



SMJE 3173

Digital System Design

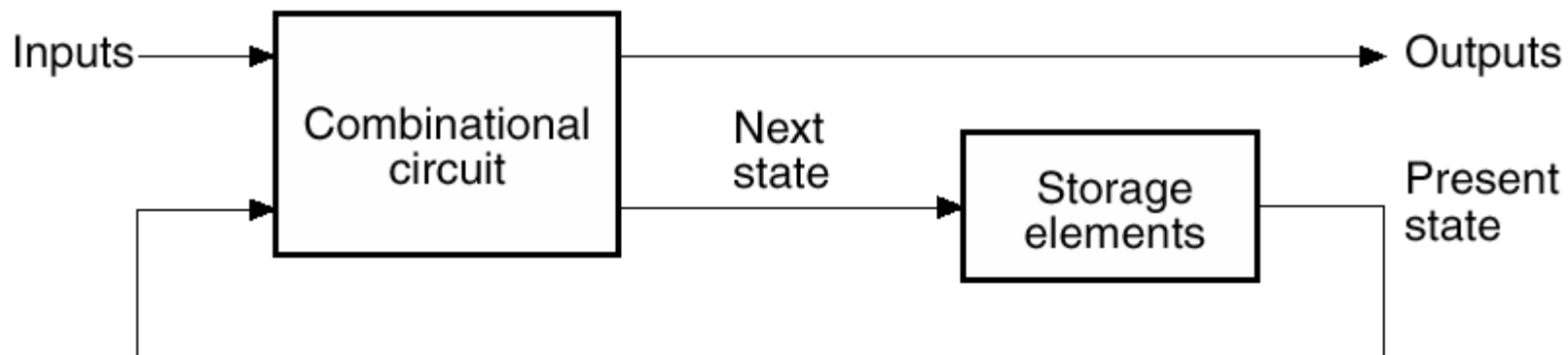
Sequential Circuits — Part 1

- ☐ ***Basic Latch***
- ☐ ***Gated SR & D Latches***
- ☐ ***D, T & JK Flip-Flops***
- ☐ ***Metastability***



Introduction

- Digital circuits are comprised of combinational and sequential circuits.
- 2 types of sequential circuits asynchronous and **synchronous**.
- This course only focus only synchronous sequential logic.
- The operation of synchronous circuit rely on basic circuits referred to as storage elements e.g. latches and flip-flops

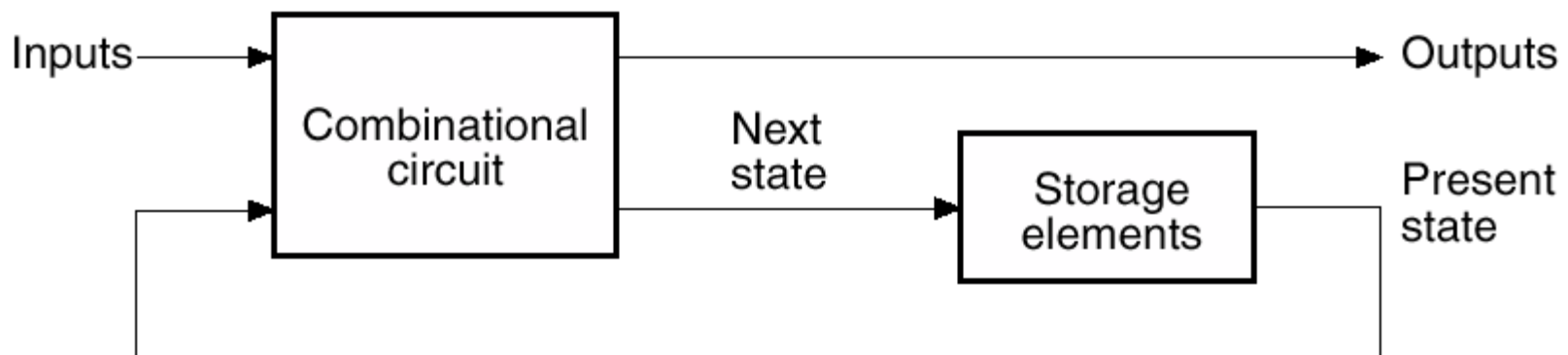


Block diagram of a digital circuit



Sequential Circuits

- **Combinational** – output depends only on the input.
 - Do not have memory
 - Cannot store state
- **Sequential** – output depends on input and past behavior.
 - Require use of storage elements.
 - Contents of storage elements is called state.
 - Circuit goes through sequence of states as a result of changes in inputs.





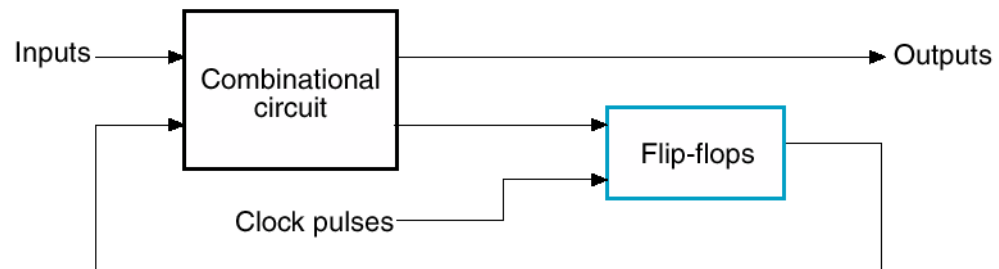
Sequential Circuits Types

■ Synchronous

- State changes synchronized by one or more clocks
- Easier to analyze because can factor out gate delays
- Set clock and data so changes allowed to occur before next clock pulse

■ Asynchronous

- Changes occur independently
- Potentially faster
- Harder to analyze



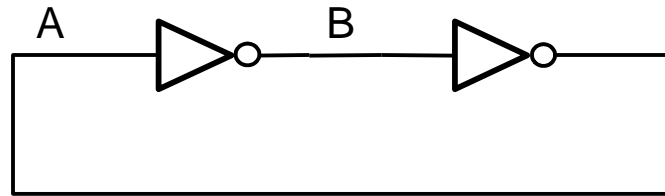
(a) Block diagram



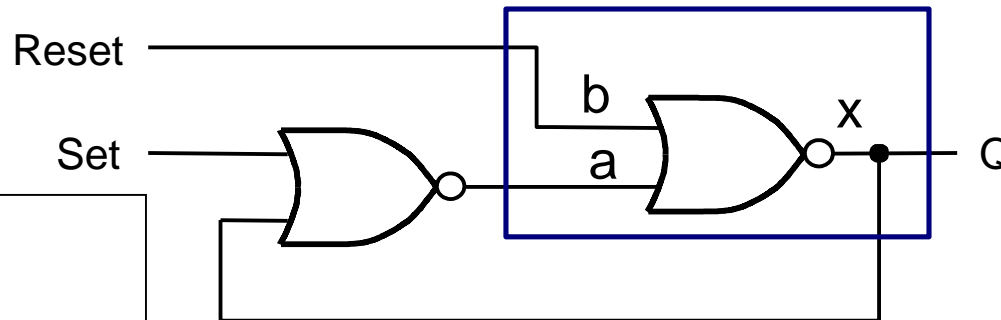
(b) Timing diagram of clock pulses



Simple Memory Elements (Review)



A simple memory element:
feedback will hold value



Either one of the inputs is
LOW , the output is a
complement to the input

Reset \neq Set

At $t=n$

if Reset = 0 \gg $Q = Q_{n-1}$

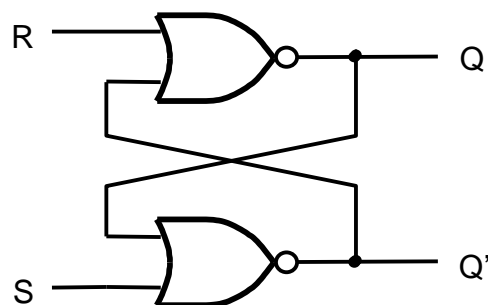
If Set = 0 \gg $Q = Q_{n-1}$

A memory element with NOR gates:
Use Set/Reset to change stored value

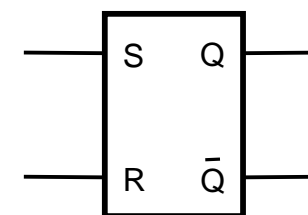


SR Latch (Review)

- Basic storage made from gates
- Rearrangement of memory element from previous slide!



Function Table			
S	R	Q(t+1)	Function
0	0	Q(t)	Hold
0	1	0	Reset
1	0	1	Set
1	1	?	Not allowed



Graphical symbol

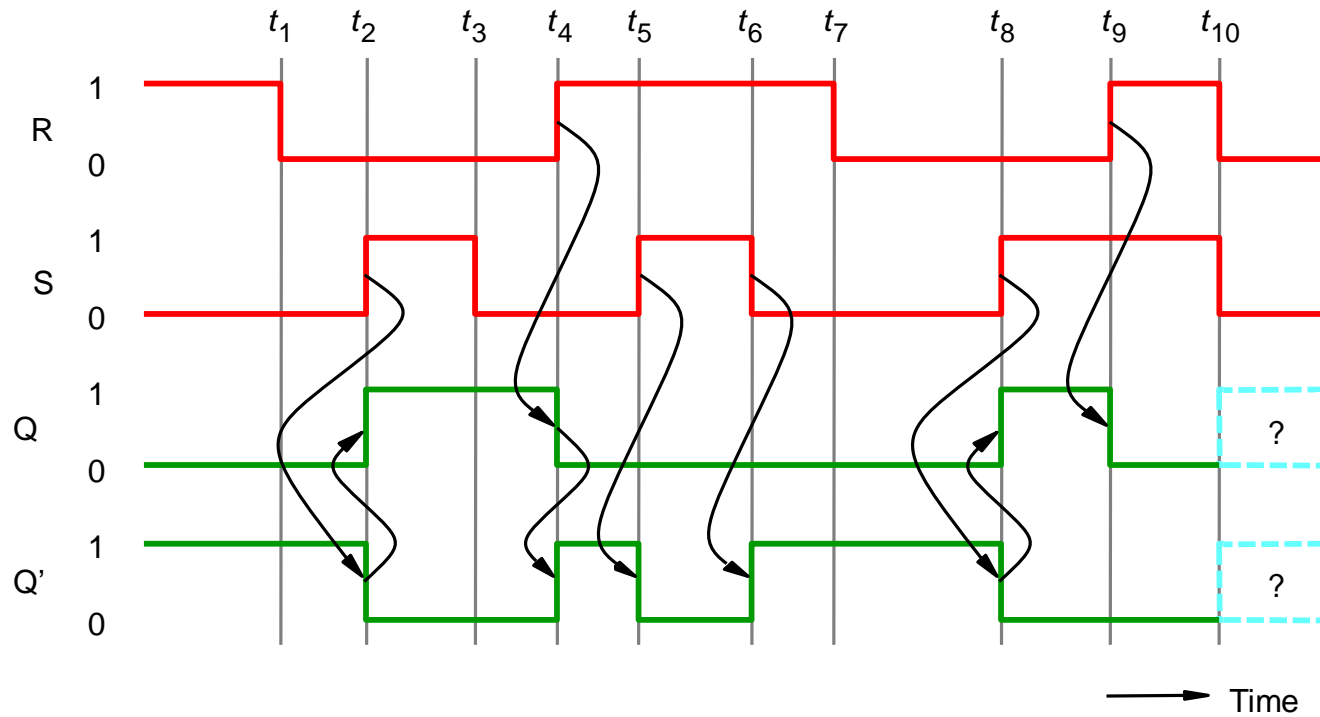
Note:

S = **S**et → Q = 1R = **R**eset → Q = 0This latch is an **active HIGH** S-R latch, i.e.when $S \neq R$ || S is active ($S=1$) → $Q' = 0 \gg Q = 1$ when $S \neq R$ || R is active ($R=1$) → $Q' = 1 \gg Q = 0$

- when $S=R$
 - If S & R both 0 at same time,
 $Q(t+1) = [Q'(t)]' \quad || \quad Q'(t+1) = [Q'(t)]'$
 - If S & R both 1 (active) at same time, $Q = Q' = 0$
 $Q(t+1) = 0 \quad || \quad Q'(t+1) = 0 \rightarrow$ **not allowed**



SR Latch (Review)





SR Latch (Review)

Detailed Function Table			
S	R	Q	Q+
0	0	0	0
0	0	1	1
0	1	0	0
0	1	1	0
1	0	0	1
1	0	1	1
1	1	0	X
1	1	1	X

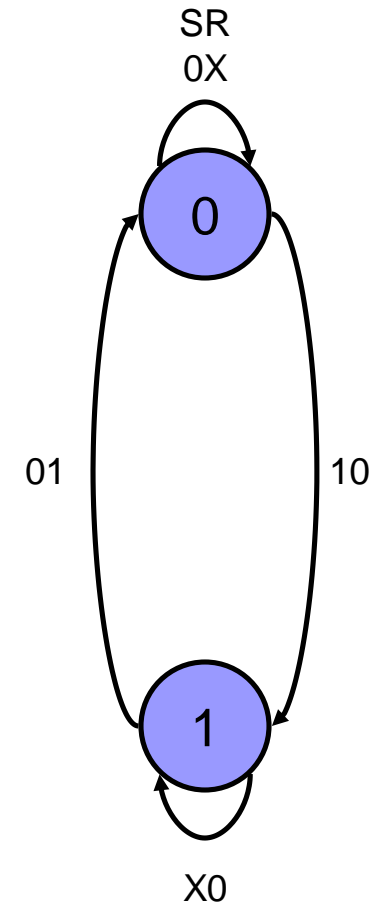
SR					
Q	SR	00	01	11	10
		0	1	X	1
0	0	0	0	X	1
1	1	1	0	X	1

Characteristic Equation:

$$Q+ = S + R'Q$$

Excitation Table			
Q	Q+	S	R
0	0	0	X
0	1	1	0
1	0	0	1
1	1	X	0

Excitation Table: What are the necessary inputs to cause a particular kind of change in state?



State Transition Diagram:

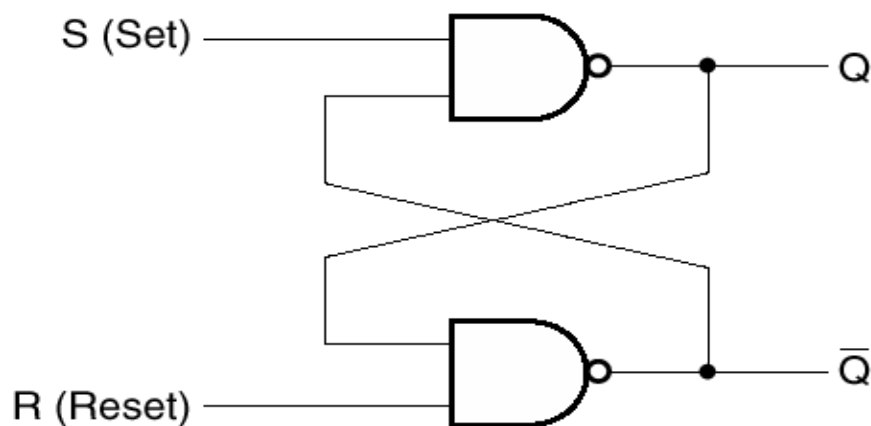
The excitation table in graphical form



$\overline{S} \overline{R}$ Latch (Review)

- Similar – made from NANDs

NAND gate:
Either one of the inputs is LOW
, the output will be HIGH



(a) Logic diagram

S	R	Q	\overline{Q}	
0	1	1	0	Set state
1	1	1	0	
1	0	0	1	Reset state
1	1	0	1	
0	0	1	1	Undefined

(b) Function table

Note:

S = **S**et → Q = 1

R = **R**eset → Q = 0

This latch is an **active LOW** S-R latch, i.e.

when $S \neq R$ || S is active ($S=0$) → **Q = 1** >> $Q' = 0$

when $S \neq R$ || R is active ($R=0$) → $Q' = 1$ >> **Q = 0**

- when $S=R$

➤ If S & R both 1 at same time,

$$Q(t+1) = [Q'(t)]' \quad || \quad Q'(t+1) = [Q'(t)]'$$

➤ If S & R both 0 (active) at same time, $Q = Q' = 0$

$$Q(t+1) = 1 \quad || \quad Q'(t+1) = 1 \rightarrow \text{not allowed}$$

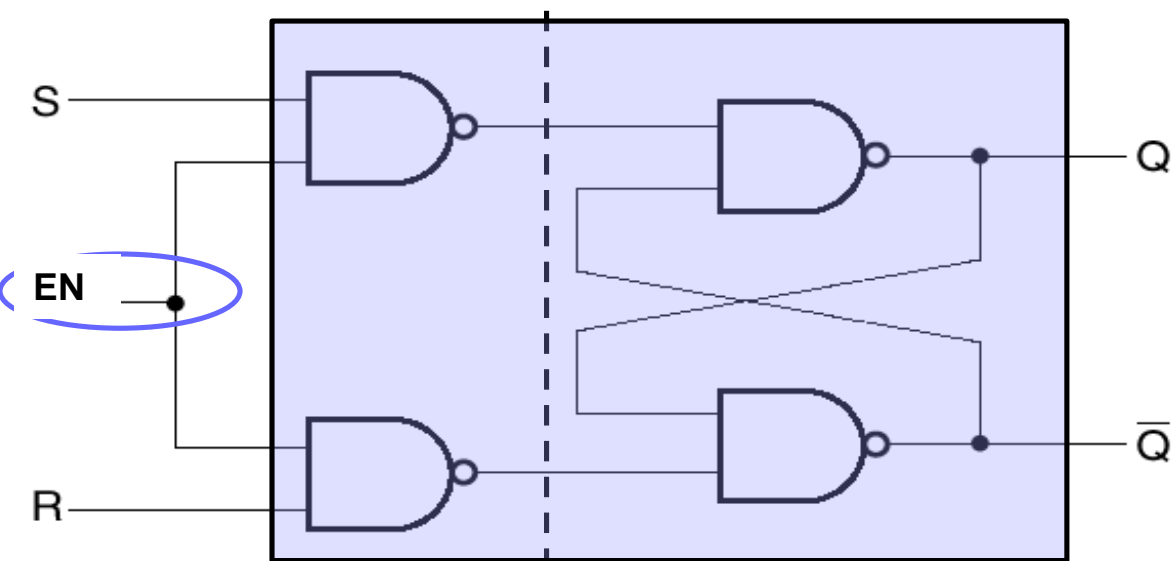


Gated SR Latch (Review)

■ Add Control Input - EN

□ Typically, control signal is referred to as a clock

■ The EN (enable) input controls when state can change



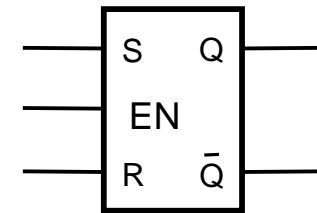
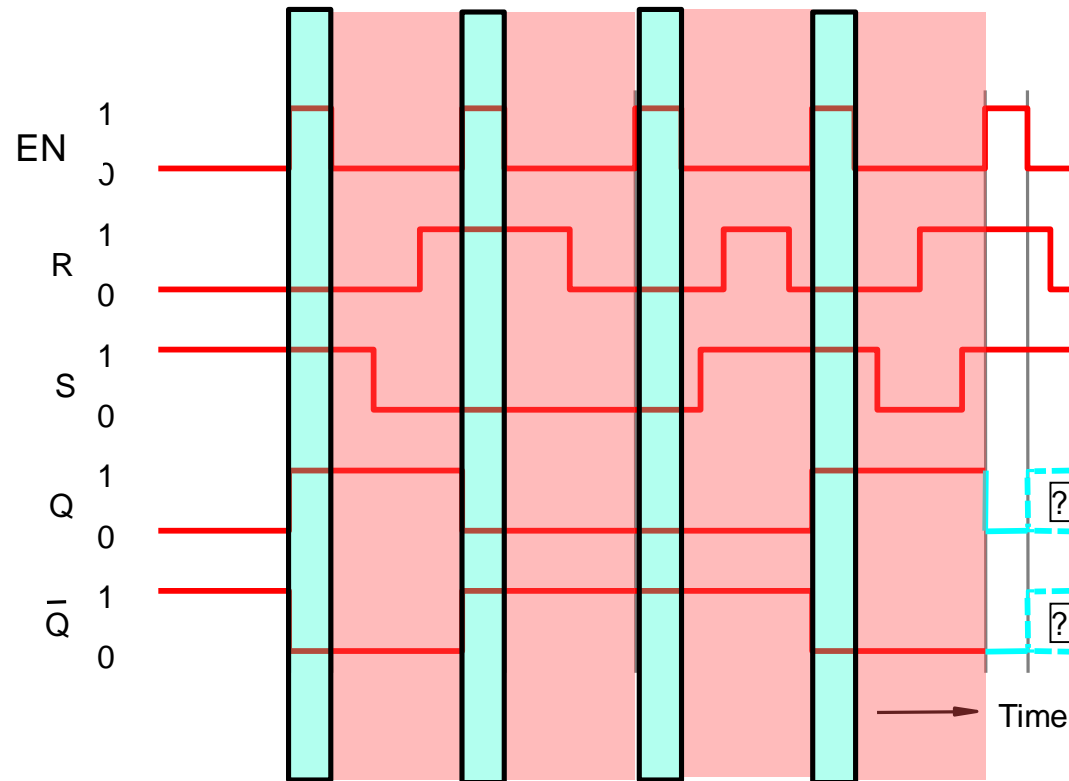
(a) Logic diagram

EN	S	R	Next state of Q
0	X	X	No change
1	0	0	No change
1	0	1	Q = 0; Reset state
1	1	0	Q = 1; Set state
1	1	1	Undefined

(b) Function table



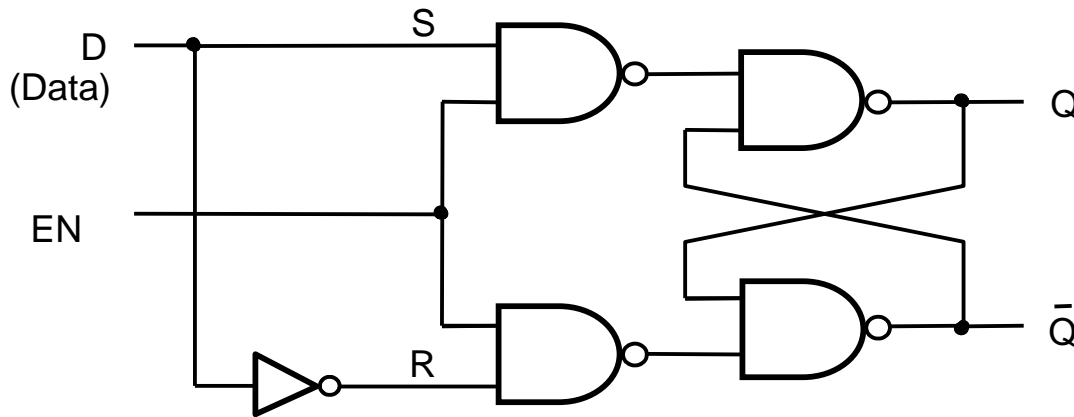
Gated SR Latch



Graphical symbol



Gated D Latch (Review)



(a) Circuit

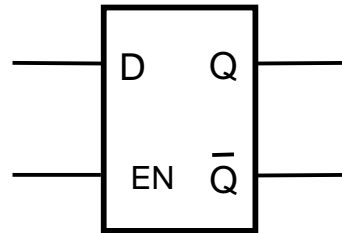
EN	D	Next state of Q
0	X	No change
1	0	Q = 0; Reset state
1	1	Q = 1; Set state

(b) Function table

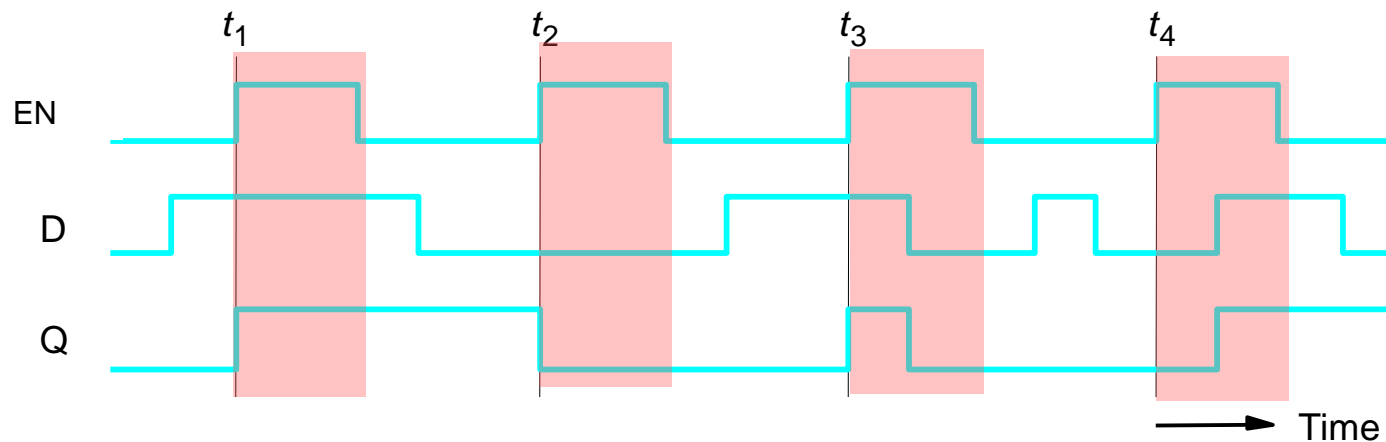
- Input **D** assures input $S \neq R$. Why?
- No illegal output state (S can never be the same R at any instant)
- The output Q will follow the same condition as input D. **Thus, the name Gated D Latch**



Gated D Latch



(c) Graphical symbol



(d) Timing diagram

Take note that Latch is a level-sensitive device



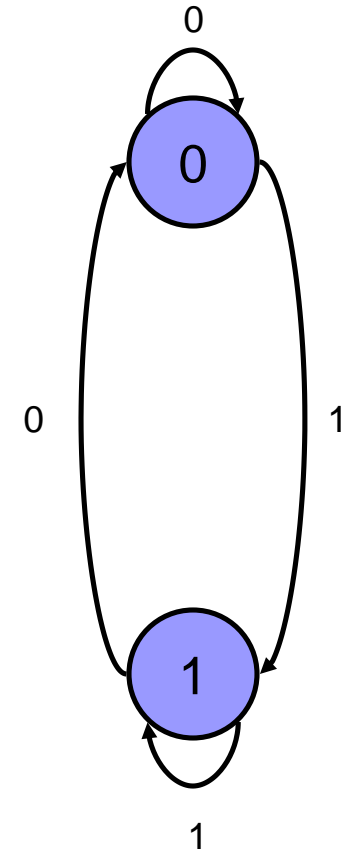
D Latch

Detailed Function Table		
D	Q	Q+
0	0	0
0	1	0
1	0	1
1	1	1

D \ Q	0	1
0	0	1
1	0	1

Characteristic Equation
 $Q+ = D$

Excitation Table		
Q	Q+	D
0	0	0
0	1	1
1	0	0
1	1	1



State Transition Diagram

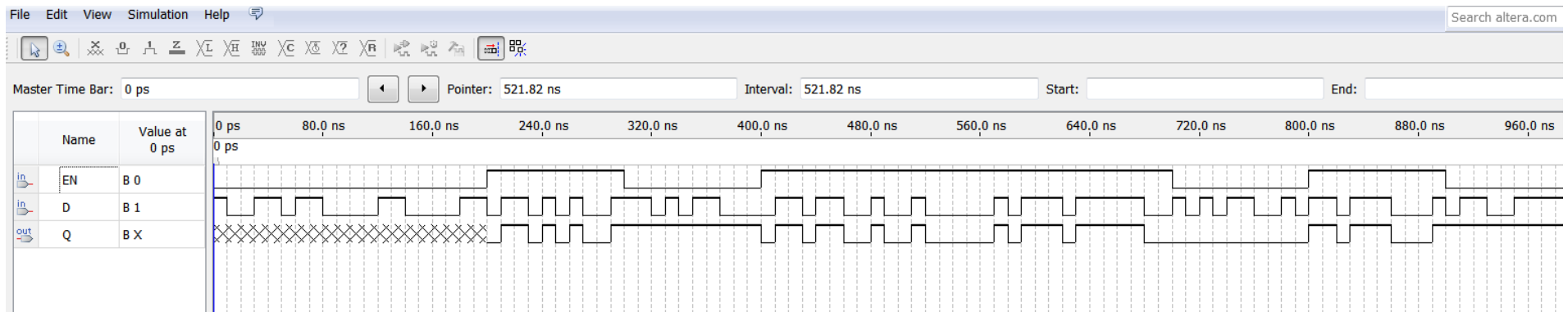


Example: D-latch (Verilog construct)

Dee_Latch.v

```
1 module Dee_Latch (D, EN, Q);
2
3 input D, EN;
4 output reg Q;
5
6 always @ (D or EN)
7     if (EN==1) Q = D;
8     //without ELSE statement. Verilog compiler assumes that when EN=0,
9     //previous value of Q is maintained (no change).
10 endmodule
```

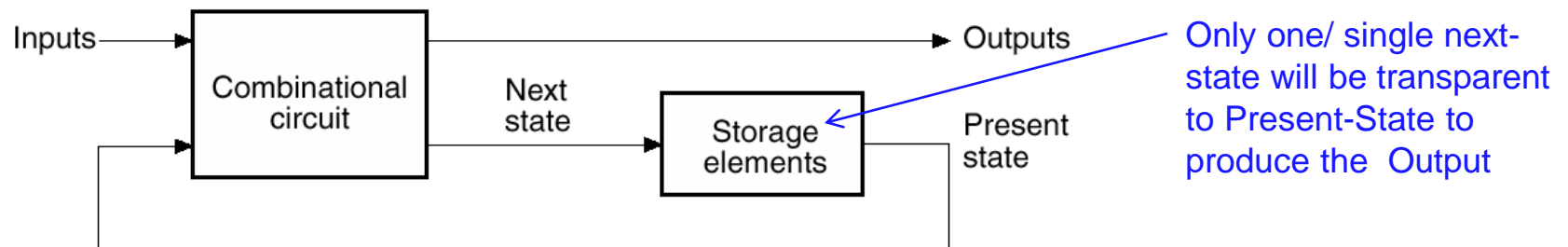
Compilation Report - Dee_Latch





Transparency

- As long as control input EN is HIGH, latch (output) state can change
- This is called *transparency*
- What's the problem with that? What if the input keeps changing state i.e. switching state while EN is HIGH



- How to make latch state change only **once**
- **What** is the factor that enables latch change state only **once**
 - Depending on inputs at time of clock



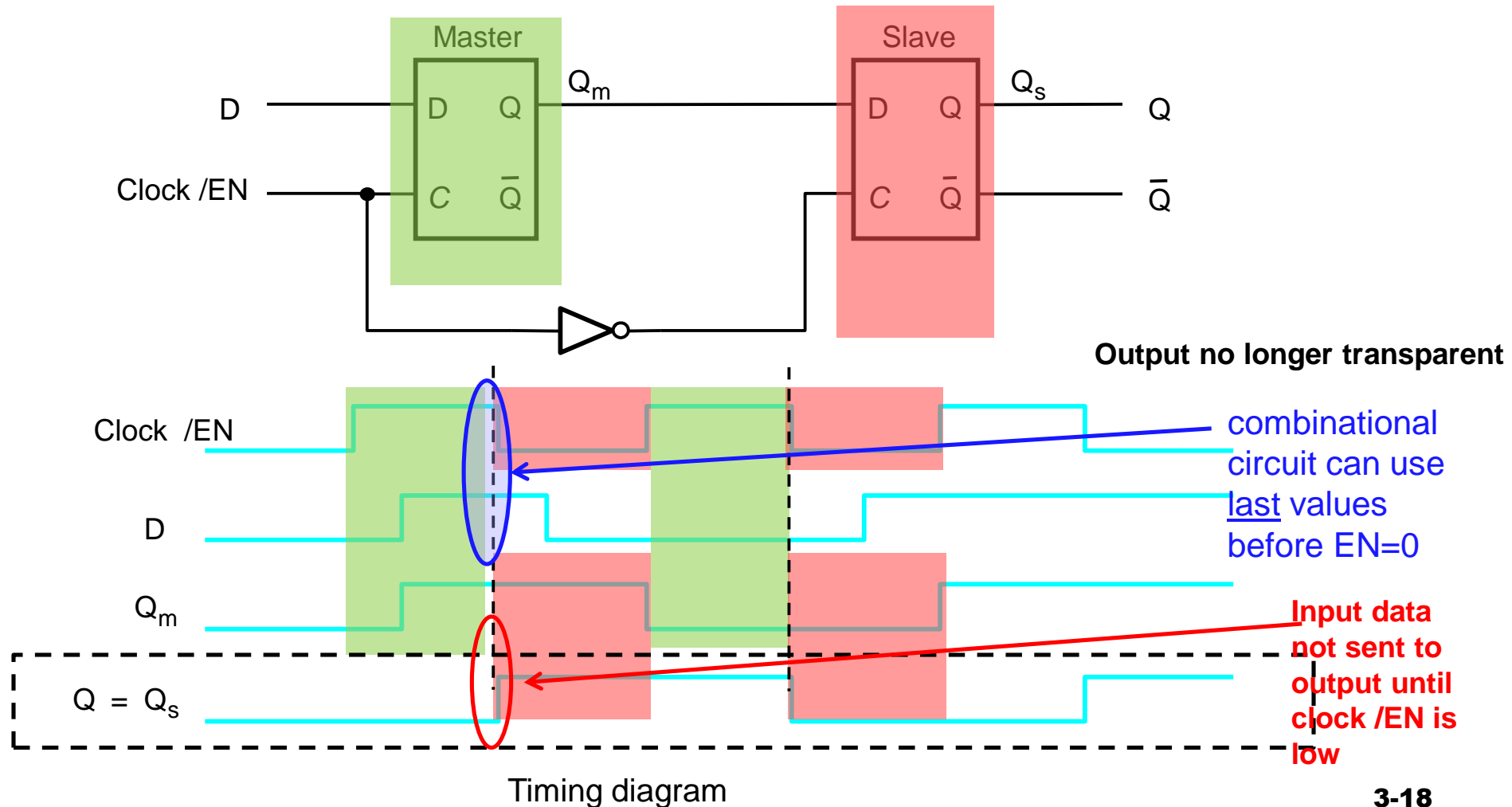
Flip-Flops

- Ensure only one transition
- Two major types:
 - (i) **Master-Slave**
 - Two stage
 - Output not changed until clock disabled
 - (ii) **Edge triggered**
 - Change happens when clock level changes



Master-Slave D Flip-Flop

- Either Master or Slave is enabled, not both





Summary of Master-Slave Flip flop

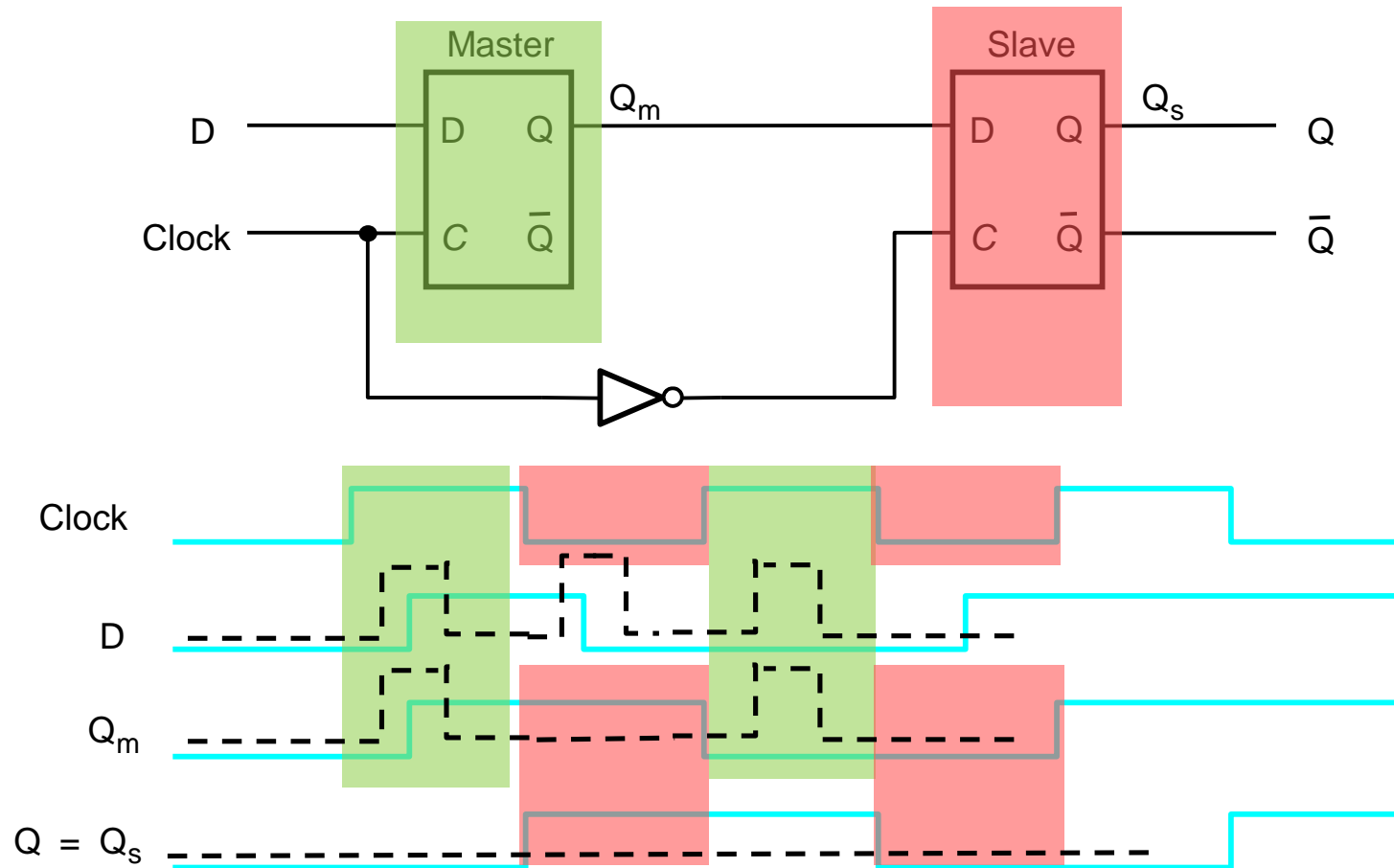
Have We Fixed the Transparency Problem?

- Output no longer transparent
 - Combinational circuit can use last values
 - New inputs appear at latches
 - Not sent to output until clock low
- But changes at input of FF when clock high can affect output
- Solution: **edge-triggered flip-flops**
- New state latched on **clock transition**
 - Low-to-high or high-to-low
- Changes when clock high are ignored
- Note: Master-Slave also called **pulse triggered**



Master-Slave D Flip-Flop

- What if data change states twice when EN is active (C=1) in Master latch?



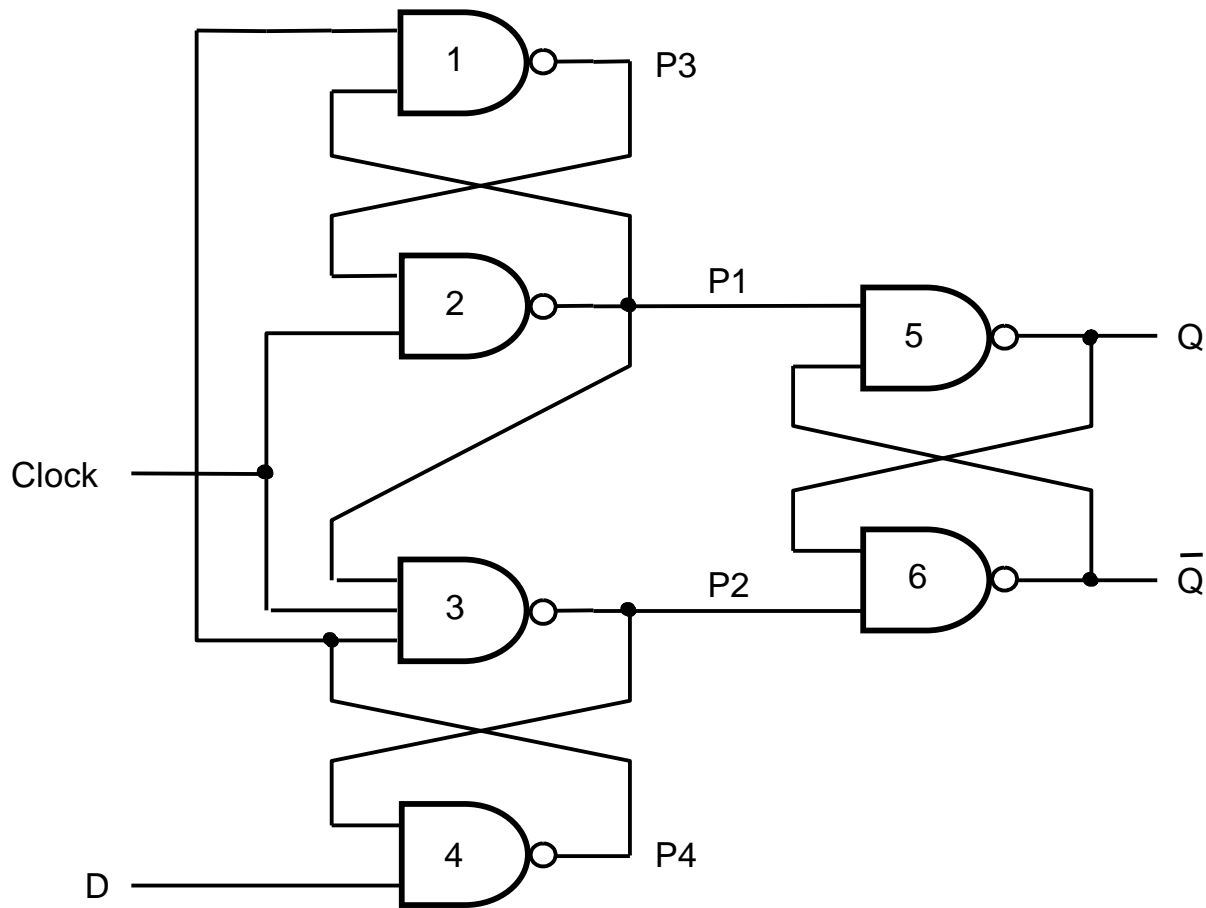


Have We Fixed the Transparency Problem?

- Output no longer transparent
 - Combinational circuit can use last values
 - New inputs appear at latches
 - Not sent to output until clock low
- But changes at input of FF when clock high can affect output
- Solution: ***edge-triggered flip-flops***
- New state latched on ***clock transition***
 - Low-to-high or high-to-low
- Changes when clock high are ignored
- Note: Master-Slave also called ***pulse triggered***

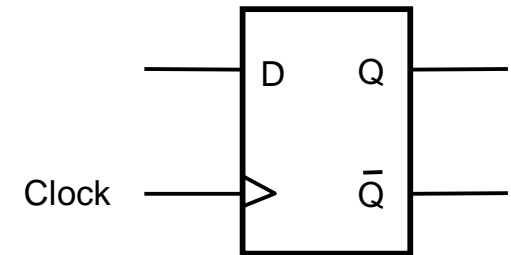


D Flip-Flop



(a) Circuit

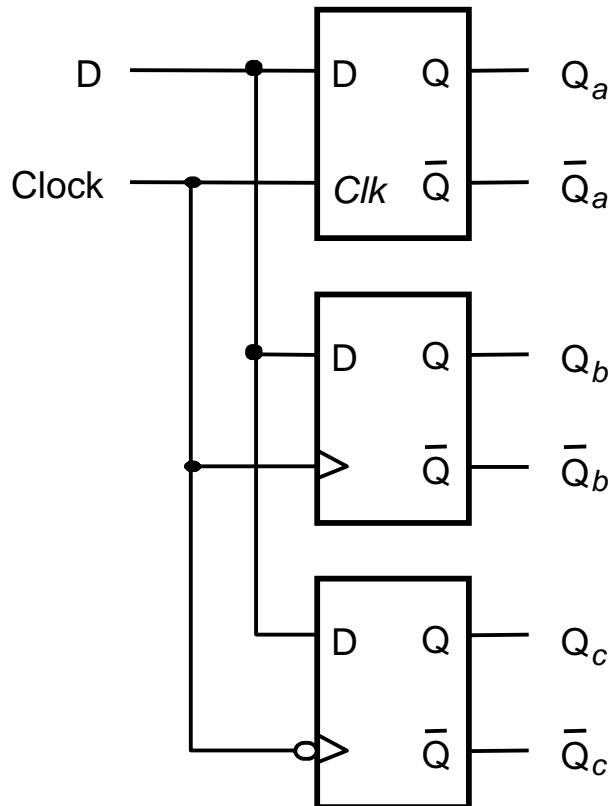
A positive-edge-triggered D flip-flop



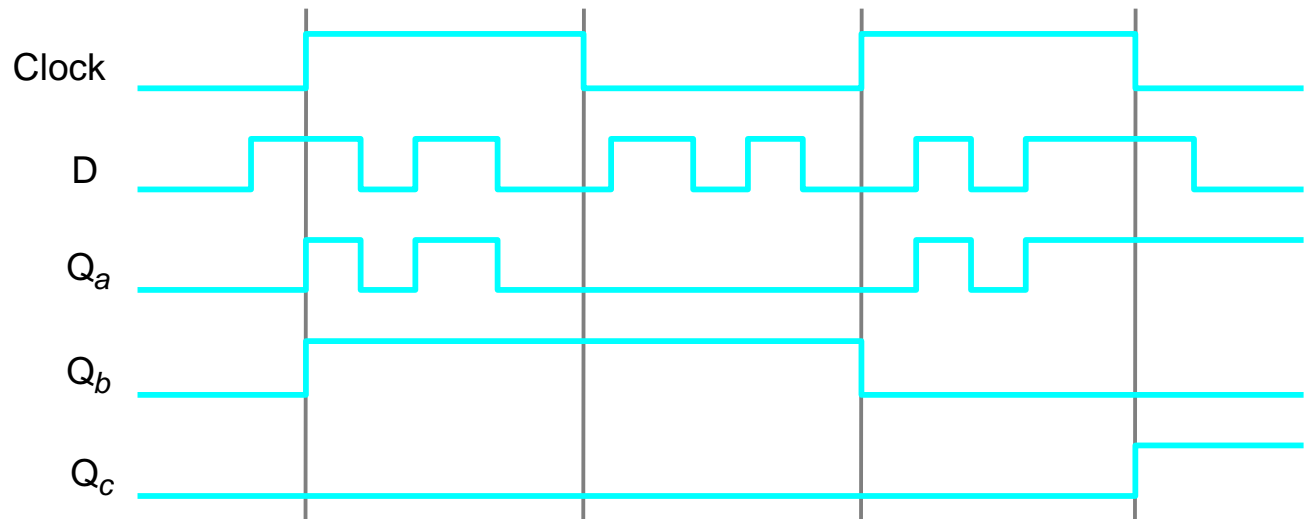
(b) Graphical symbol



D Latch versus D Flip-Flop



(a) Circuit

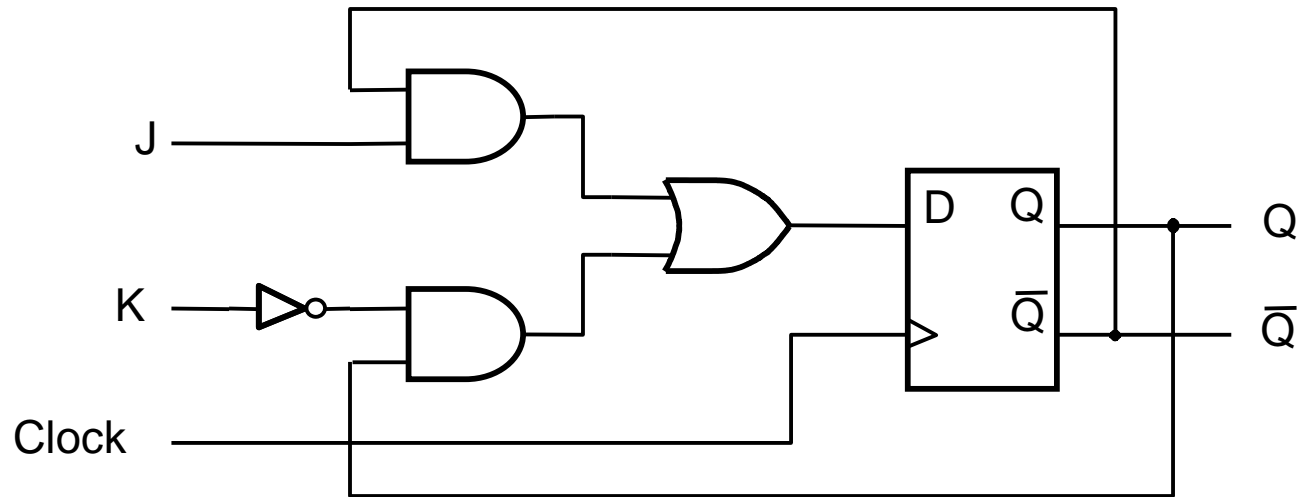


(b) Timing diagram

Comparison of level-sensitive and edge-triggered devices



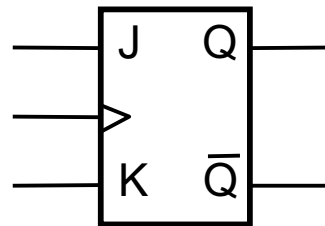
JK Flip-Flop



(a) Circuit

J	K	$Q(t+1)$
0	0	$Q(t)$
0	1	0
1	0	1
1	1	$\bar{Q}(t)$

(b) Truth table



(c) Graphical symbol

- Not used much anymore in VLSI
- Advantageous only if using FF chips



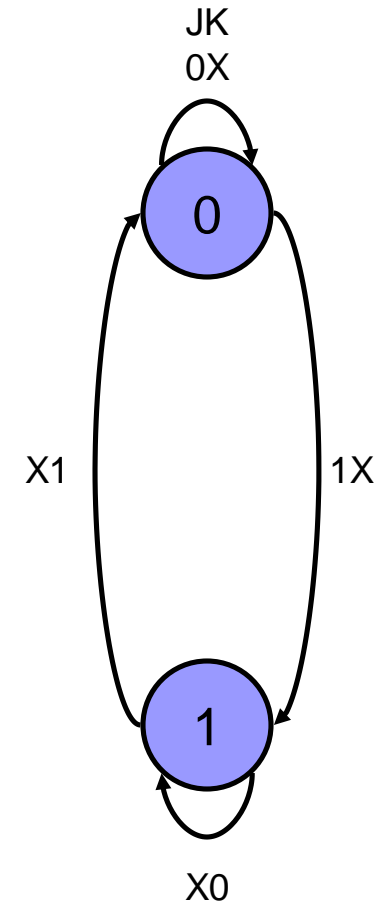
JK Flip-Flop

Detailed Function Table			
J	K	Q	Q+
0	0	0	0
0	0	1	1
0	1	0	0
0	1	1	0
1	0	0	1
1	0	1	1
1	1	0	1
1	1	1	0

JK Q	JK			
	00	01	11	10
0	0	0	1	1
1	1	0	0	1

Characteristic Equation
 $Q+ = JQ' + K'Q$

Excitation Table			
Q	Q+	J	K
0	0	0	X
0	1	1	X
1	0	X	1
1	1	X	0



State Transition Diagram

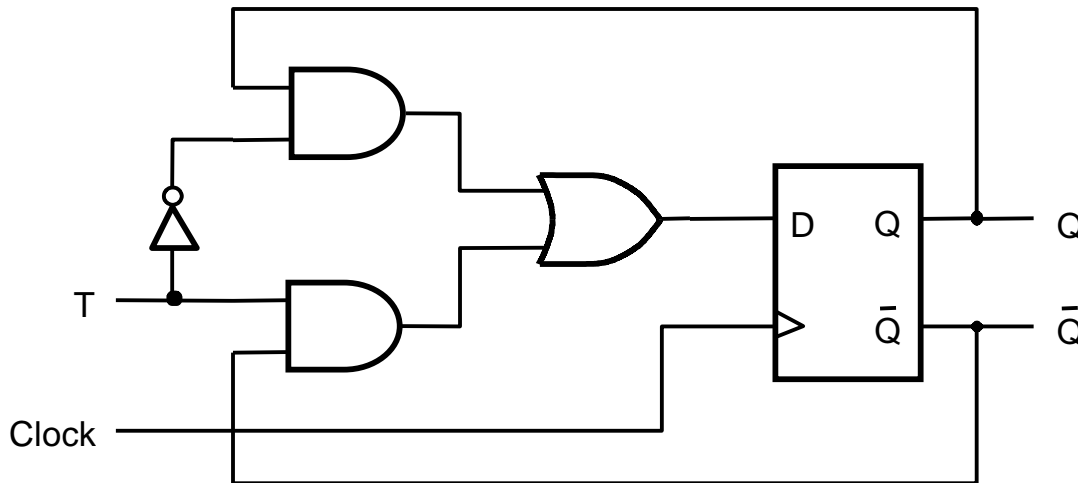


T Flip-Flop

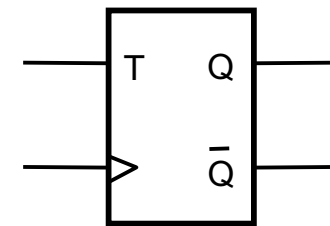
- Useful in counters
- Not available in IC form
- T Latches do not exist

T	$Q(t+1)$
0	$Q(t)$
1	$\bar{Q}(t)$

(b) Truth table



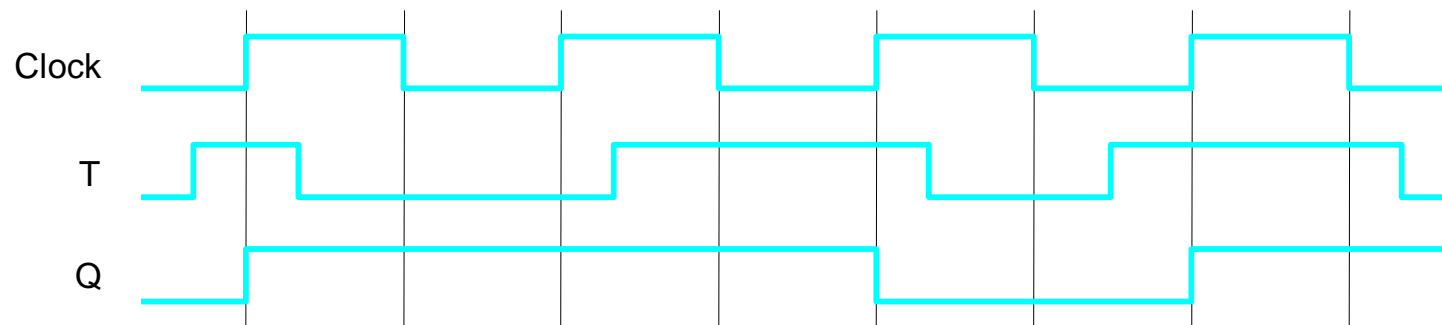
(a) Circuit



(c) Graphical symbol



T Flip-Flop



(d) Timing diagram



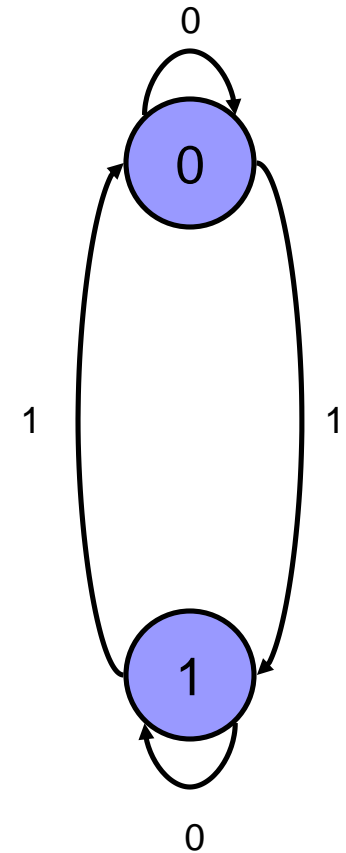
T Flip-Flop

Detailed Function Table		
T	Q	Q+
0	0	0
0	1	1
1	0	1
1	1	0

T \ Q	0	1
0	0	1
1	1	0

Characteristic Equation
 $Q+ = T'Q + TQ' = T \oplus Q$

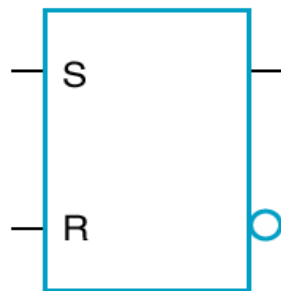
Excitation Table		
Q	Q+	T
0	0	0
0	1	1
1	0	1
1	1	0



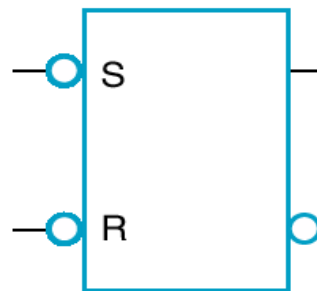
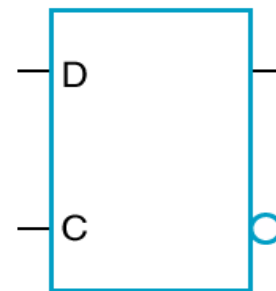
State Transition Diagram



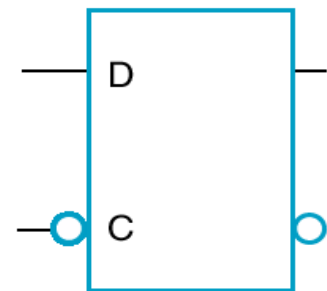
Standard Symbols – Latches



SR

 $\overline{S}\overline{R}$ 

D with 1 Control



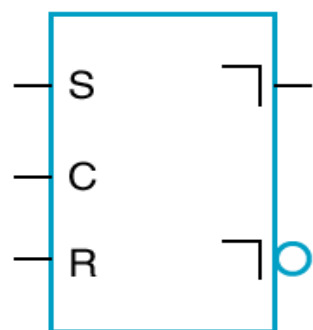
D with 1 Control

(a) Latches

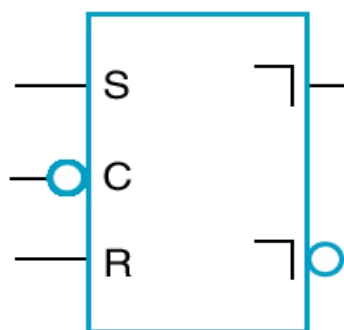
Circle at input indicates negation



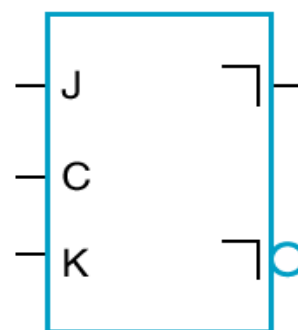
Symbols – Master-Slave



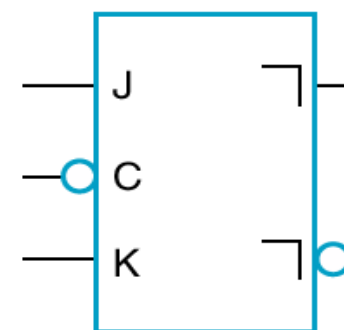
Triggered SR



Triggered SR



Triggered JK



Triggered JK

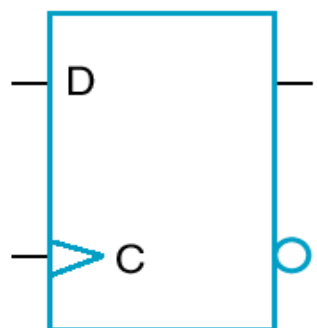
(b) Master-Slave Flip-Flops

Inverted L indicates postponed output

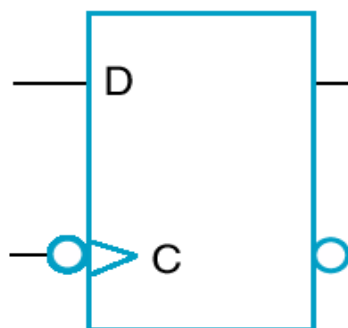
Circle indicates whether enable is positive or negative



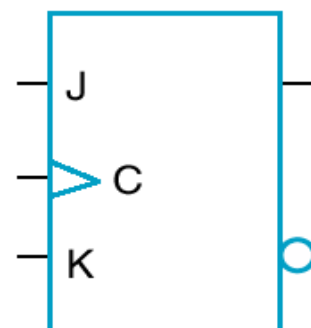
Symbols – Edge-Triggered



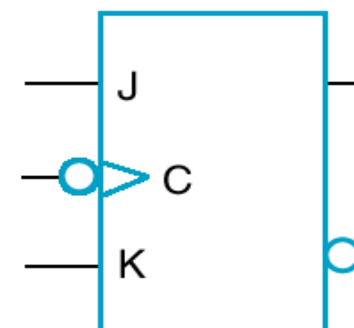
┐ Triggered D



┘ Triggered D



┐ Triggered JK



┘ Triggered JK

(c) Edge-Triggered Flip-Flops

Arrow indicates edge trigger



Flip-flop in Verilog

- In edge-triggered FF the o/p Q changes only at the transition of the clock signal and not due to the level of the signal.
- In Verilog the event qualifier **posedge** or **negedge** is used in the sensitivity list.

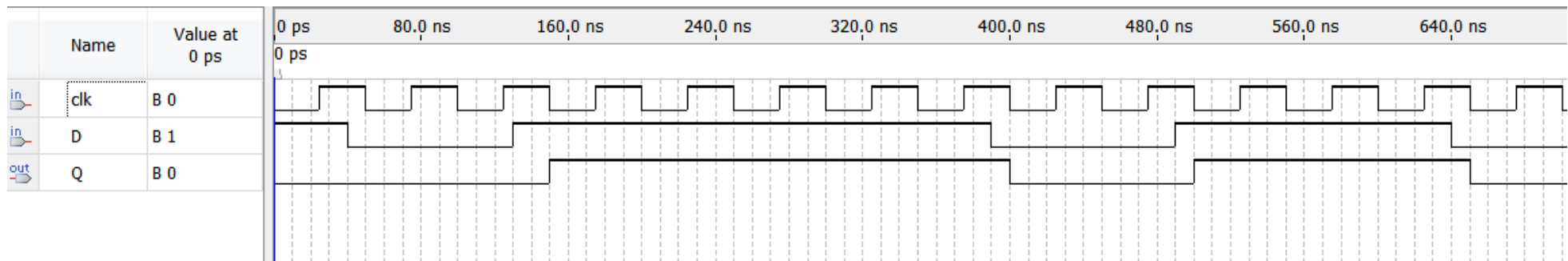
always @ (posedge clock) // positive-triggerred FF

always @ (negedge clock) // negative-triggerred FF



Flip flop in Verilog

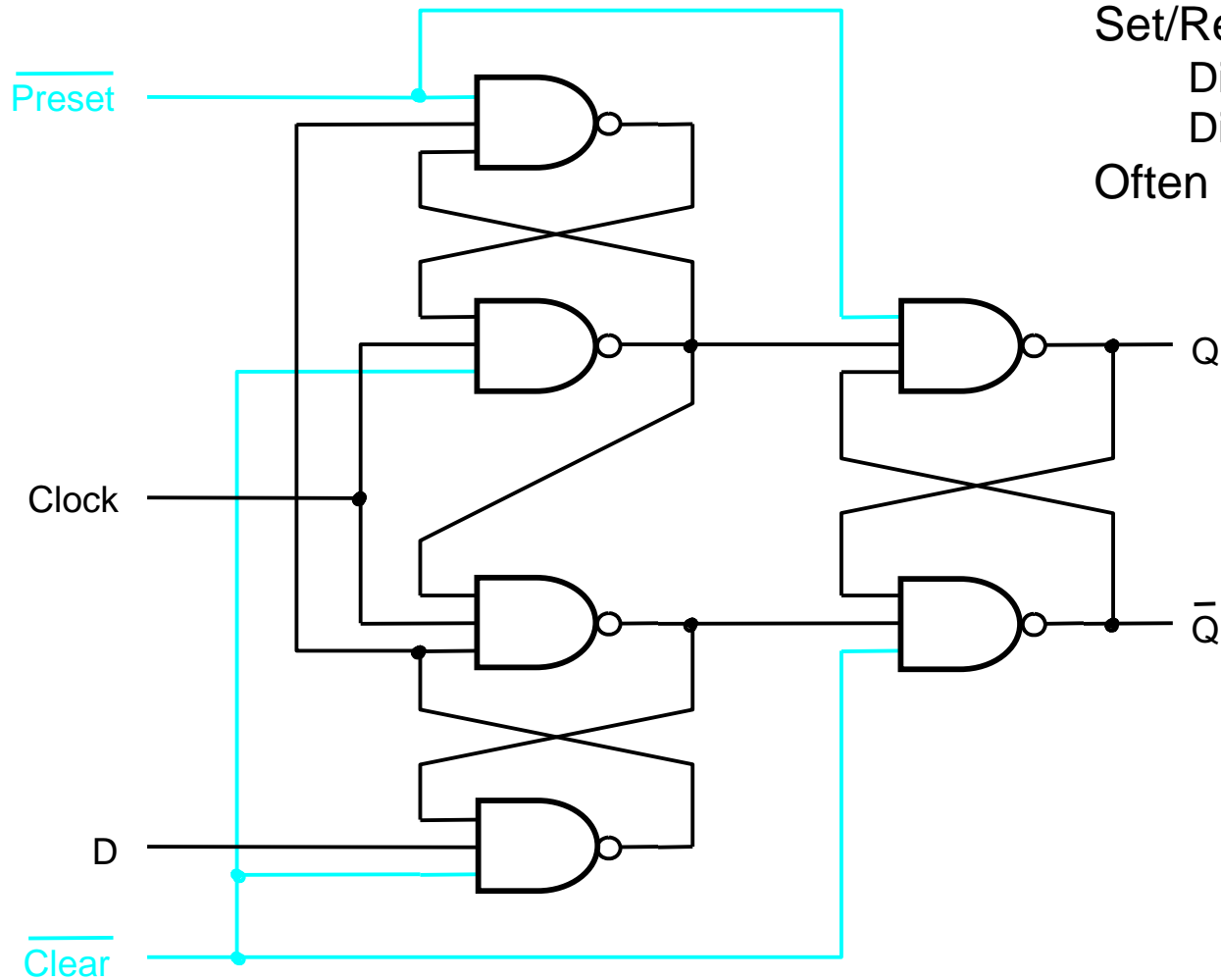
```
module negDff (clk, D, Q);  
  input D, clk;  
  output reg Q;  
  
  always @ (negedge clk)  
    Q = D;  
endmodule
```



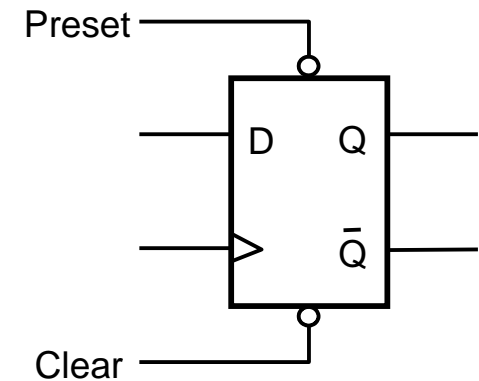


Clear and Preset Inputs

Set/Reset independent of clock
Direct set or *preset*
Direct reset or *clear*
Often used for power-up reset



(a) Circuit



(b) Graphical symbol



Choosing a Flip-Flop

- RS Clocked Latch:
 - ☐ used as storage element in narrow width clocked systems
 - ☐ its use is not recommended!
 - ☐ however, fundamental building block of other flipflop types
- D Flipflop:
 - ☐ minimizes wires, much preferred in VLSI technologies
 - ☐ simplest design technique
 - ☐ best choice for storage registers
- JK Flipflop:
 - ☐ versatile building block: can be used to implement D and T FFs
 - ☐ usually requires least amount of logic to control Q+
 - ☐ however, has two inputs with increased wiring complexity
- T Flipflop:
 - ☐ doesn't really exist, constructed from J-K FFs
 - ☐ usually best choice for implementing counters
- Preset and Clear inputs highly desirable!!



Characteristic Equation & Excitation Table Summary

<i>Device Type</i>	<i>Characteristic Equation</i>
--------------------	--------------------------------

D latch

$$Q^+ = D$$

T flip-flop

$$Q^+ = TQ' + T'Q$$

Q	Q+	D	T
0	0	0	0
0	1	1	1
1	0	0	1
1	1	1	0



Implementing One FF in Terms of Another (Method 1)

- Design Procedure: Implementing D FF with a J-K FF:
 1. Start with K-map of $Q^+ = f(D, Q)$
 2. Create K-maps for J and K with same inputs (D, Q)
 3. Referring to excitation table, fill in K-maps with appropriate values for J and K to cause the same state changes as in the original K-map

Q	Q ⁺	S	R	D	J	K	T
0	0	0	X	0	0	X	0
0	1	1	0	1	1	X	1
1	0	0	1	0	X	1	1
1	1	X	0	1	X	0	0

		D	
	Q	0	1
0		0	1
1		0	1

$Q^+ = D$

		D	
	Q	0	1
0		0	1
1		X	X

		D	
	Q	0	1
0		X	X
1		1	0

e.g., $D = Q = 0, Q^+ = 0$

1. From table: $J = 0, K = X$
2. Fill 0 to $QD = 00$ for J map
3. Fill X to $QD = 00$ for K map



Implementing One FF in Terms of Another (Method 2)

- Design Procedure: Implementing D FF with a J-K FF:

D	Q
0	0
1	1

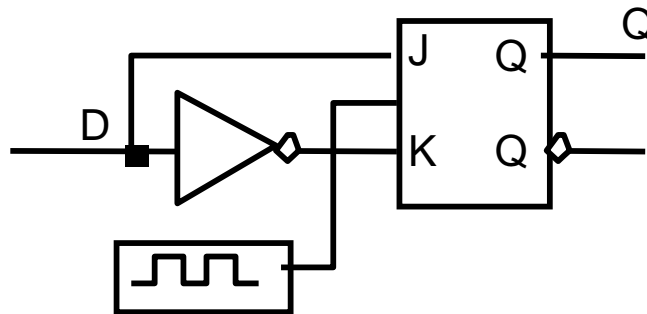
Where is the similar operations

J	K	Q
0	0	Hold
0	1	0 (reset)
1	0	1 (set)
1	1	Toggle



Implementing One FF in Terms of Another

- The circuit

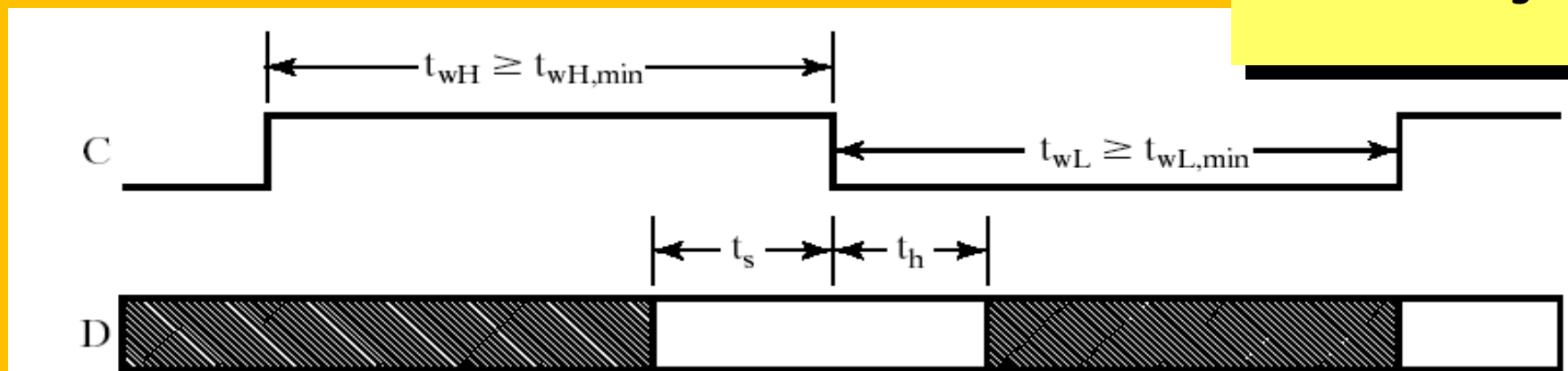


How to implment a T-flip flop using J-K f/f?



Flip-Flop Timing

There is a timing "window" around the clocking event during which the input must remain stable and unchanged in order to be recognized

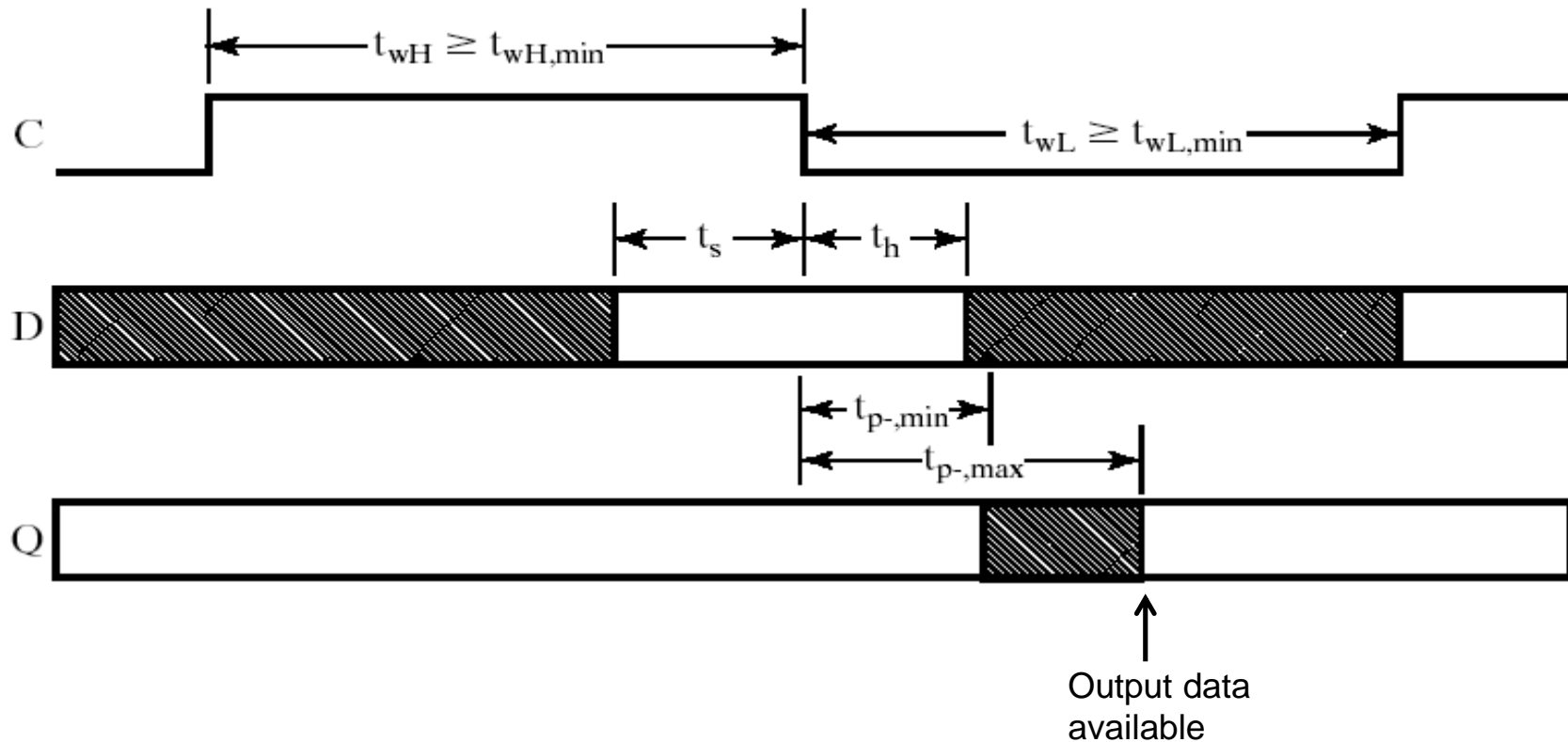


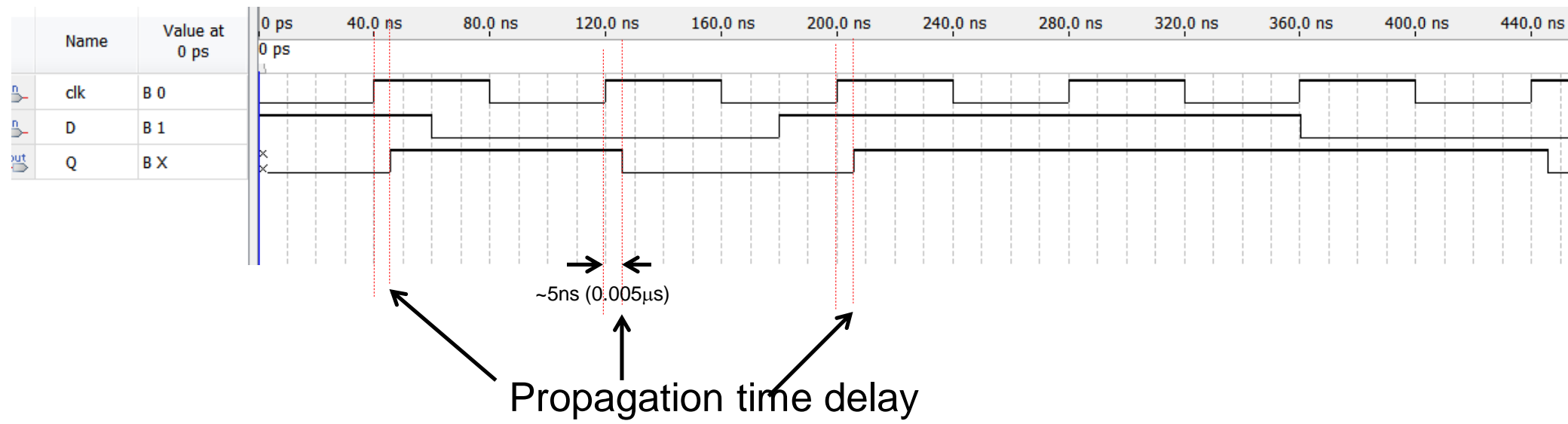
- Clock - Periodic event, causes state of memory element to change
- Setup time (t_s) – time that D must be available before clock edge
- Hold time (t_h) – time that D must be stable after clock edge



Propagation Delay

- Propagation delay – time after active triggering edge when output is available

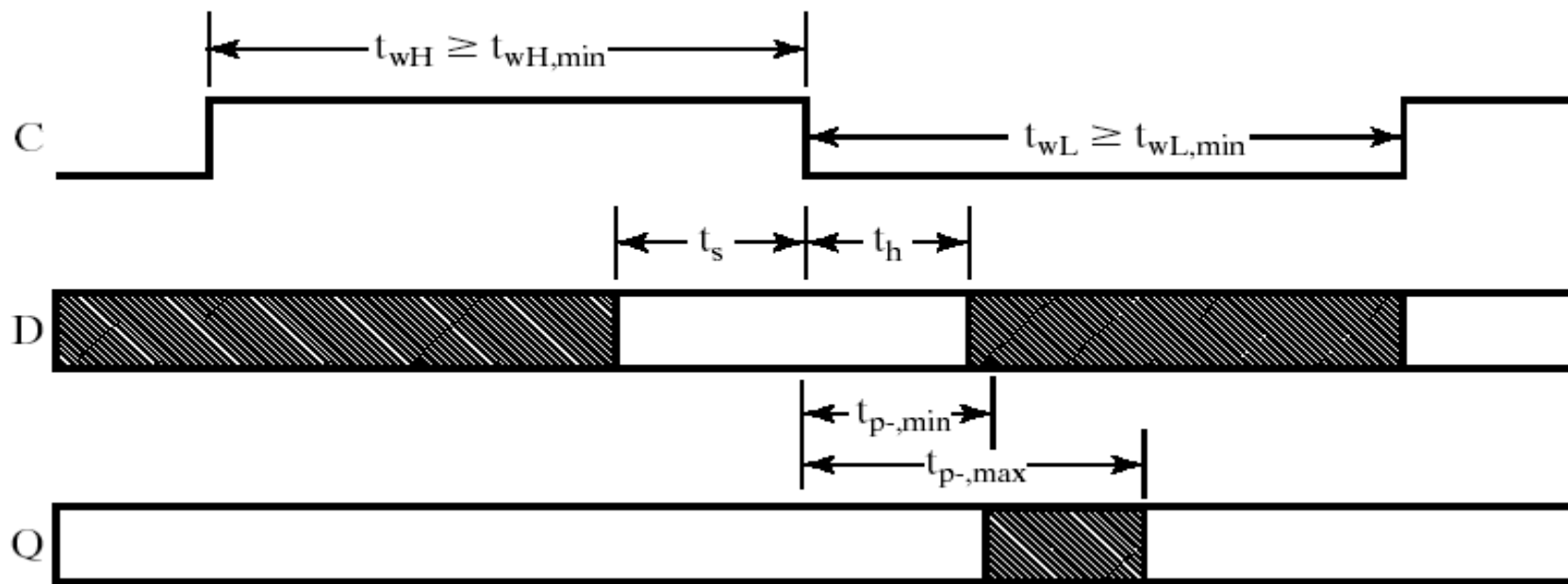






Clock Pulse Requirements

- Basically a max clock frequency = $1 / (t_{wH} + t_{wL})$
- Pulse (t_{wH}) cannot be too narrow. Why?

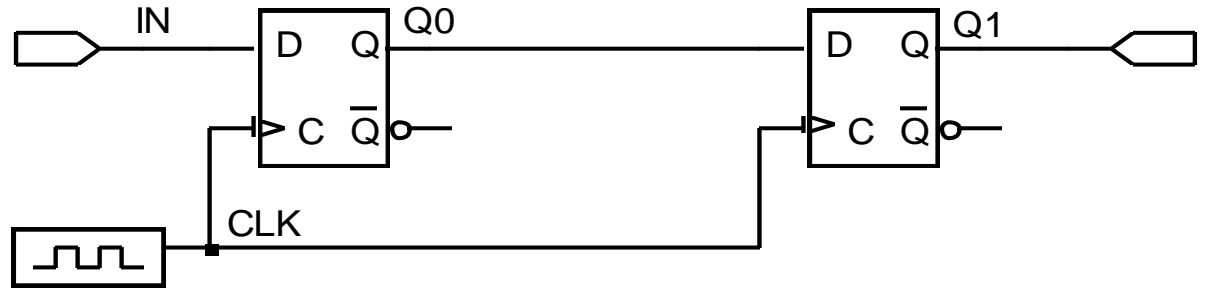




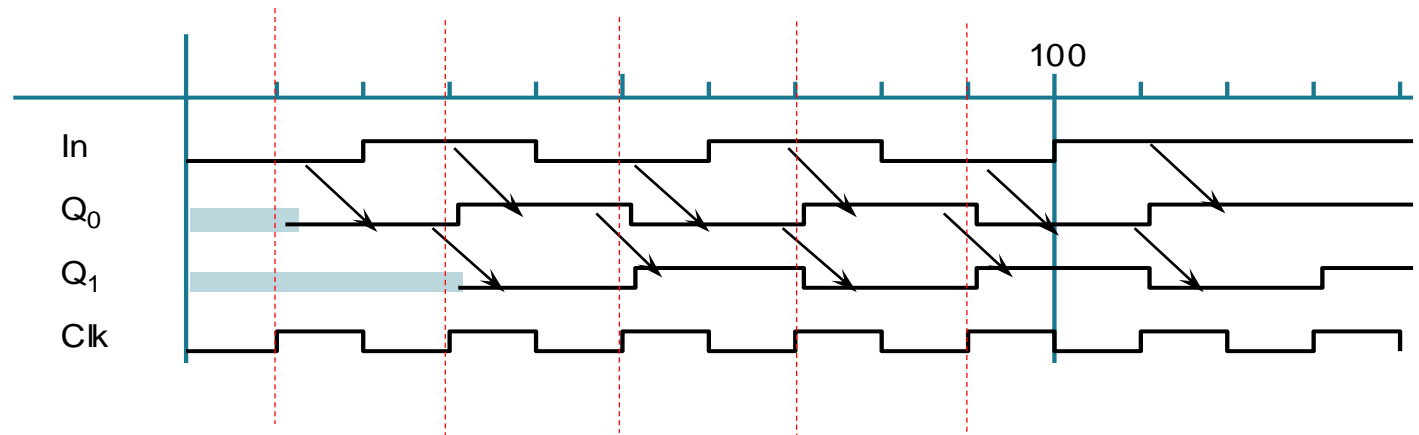
Cascaded Flip-Flops

Shift Register
Have S, R (preset, preclear)
inputs

New value to first stage
while second stage
obtains current value
of first stage

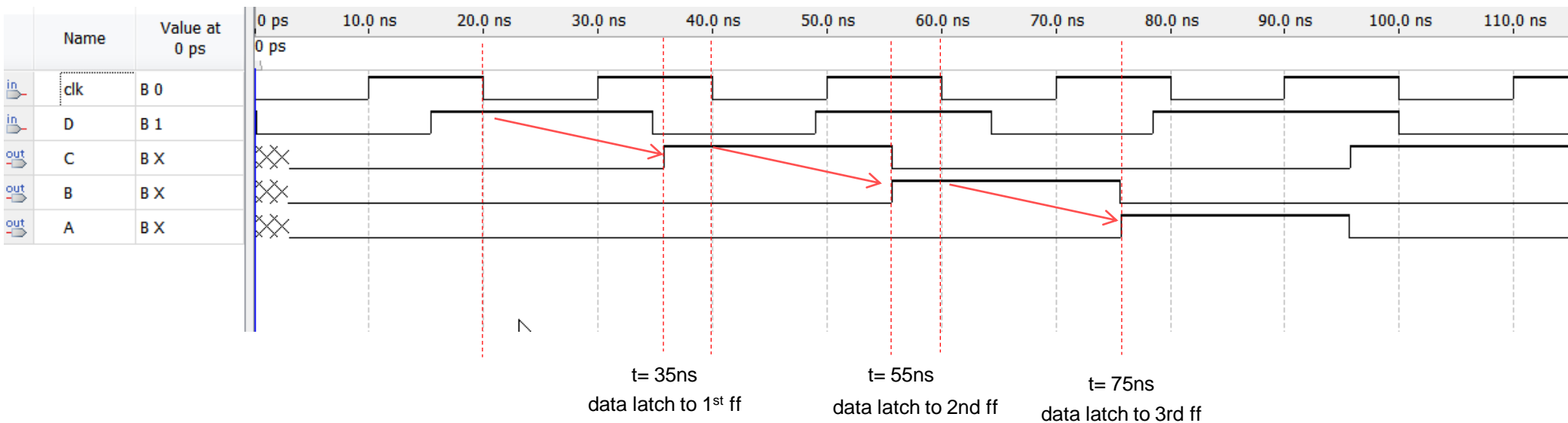


Correct Operation,
assuming positive
edge triggered FF





Cascaded flip-flop



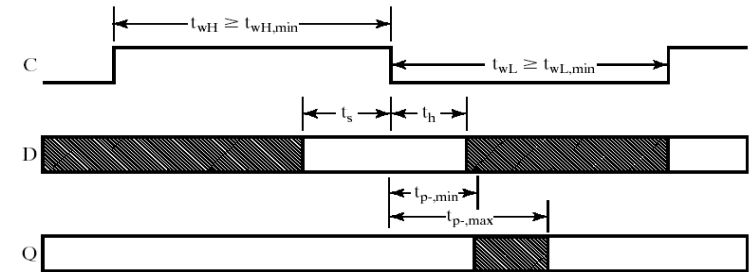
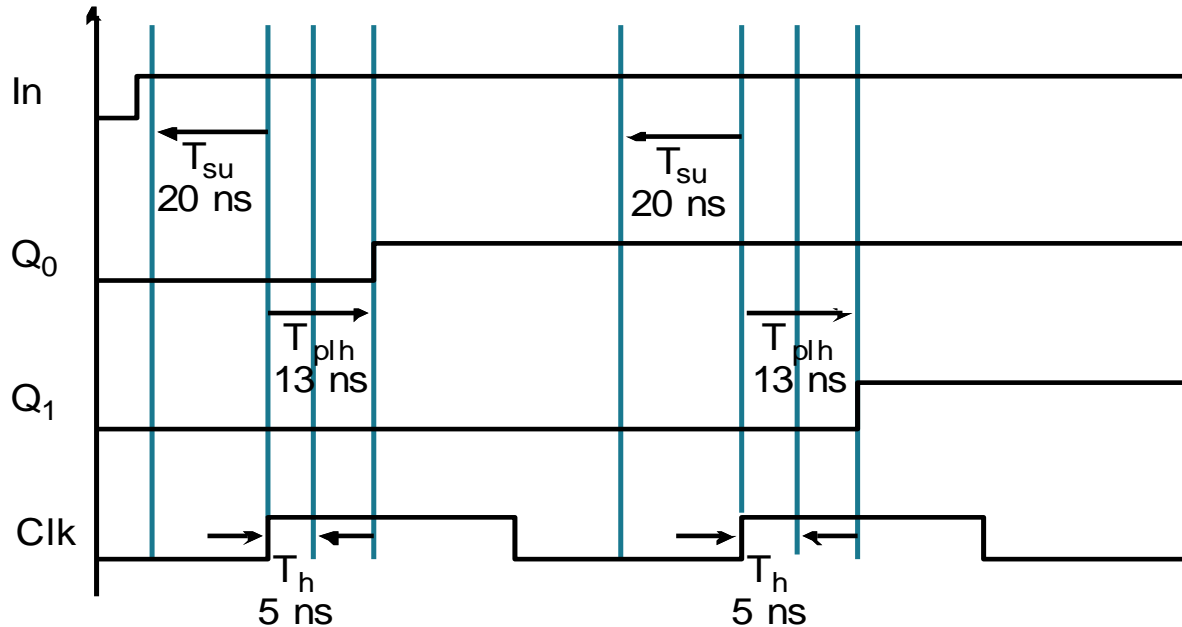
Total time taken to 3rd f/f/ is 75ns



Cascaded Flipflops

■ Why this works:

- Propagation delays far exceed hold times;
- Clock width (t_w) constraint must exceed setup time - **incorrect operation if clock period too short!**
- This guarantees following stage will latch current value before it is replaced by new value
- Assumes infinitely fast distribution of the clock



Timing constraints
guarantee proper
operation of
cascaded components

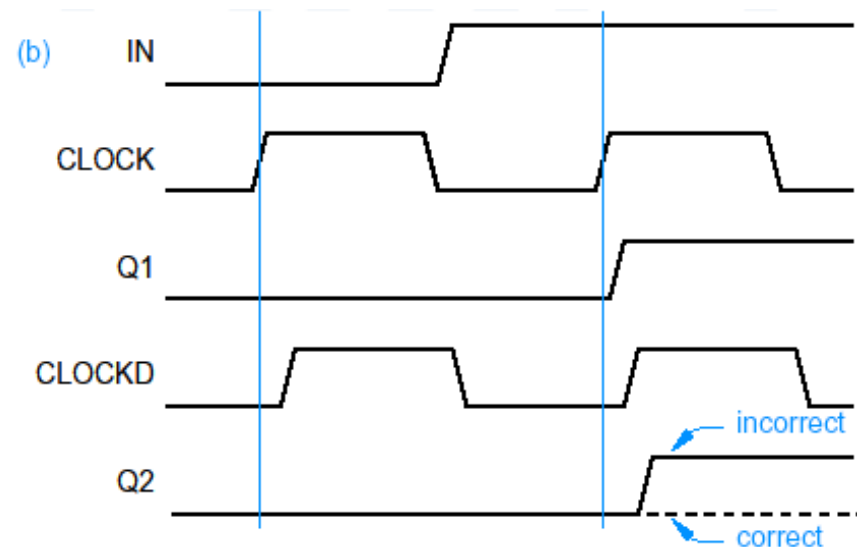
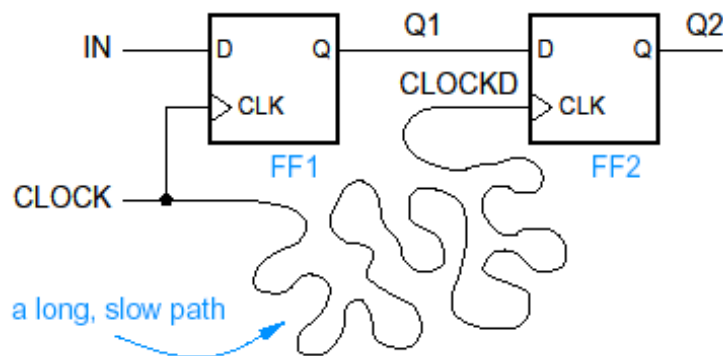
* T_{plh} - L to H propagation
delay



Clock Skew

■ Clock skew

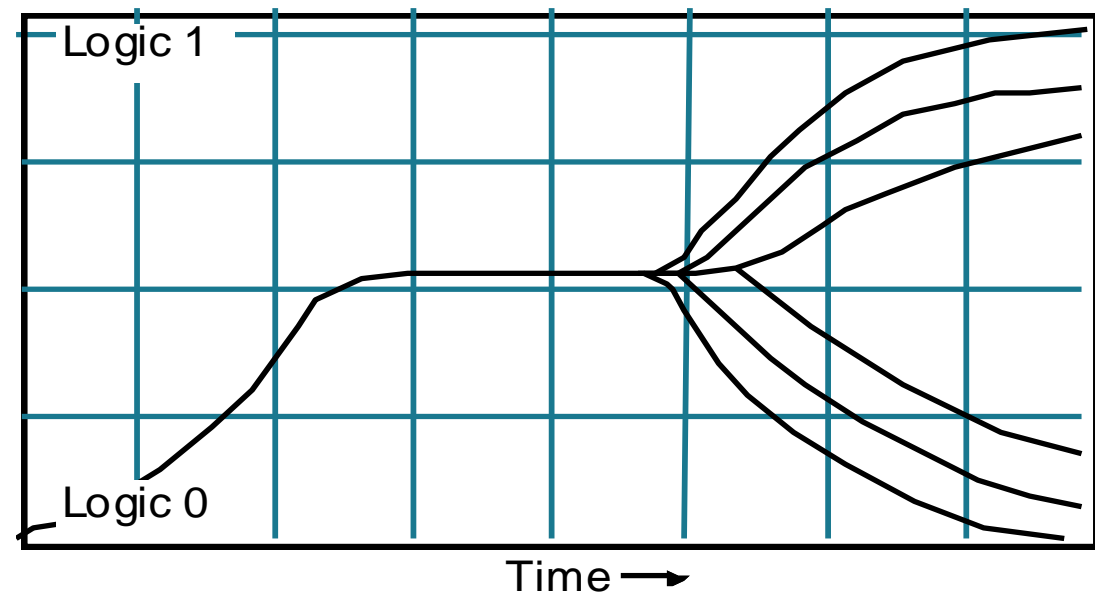
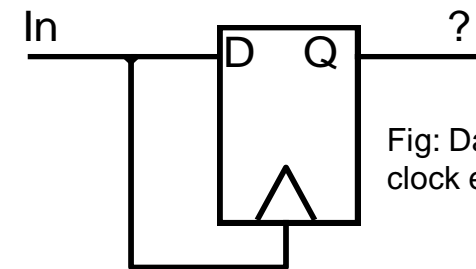
- Definition: difference between arrival times of the clock at different FFs
- Clock skew also caused by difference setup and hold times of different devices
- Correct behavior assumes next state of all storage elements determined by all storage elements at the same time
- Not possible in real systems!





Asynchronous Inputs and Metastability

- Asynchronous Inputs Are Dangerous!
 - Since they take effect immediately, glitches can be disastrous
- Synchronous inputs are greatly preferred!
- What happens when an asynchronous input is not synchronized?
- When FF input changes close to clock edge, the FF may enter the **metastable** state: neither a logic 0 nor a logic 1
- It may stay in this state an indefinite amount of time, although this is not likely in real circuits

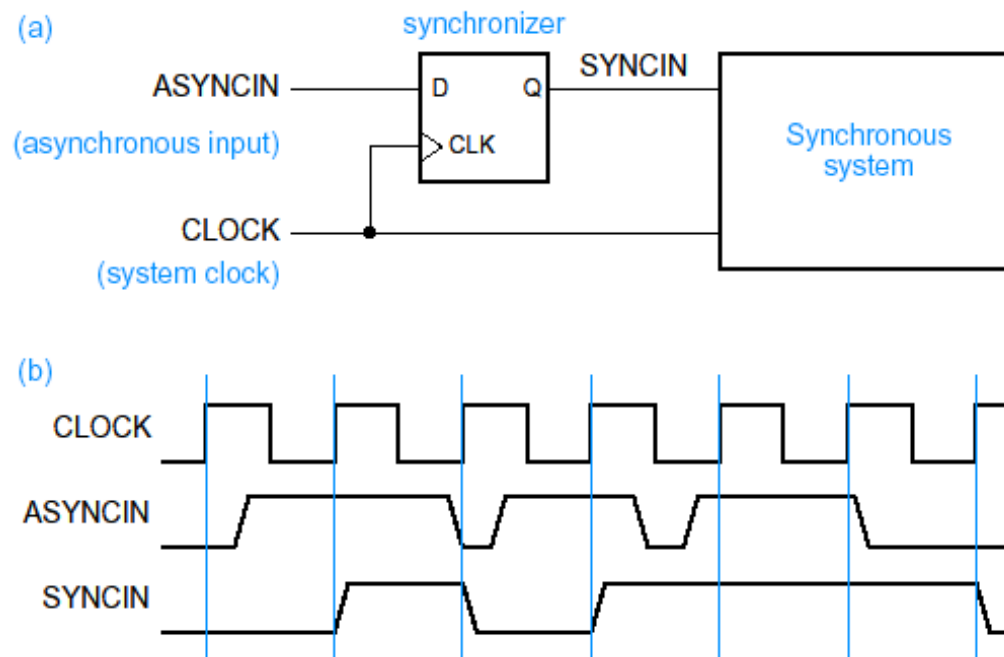


Oscilloscope Traces Demonstrating Synchronizer Failure and Eventual Decay to Steady State



Simple Synchronizer

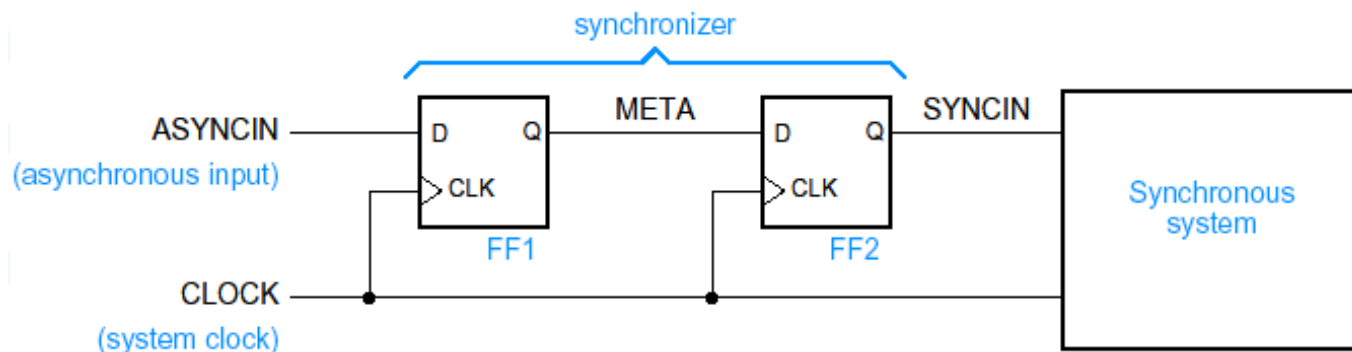
- But asynchronous inputs cannot be avoided
 - e.g., reset signal, memory wait signal
- Initial versions of commercial IC's that suffered metastability
 - Zilog Z-80 Serial I/O Interface
 - Intel 8048 microcontroller
 - AMD 29000 RISC microprocessor
- A *synchronizer* samples an asynchronous input and produces an output that meets the setup and hold times required in a synchronous system





A Better Synchronizer

- The probability of failure can never be reduced to 0, but it can be reduced
 - Synchronizer failure becomes a big problem for very high speed systems
- How to get a flip-flop out of the metastable state:
 - Force the flip-flop into a valid logic state using input signals that meet the published specifications for minimum pulse width, setup time, and so on.
 - Wait “long enough,” so the flip-flop comes out of metastability on its own
 - slow down the system clock
 - this gives the synchronizer more time to decay into a steady state
 - Use fastest possible logic in the synchronizer
 - S or AS TTL D-FFs are recommended (applicable only to TTL circuits)
 - Cascade two synchronizers



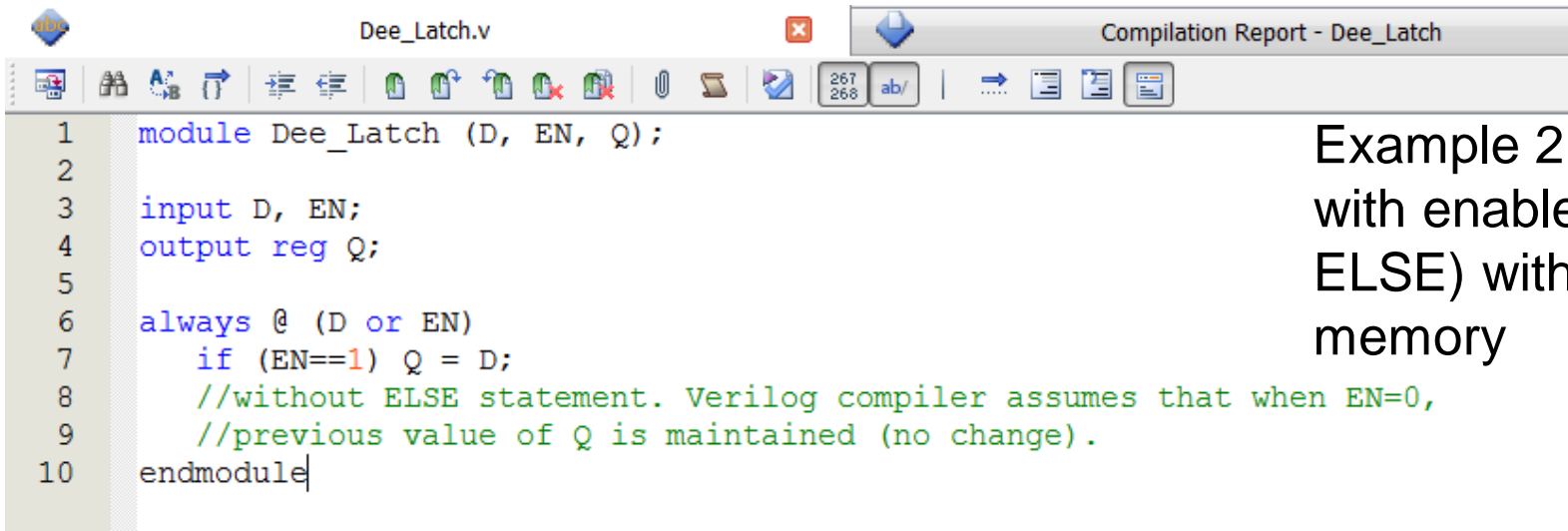


Level detection D-latch

```
//level-triggered D-latch
module latchD (en, D, Q, clk, q, d);
input en, D, clk, d;
output reg q, Q;

always @ (en or D) begin
    if (en==1) Q = D;
    else Q = Q; // this line can be omitted
                //en==0 is implied
end
```

Example 1 – D-latch with enable input (IF-ELSE)



```
1 module Dee_Latch (D, EN, Q);
2
3 input D, EN;
4 output reg Q;
5
6 always @ (D or EN)
7     if (EN==1) Q = D;
8     //without ELSE statement. Verilog compiler assumes that when EN=0,
9     //previous value of Q is maintained (no change).
10 endmodule
```

Example 2 – D-latch with enable input (IF-ELSE) with implied memory



Level detection D-latch

Example 2

– positive edge triggered D-latch

```
//edge-triggered D-latch
module latchDpos (CLK, D, Q);
input D, CLK;
output Q;
reg Q;

always @ (posedge CLK)
    Q =D;
endmodule
```

always @ (posedge clock or reset) ✗

always @ (posedge clock or negedge reset) ✓
// active low reset

- The output Q only changes as a result of the *event control operator (CLK)* change at the *event qualifier (posedge)*
- Both must be specified in the sensitivity list – event control & event qualifier

- If there are more than one event control operators (eg. clk, reset) in the sensitivity list then all must be written of the same event qualifier. Example.



Flip flop in Verilog

Verilog supports two types of assignments within always blocks, with subtly different behaviors : blocking & non-blocking assignments

Blocking Assignment

- In previous examples the “=” symbol is frequently used in assignment statements.
- When used in sequential block (or **always** block), these statements are called **blocking assignments** meaning the statements are evaluated in the order they are specified or written.
- The assignment is said to be blocked and the statement must complete its execution or update its value before next line of statement can be executed.
- The new updated value for a given variable will be used in evaluating all subsequent statements in the sequential block.



Blocking and Non-blocking assignments

Non-Blocking Assignment

- A non- blocking assignment construct is denoted with “ **<=** ” symbol.
- A non- blocking assignment allows one non-blocking assignment statement to execute and continues to the next statement without waiting for the current statement to complete execution,



Blocking and Non-blocking assignments

- **Blocking assignment:** evaluation and assignment are immediate and in sequential order

```
always @ (a or b or c) begin
```

```
x = a | b;           // 1. Evaluate a | b, assign result to x
```

```
y = a ^ b ^ c;       // 2. Evaluate a^b^c, assign result to y
```

```
z = b & ~c; end      // 3. Evaluate b&(~c), assign result to z
```

- **Non-blocking assignment:** all assignments deferred until all right-hand sides have been evaluated (end of simulation time step)

```
always @ (a or b or c) begin
```

```
x <= a | b;           // 1. Evaluate a | b but defer assignment of x
```

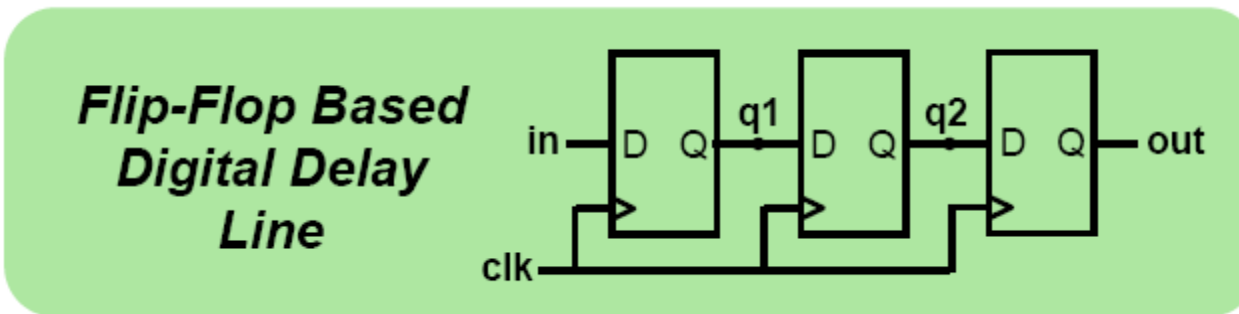
```
y <= a ^ b ^ c;       //2. Evaluate a^b^c but defer assignment of y
```

```
z <= b & ~c;          //3. Evaluate b&(~c) but defer assignment of z
```

```
end                   //4. Assign x, y, and z with their new values
```



Blocking and Non-Blocking Assignments



- Will non-blocking and blocking assignments both produce the desired result?

```
module blocking (in, clk, out);  
  input in, clk;  
  output out;  
  reg q1, q2, out;  
  always @ (posedge clk) begin  
    q1 = in;  
    q2 = q1;  
    out = q2;  
  end  
endmodule
```

```
module nonblocking (in, clk, out);  
  input in, clk;  
  output out;  
  reg q1, q2, out;  
  always @ (posedge clk) begin  
    q1 <= in;  
    q2 <= q1;  
    out <= q2;  
  end  
endmodule
```



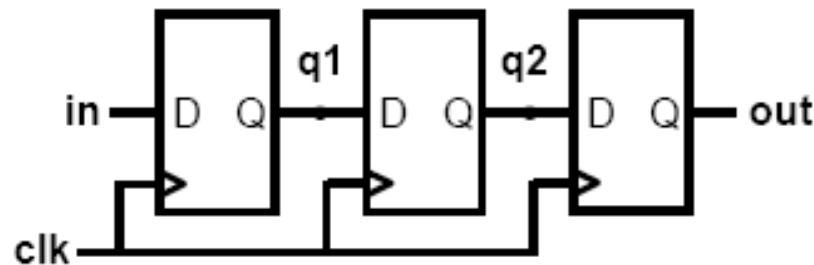

Blocking and Non-Blocking Assignments

```
module blocking (in, clk, out);  
input in, clk;  
output out;  
reg q1, q2, out;  
always @ (posedge clk) begin  
    q1 = in;  
    q2 = q1;  
    out = q2;  
end  
endmodule
```

```
module nonblocking (in, clk, out);  
input in, clk;  
output out;  
reg q1, q2, out;  
always @ (posedge clk) begin  
    q1 <= in;  
    q2 <= q1;  
    out <= q2;  
end  
endmodule
```

// 1) q1 = in;
// 2) q2 = q1 >>> which means q2 = in;
// 3) out = q2; >>> which means out = in

∴ at each clock edge out = in



“At each rising clock edge, *q1*, *q2*, and *out* **simultaneously receive the old values** of *in*, *q1*, and *q2*.”



Blocking and Non-Blocking Assignments

- Blocking assignments do not reflect the intrinsic behavior of multi-stage sequential logic
- **Guideline: use nonblocking assignments for sequential always blocks**

“ **<=** ”

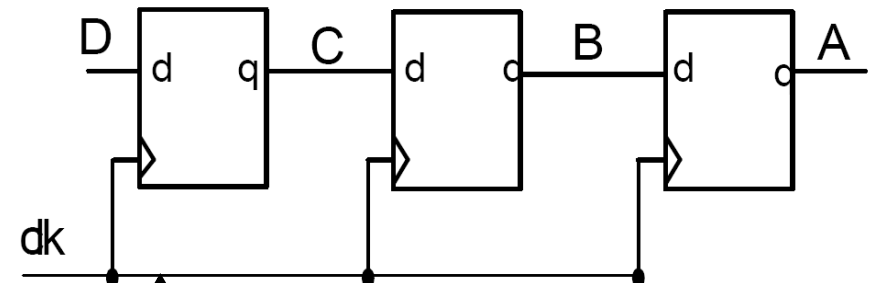


Examples of Blocking and Non-Blocking Assignment

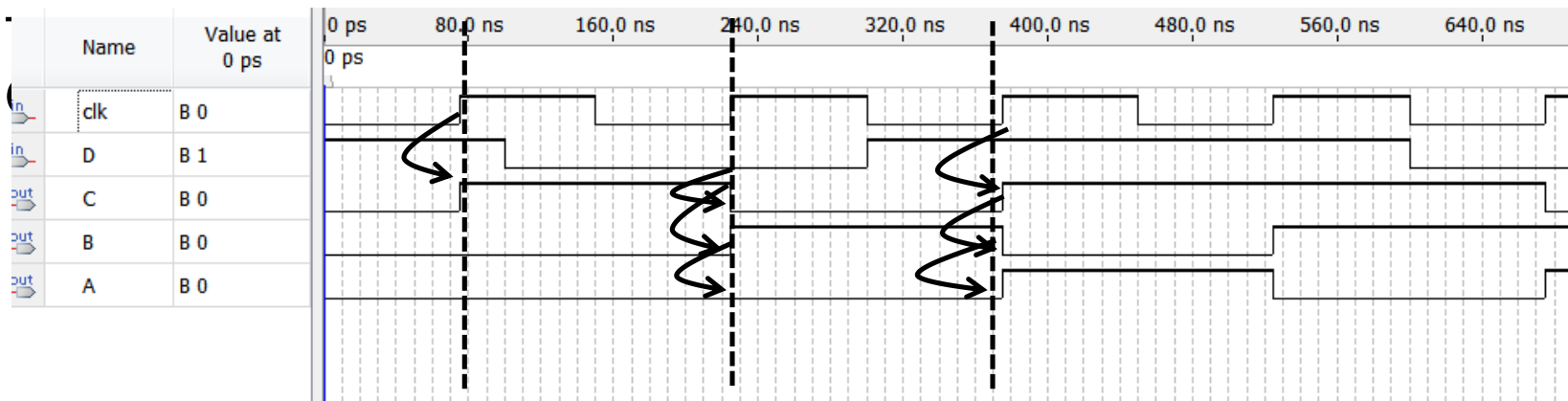
Blocking Assignment

```
module shiftregX1 (D, clk, A,B, C)
input DATA, clk;
output A, B, C;
reg A, B, C;

always @ posedge clk) begin
    A = B;
    B = C;
    C = D;
end
endmodule
```



This code infers a shift register





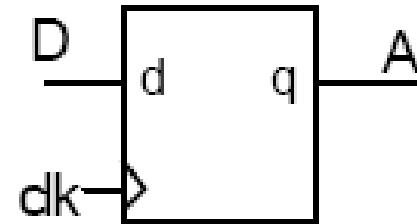
Examples of Blocking and Non-Blocking Assignment

Blocking Assignment

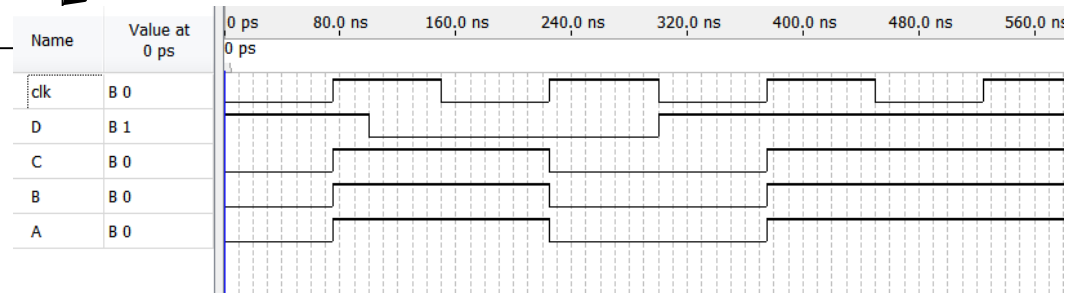
```
module shiftregY1 (D, clk, A, B, C)
input DATA, clk;
output A, B, C;
reg A, B, C;

always @ posedge clk) begin
    C = D;
    B = C;
    A = B;
end
endmodule
```

The list of sequence executes in sequential order, from top to bottom



This code infers a shift register



Conclusion: The order of the statement matters in blocking assignment

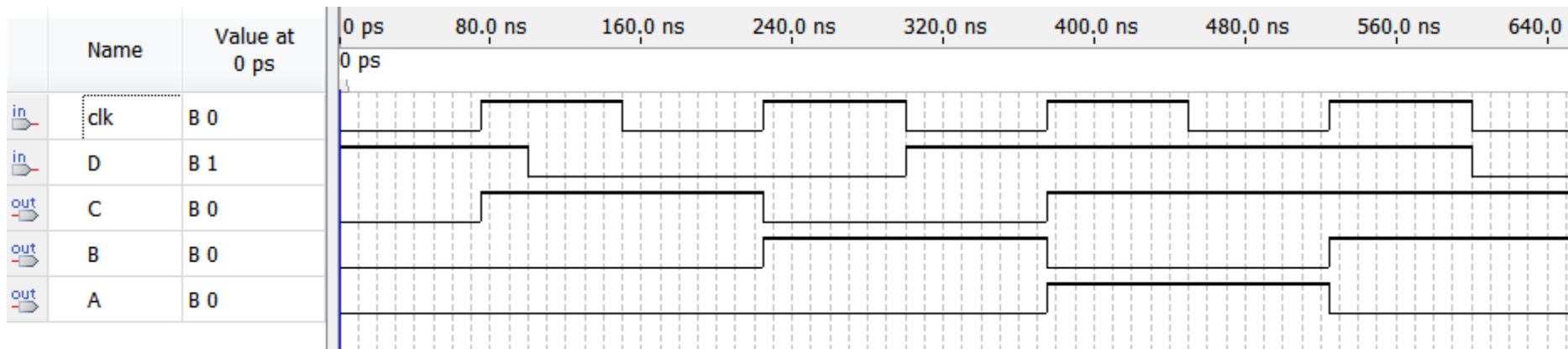


Examples of Blocking and Non-Blocking Assignment

Non-Blocking Assignment

```
module shiftregX2 (D, clk, A,B, C)
input DATA, clk;
output A, B, C;
reg A, B, C;

always @ (posedge clk) begin
    A <= B;
    B <= C;
    C <= D;
end
endmodule
```





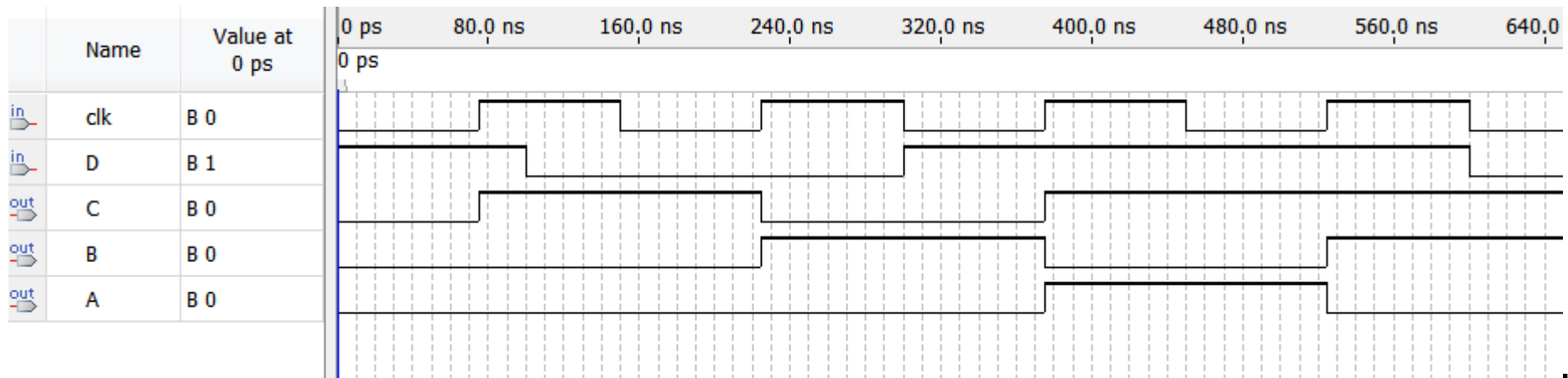
Examples of Blocking and Non-Blocking Assignment

Non-Blocking Assignment

```
module shiftregX2 (D, clk, A,B, C)
input DATA, clk;
output A, B, C;
reg A, B, C;

always @ (posedge clk) begin
    C <= D;
    B <= C;
    A <= B;
end
endmodule
```

So, the order of the statement does
not matter in non-blocking assignment



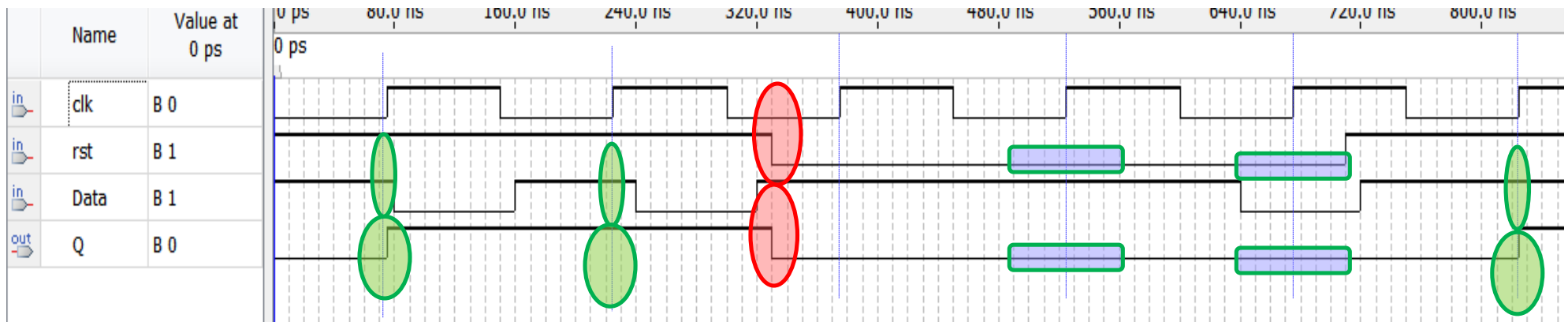


D-FF with Asynchronous inputs

Example 2 – D-latch with asynchronous input reset

```
//D-latch with active-low asynchronous RESET
module latchD (rst, Data, clk, Q);
input rst, Data, clk;
output reg Q;

always @ (negedge rst or posedge clk) begin
    if (~rst) Q <= 0;
    else Q <= Data;
end
endmodule
```





Level detection D-latch

Example 2 – D-latch with asynchronous input reset

```
//D-latch with active-low RESET input
module dlatch_reset (data, en, reset, Q);
input data, en, reset;
output Q;
reg Q;

always @ (en or reset or data) begin
    if (~reset) Q <= 1'b0;
    else if (en) Q <= data;
    // else (en==0) implied memory
end
endmodule
```




D-latch with Asynchronous inputs

Example 2 – D-latch with active-low asynchronous reset and active-high preset

```
//D-latch with active-low asynchronous RESET and active-high PRESET
module latchD (rst, prst, Data, clk, Q);
input rst, prst, Data, clk;
output reg Q;

always @ (negedge rst or posedge prst or negedge clk) begin
    if (~rst) Q <=0;
    else if (prst) Q <=1; // this line onward rst=1 (not active) >> if
rst                        // thus rst NOT active prst active
    else Q <= Data;       // this line onward prst = 1 (not active)
end
endmodule
```

