- 00 — 1st cycle
- 01 — 2nd cycle
- 10 — 3rd cycle

FSM (States + Transitions)



Example Finite State Machine

Specification: Counter

Output P = 1 every 3rd clock cycle
Output P = 0 in other cycles
If input R = 0, reset count to zero
If input R = 1, resume counting

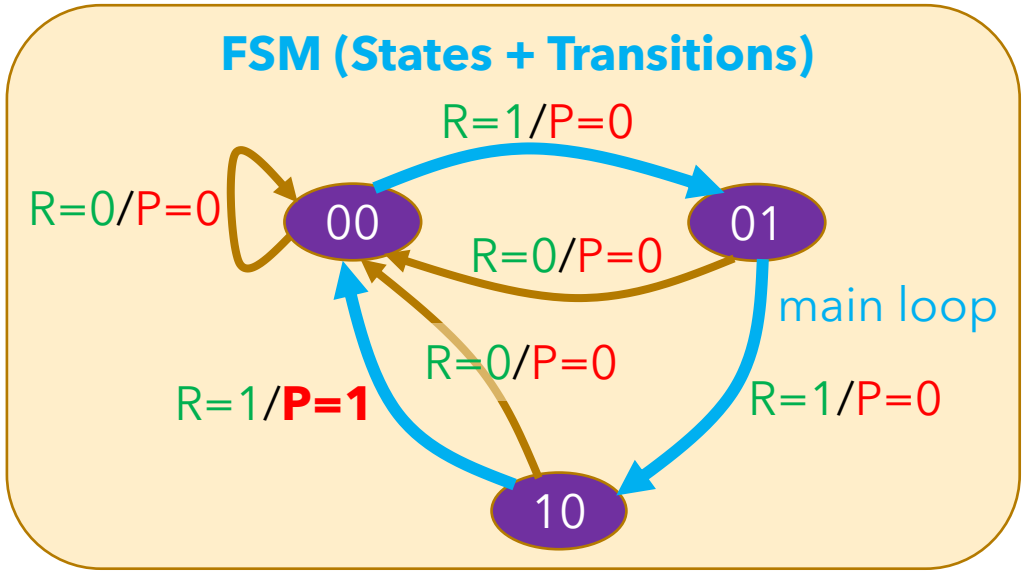
Strategy

- Let us count: 0, 1, 2, 0, 1, 2, 0, 1, 2....
- Every time we hit 0, set P=1

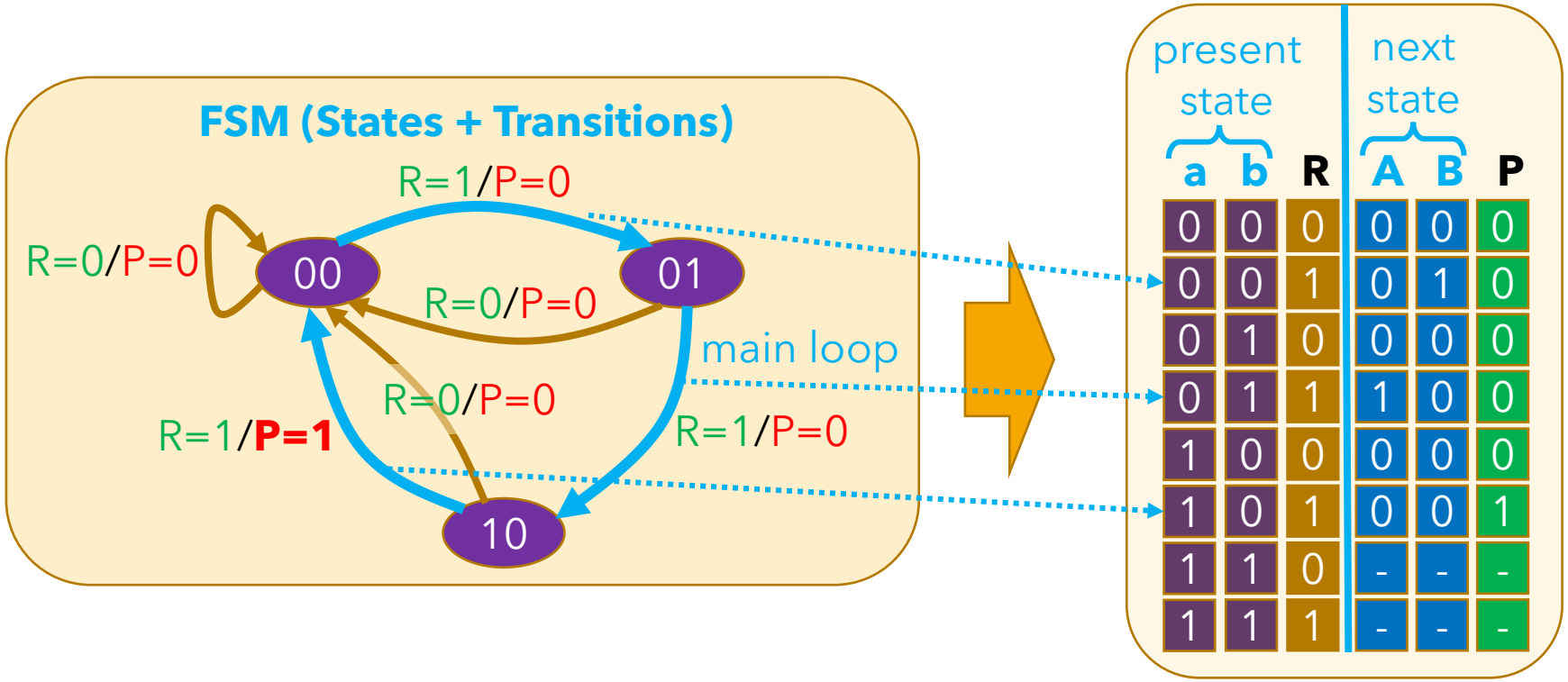
States

- 00 — 1st cycle
- 01 — 2nd cycle
- 10 — 3rd cycle

Assume: P=0 upon reset

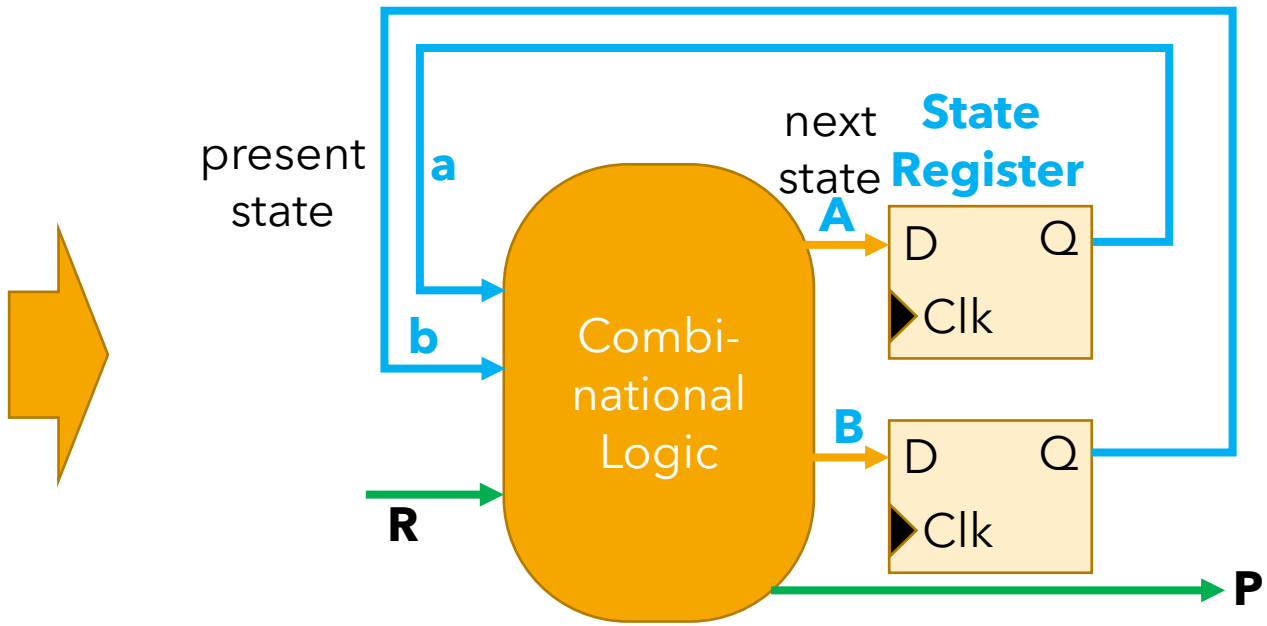


Deriving State Table from FSM



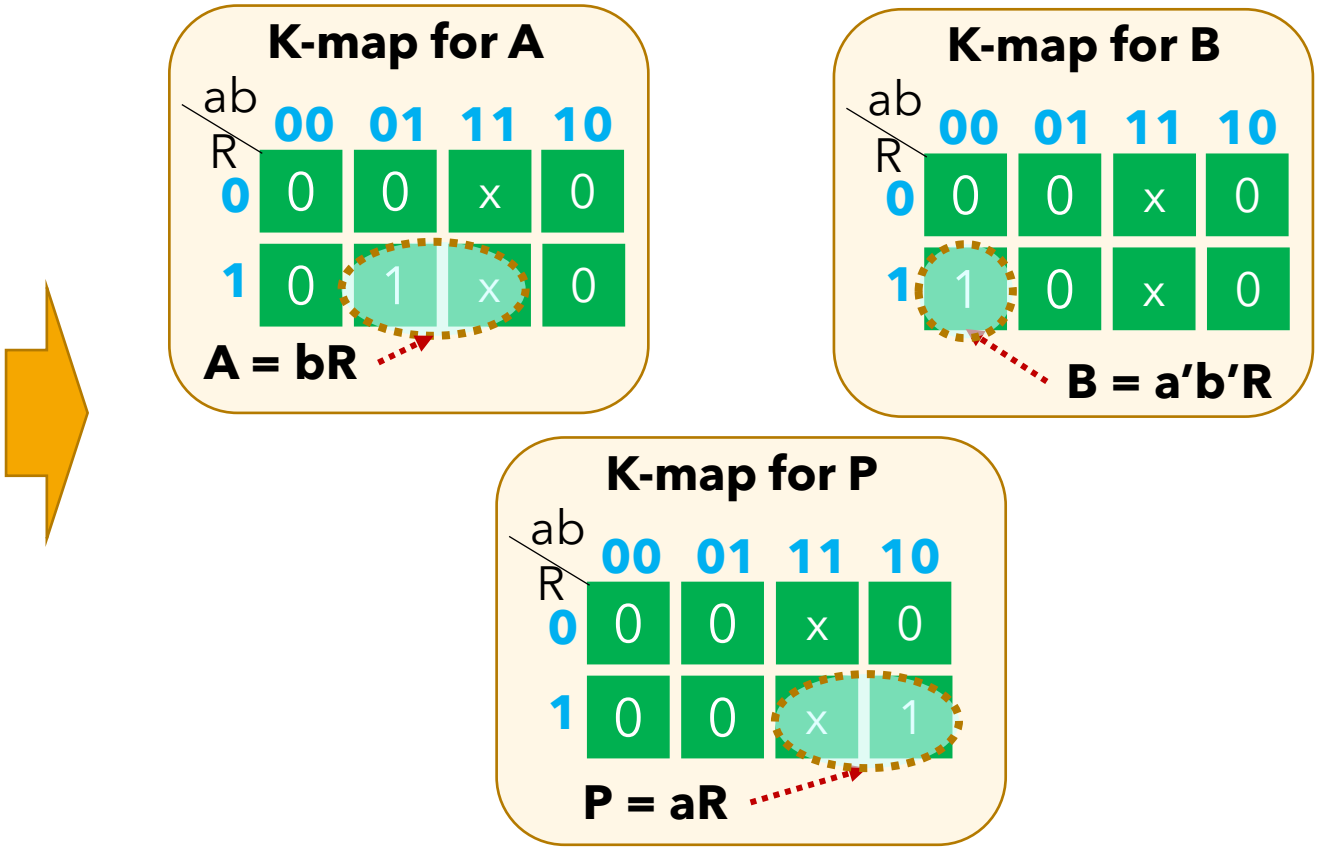
Deriving Sequential Design from State Table

present state			next state		
a	b	R	A	B	P
0	0	0	0	0	0
0	0	1	0	1	0
0	1	0	0	0	0
0	1	1	1	0	0
1	0	0	0	0	0
1	0	1	0	0	1
1	1	0	-	-	-
1	1	1	-	-	-

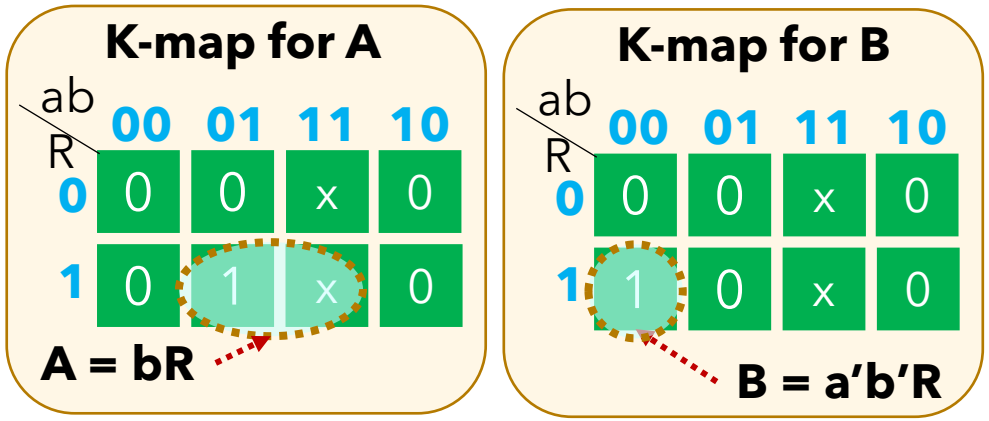


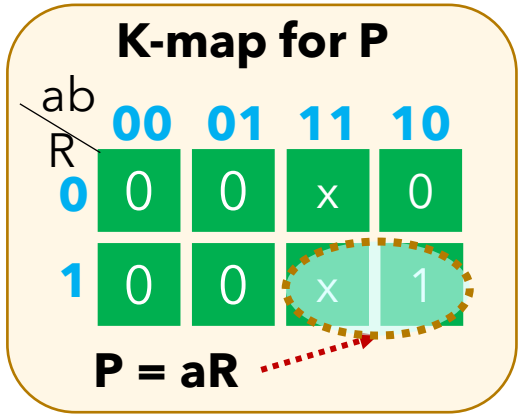
Boolean Expressions for Next State, Output

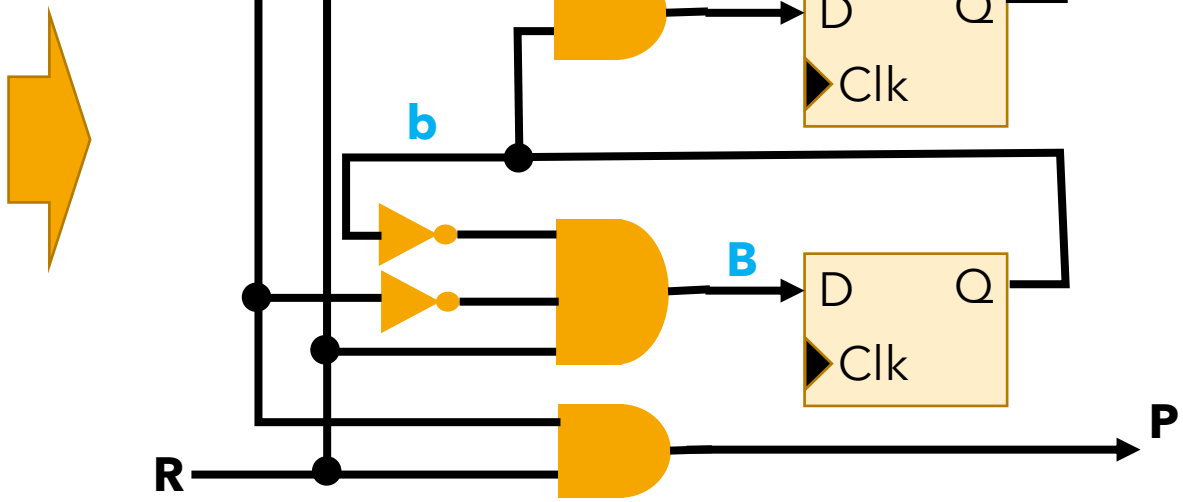
present state			next state		
a	b	R	A	B	P
0	0	0	0	0	0
0	0	1	0	1	0
0	1	0	0	0	0
0	1	1	1	0	0
1	0	0	0	0	0
1	0	1	0	0	1
1	1	0	-	-	-
1	1	1	-	-	-



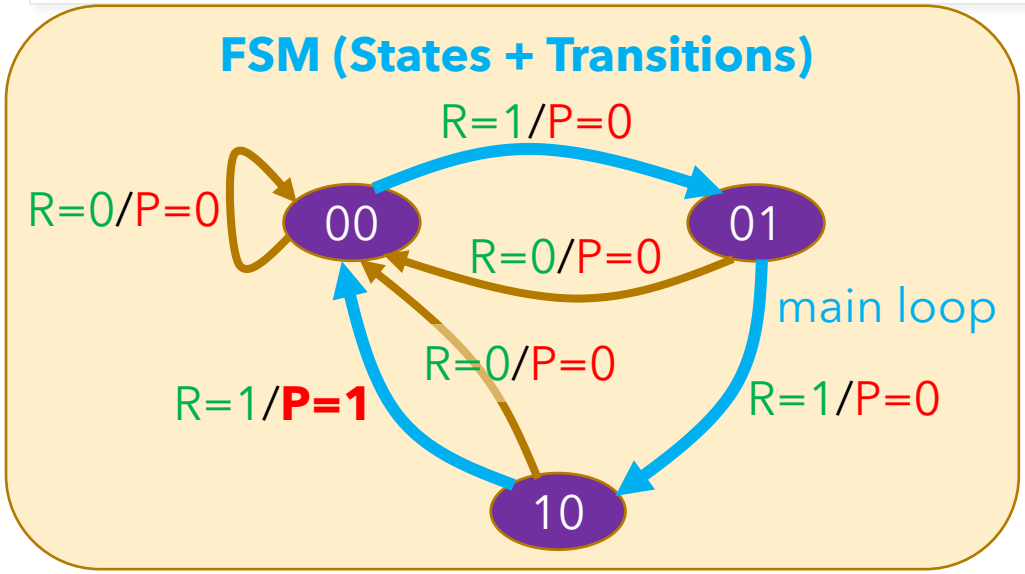
Gate-level Design



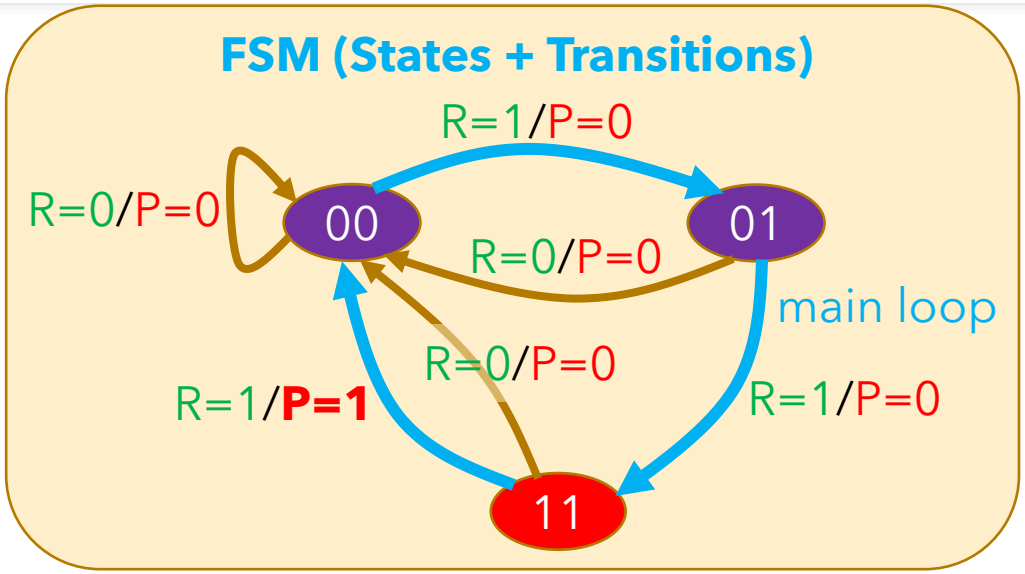




What if we used a different sequence?



Replace
00 by **11**



What changes?
What remains unchanged?
Is the functionality still delivered?

Specification

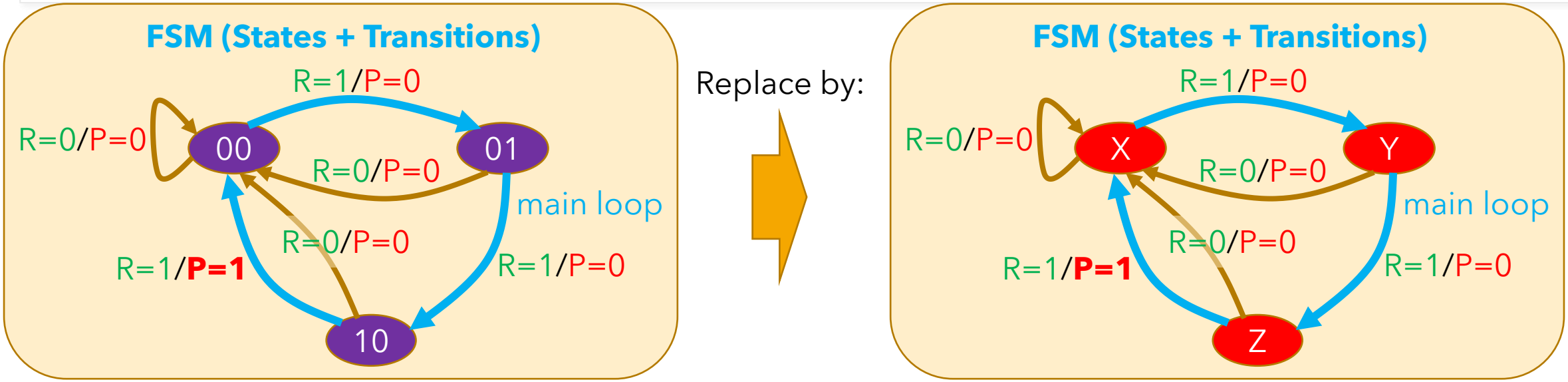
Output P = 1 every 3rd clock cycle
Output P = 0 in other cycles
If input R = 0, reset count to zero
If input R = 1, resume counting

Sequence of states: **0,1,3,0,1,3,...**

Output P sequence **unchanged**!

Specification still met.
Values **00**, **01**, etc., don't matter

FSM States can be Symbolic

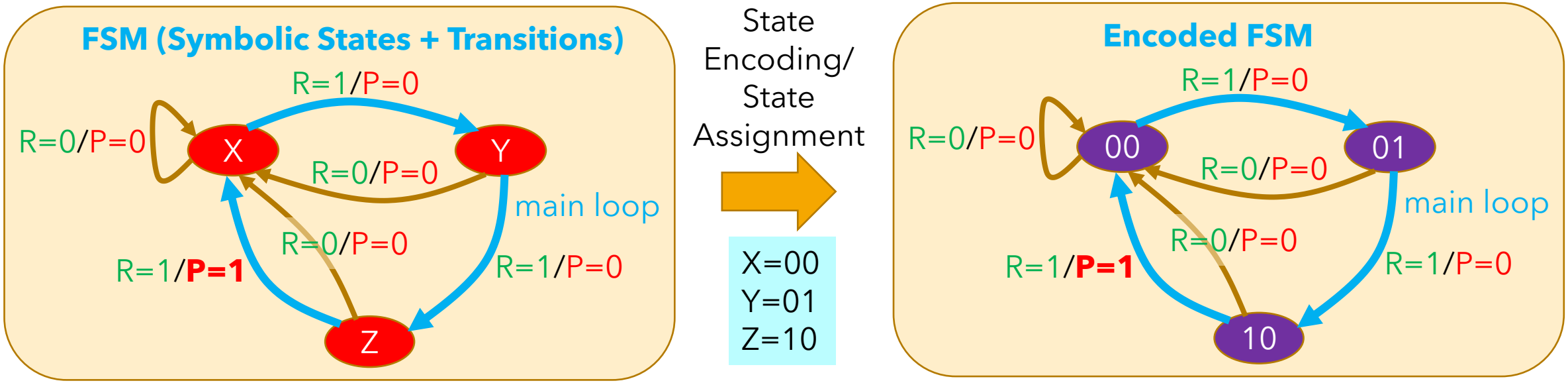


Specification still met.
Values **00**, **01**, etc., don't matter
States can be **Symbolic** (labelled **X**, **Y**, **Z**)

Start with Symbolic FSM

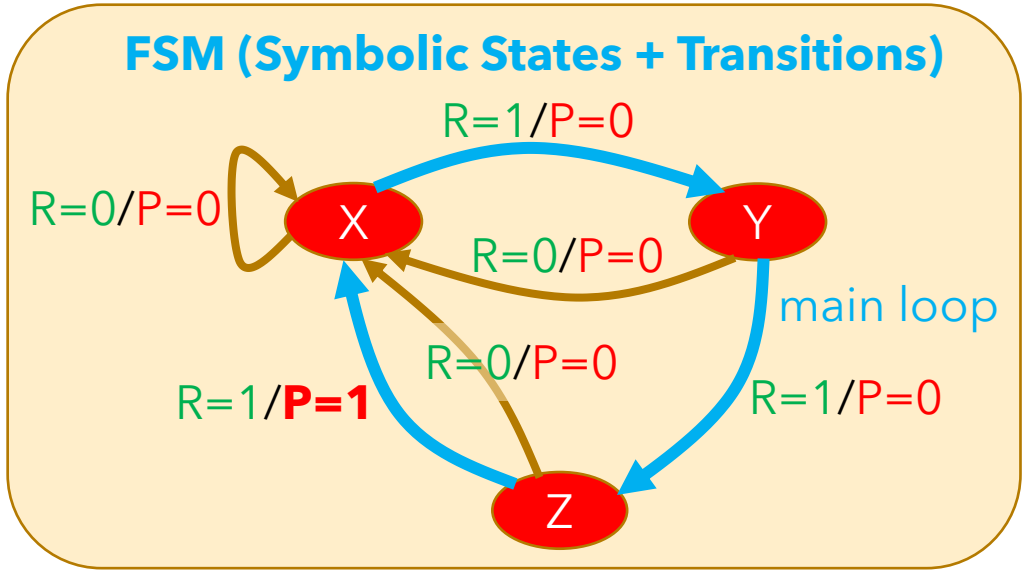
Specification

Output P = 1 every 3rd clock cycle
Output P = 0 in other cycles
If input R = 0, reset count to zero
If input R = 1, resume counting



Other encodings also valid

Symbolic State Table



Symbolic State Table

PS	R	NS	P
X	0	X	0
X	1	Y	0
Y	0	X	0
Y	1	Y	0
Z	0	X	0
Z	1	X	1

PS: present state
NS: next state

Encoded State Table (Truth Table)

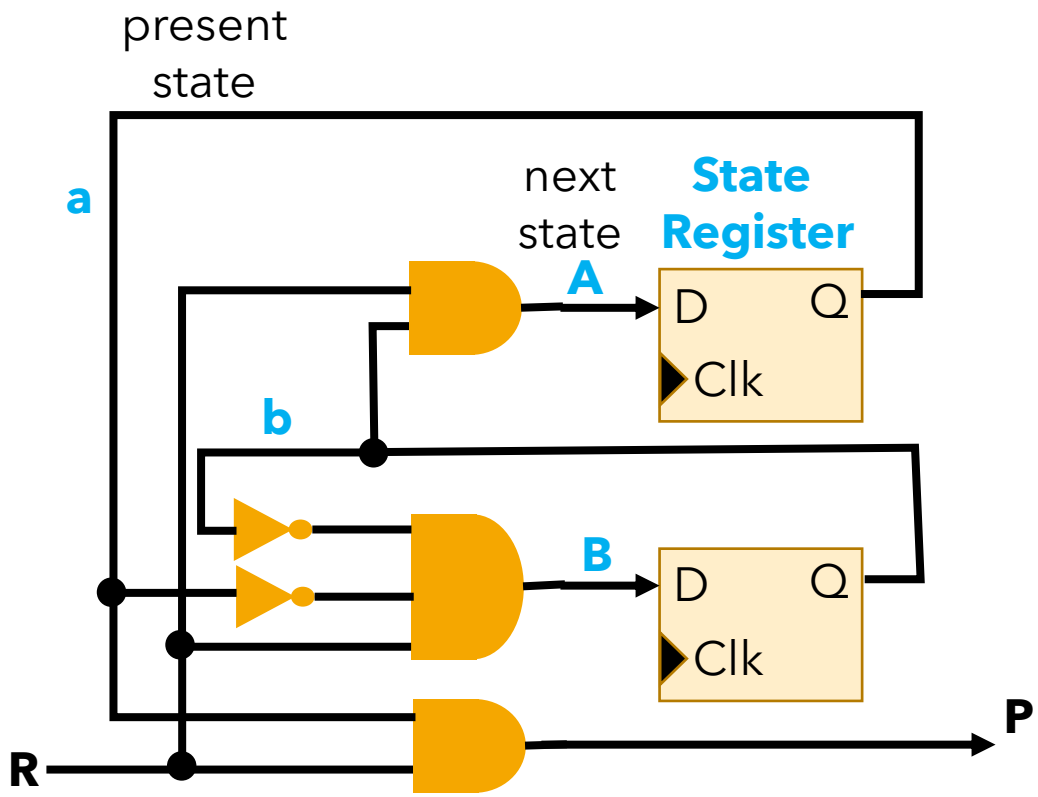
present state			next state		
a	b	R	A	B	P
0	0	0	0	0	0
0	0	1	0	1	0
0	1	0	0	0	0
0	1	1	1	0	0
1	0	0	0	0	0
1	0	1	0	0	1

No need for don't care rows

Finally, Sequential Circuit

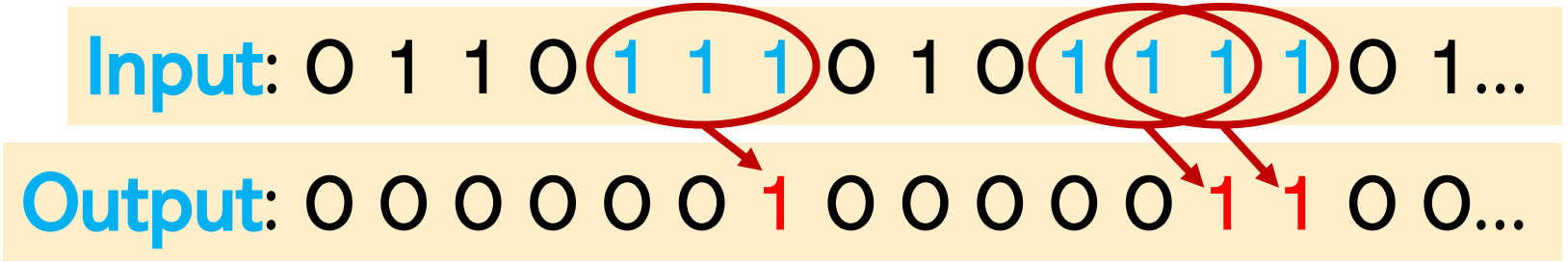
Encoded State Table (Truth Table)

present state			next state		
a	b	R	A	B	P
0	0	0	0	0	0
0	0	1	0	1	0
0	1	0	0	0	0
0	1	1	1	0	0
1	0	0	0	0	0
1	0	1	0	0	1



Another FSM Example

- Design a Sequence Detector
  - Detect a sequence of 3 or more 1's on an input line



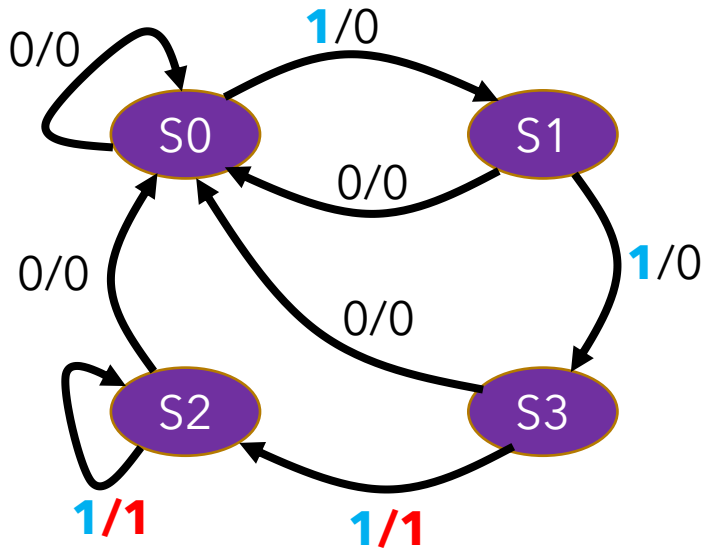
One bit per clock

- Design FSM
  - Inputs? Outputs? States? Transitions?

FSM Design

Input: 0 1 1 0 1 1 1 0 1 0 1 1 1 1 0 1

Output: 0 0 0 0 0 0 1 0 0 0 0 0 1 1 0 0



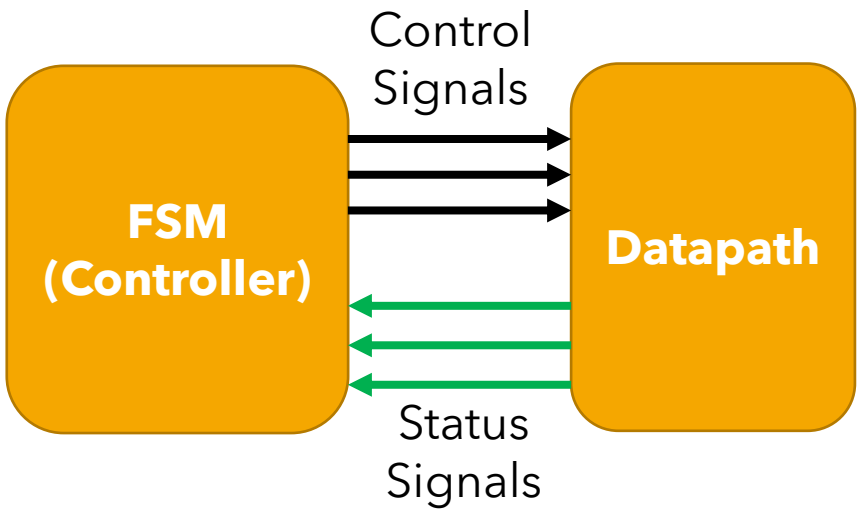
Style: **Mealy Machine**
Output is a function of both **state** and **input.**
Output is associated with transition.

Alternative Style: **Moore Machine**
Output is a function of **state** only
(Ref: Textbook. Figure 5.27)

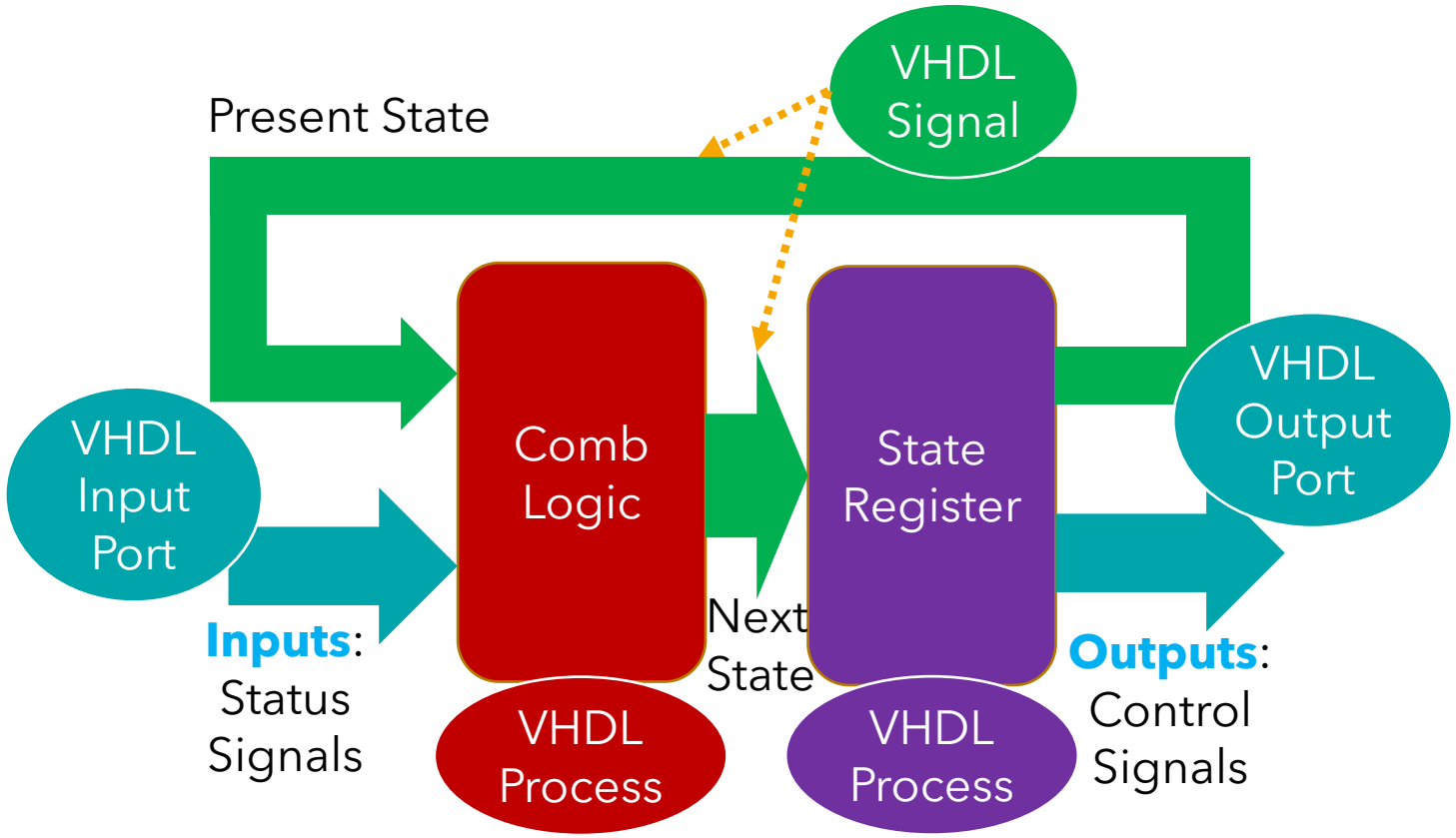
The 2 styles are equivalent

Organising complex designs

- Controller + Datapath
  - **Controller**: **FSM** (sends control signals)
    - What activity to enable in each clock cycle?
    - MUX select inputs
    - Load enable signals to flip-flops/registers
  - **Datapath**: **Computation**
    - Adders, Multipliers, MUXes
    - May send status signals back to FSM
      - results of comparisons
      - overflow/exceptions



Specifying FSMs in HDL



VHDL Model

```
entity...
  port...
end entity
architecture...
  signal...
  process...
  -- state register
  end process
  process...
  -- comb. logic
  end process
end architecture
```

State Register in VHDL

State Register

```
type state_type is (ADD, SUB);
signal cur_state : state_type := ADD;
signal next_state : state_type := ADD;

process (clk)
begin
if (clk'EVENT AND clk = '1') then
cur_state <= next_state;
end if;
end process;
```

states

initial state

rising edge

Q <= D

State Register with Reset

```
process (clk, reset)
begin
if (reset = '1') then
cur_state <= ADD;
elsif (clk'EVENT AND clk = '1')
then
cur_state <= next_state;
end if;
end process;
```

initial/reset state

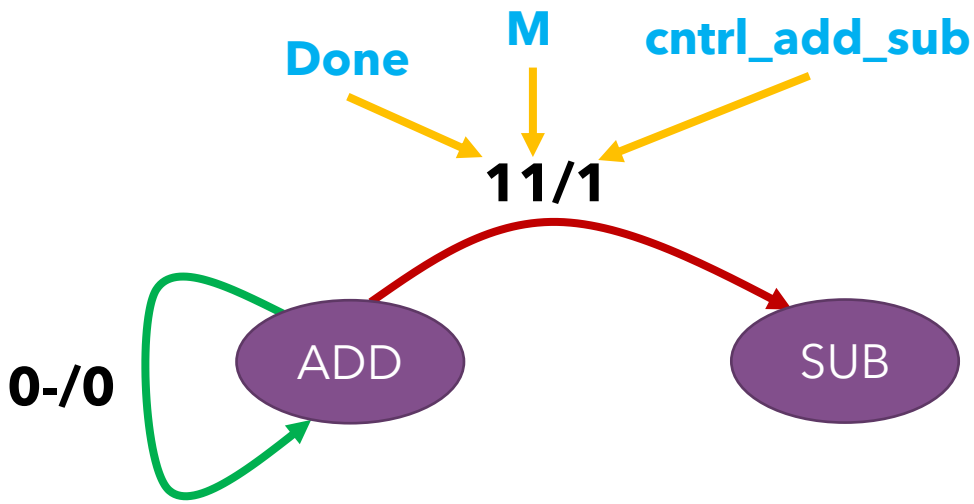
normal function

- Synthesised into set of D flip-flops
- #flip-flops (#bits) not specified here (automatically inferred)

Source: COL215 Hardware Assignment
Author: Naman Jain

Next State and Output Logic in VHDL

```
process (cur_state, M, Done)
begin
next_state <= cur_state;

case cur_state is
when ADD =>
  if Done = '1' and M = '1' then
     next_state <= SUB;
     cntrl_add_sub <= '1';
  elsif Done = '0' then
     next_state <= ADD;
     cntrl_add_sub <= '0';
  end if;
when SUB => ...
end case;
end process;
```

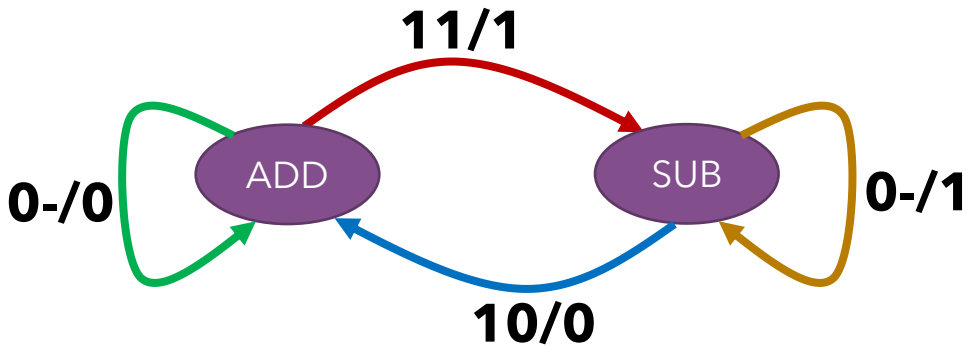


Source: COL215 Hardware Assignment
Author: Naman Jain

Next State and Output Logic in VHDL

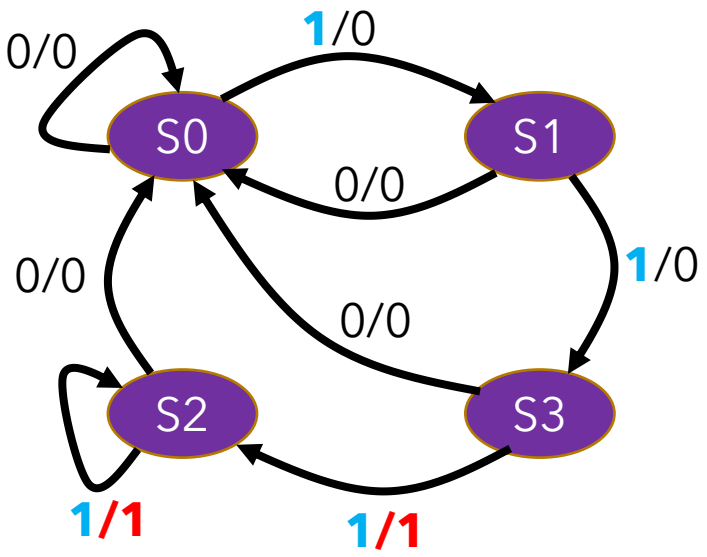
```
process (cur_state, M, Done)
begin
next_state <= cur_state;

case cur_state is
when ADD =>
  if Done = '1' and M = '1' then
     next_state <= SUB;
     cntrl_add_sub <= '1';
  elsif Done = '0' then
     next_state <= ADD;
     cntrl_add_sub <= '0';
  end if;
...
```

```
...
when SUB =>
  if Done = '1' and M = '0' then
     next_state <= ADD;
     cntrl_add_sub <= '0';
  elsif Done = '0' then
     next_state <= SUB;
     cntrl_add_sub <= '1';
  end if;
end case;
end process;
```



Source: COL215 Hardware Assignment
Author: Naman Jain

State Reduction: Equivalent States

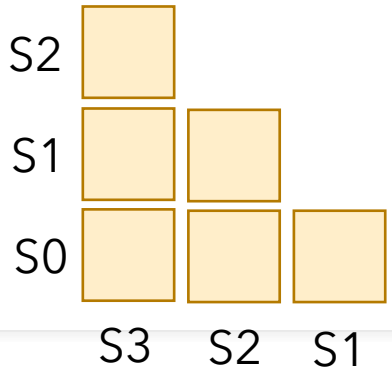


2 States are equivalent if:

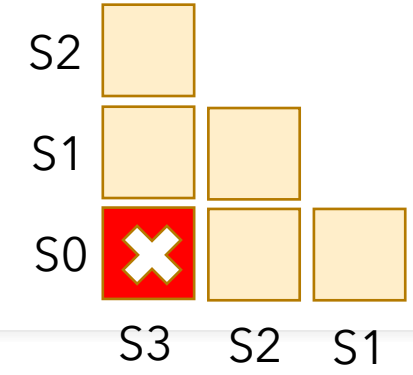
For every input:

- Same **output** values
- Same/Equivalent **next states**

Step 0



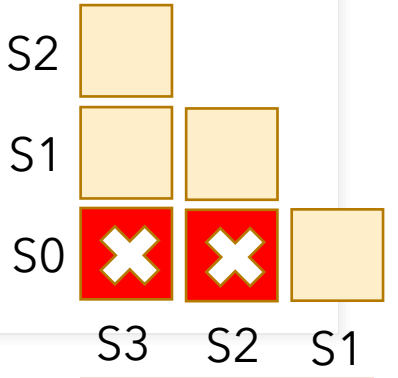
Step 1



S0: 0/0 1/0
S3: 0/0 1/1

O/P Conflict

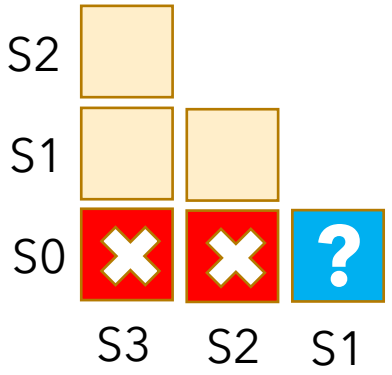
Step 2



S0: 0/0 1/0
S2: 0/0 1/1

O/P Conflict

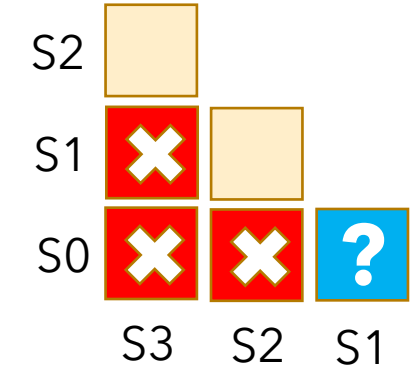
Step 3



S0: 0/0 1/0
S1: 0/0 1/0

O/P Compatible

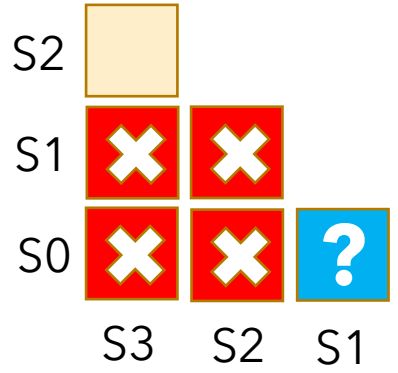
Step 4



S3: 0/0 1/1
S1: 0/0 1/0

O/P Conflict

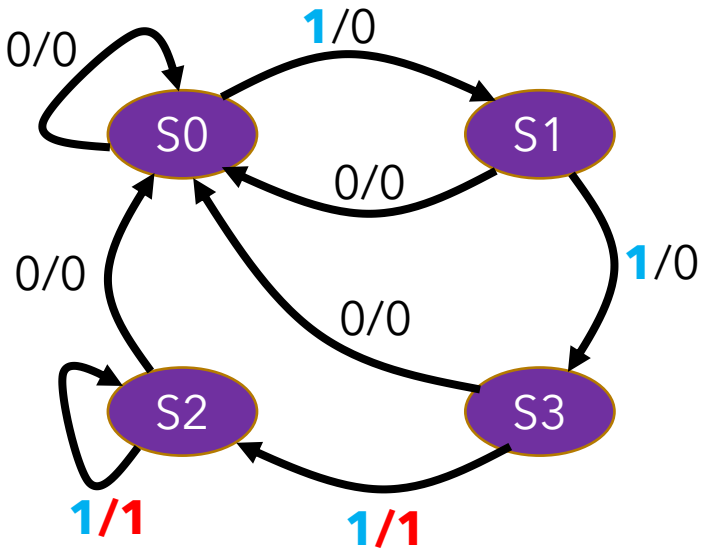
Step 5



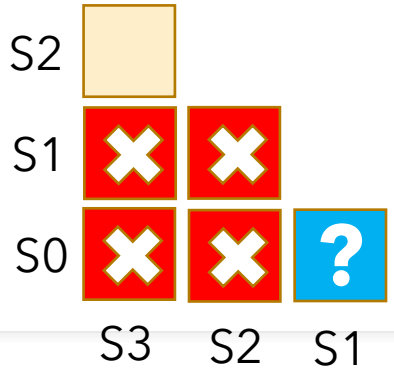
S2: 0/0 1/1
S1: 0/0 1/0

O/P Conflict

State Reduction : Equivalent States



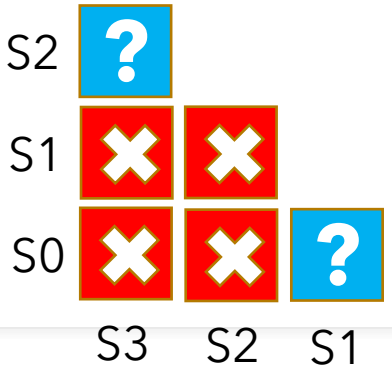
Step 5



S2: 0/0 1/1
S1: 0/0 1/0

O/P Conflict

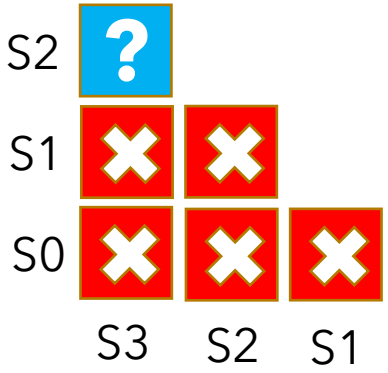
Step 6



S2: 0/0 1/1
S3: 0/0 1/1

O/P Compatible

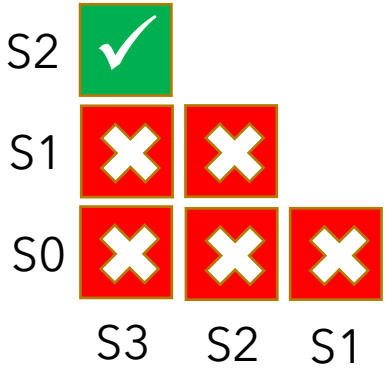
Step 7



S0: 0/[0,S0] 1/[0,**S1**]
S1: 0/[0,S0] 1/[0,**S3**]

Next State Conflict

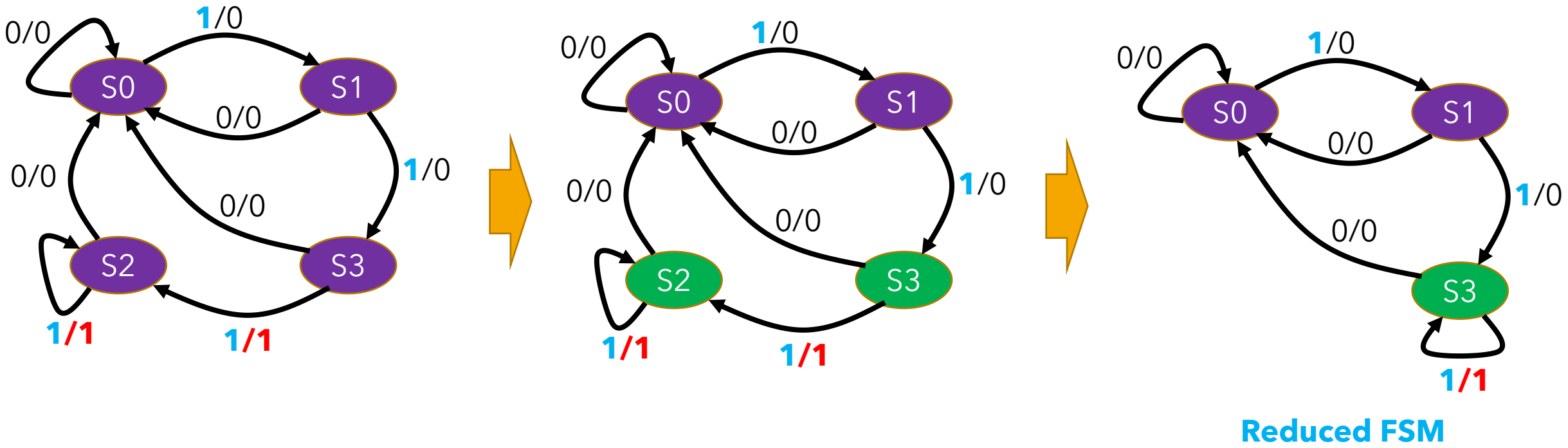
Step 8



S2: 0/[0,S0] 1/[1,S2]
S3: 0/[0,S0] 1/[1/S2]

Next State Compatible

Reducing the FSM



Reduced FSM