Unit 13

Sequential Logic Constructs

Learning Outcomes

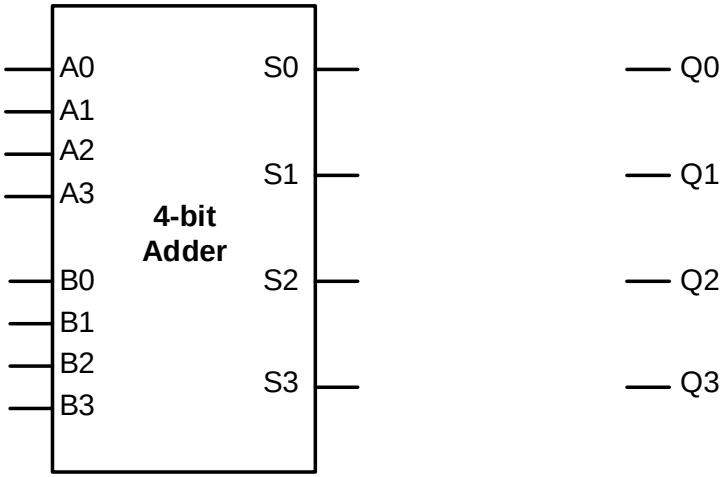
- I understand the difference between level-sensitive and edge-sensitive
- I understand how to create an edge-triggered FF from 2 latches

How sequential building blocks work

LATCHES AND FLIP-FLOPS

Sequential Logic

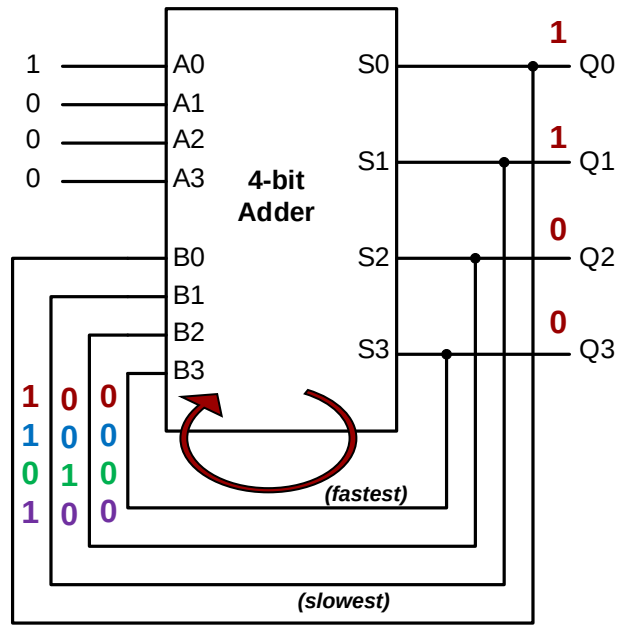
- Suppose we want to build a **4-bit counter** which produces a 4-bit output Q whose value increases by 1 every time period
- Possible solution: Route the outputs back to the inputs so we can add 1 to the current counter value (i.e. Q+1)



Sequential Logic

- Suppose we want to build a **4-bit counter** which produces a 4-bit output Z whose value increases by 1 every time period
- Possible solution: Route the outputs back to the inputs so we can add 1 to the current counter value (i.e. Q+1)
- Problem 1: No way to initialize sum
- Problem 2: Outputs can race around to inputs with different delays leading to arbitrary output values

Possible Solution

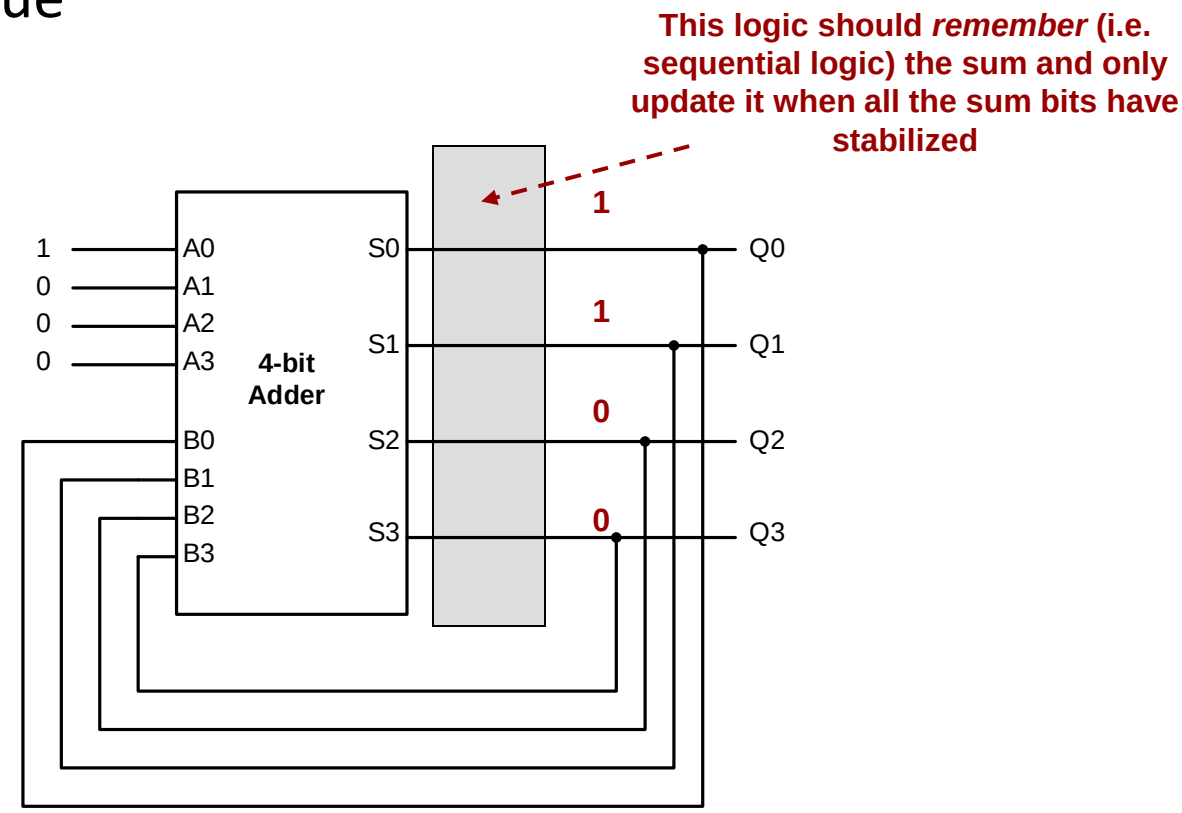


Suppose logic for S0 take delay, d, while S1 takes 2*d, S2 takes 3*d, and S3 = 4*d

red=value at t
blue=value at t+1
green=value at t+2
purple=value at t+3

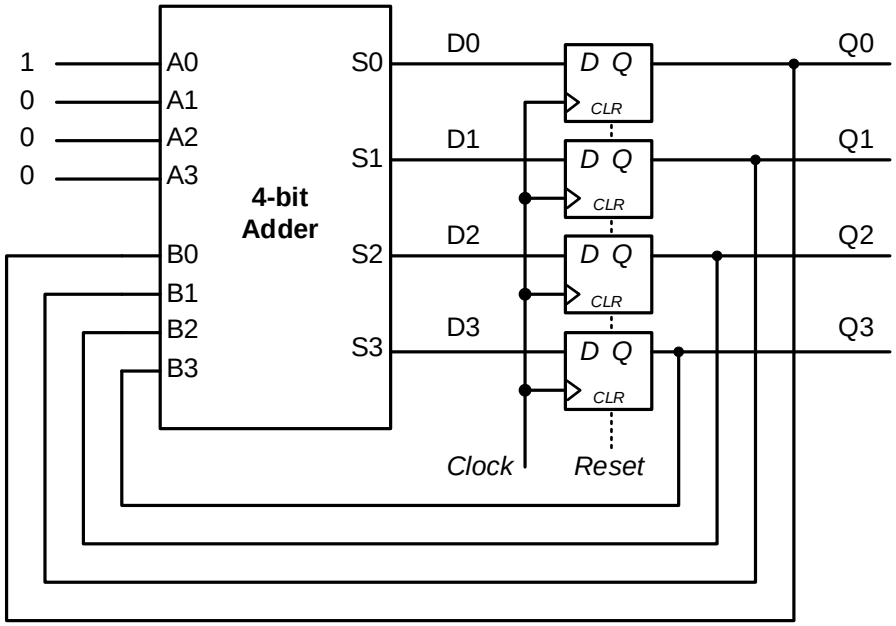
Sequential Logic

- Add logic at outputs to help initialize the output AND to synchronize and hold the output until we are ready to update to the next value



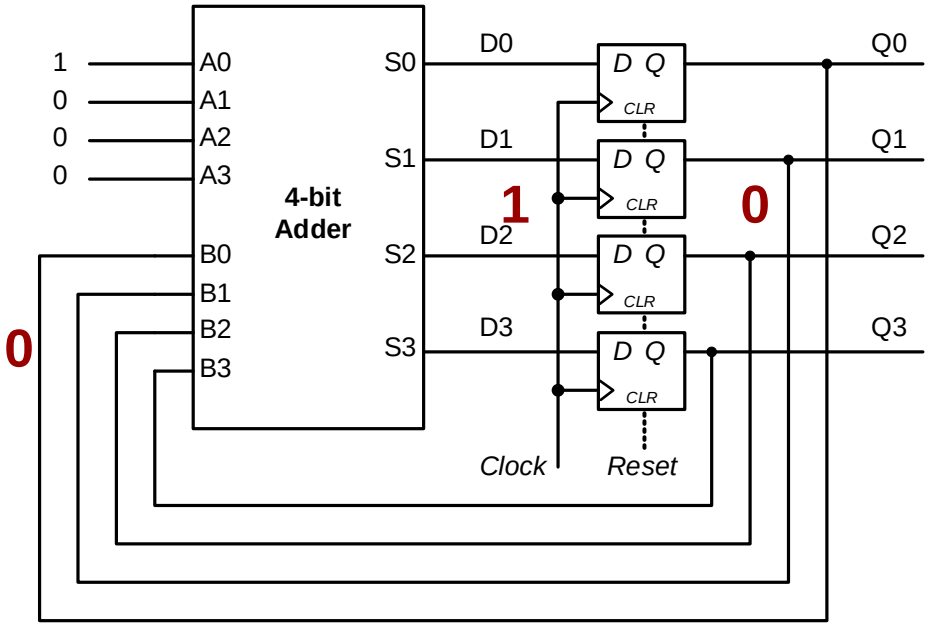
Sequence Adder

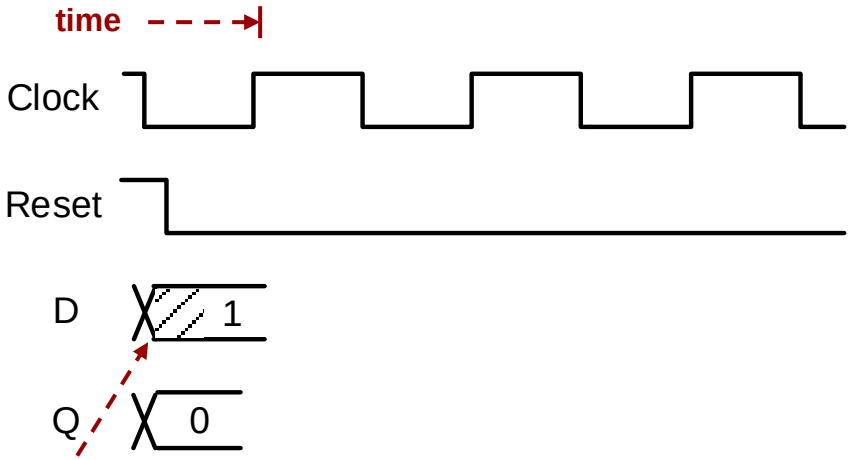
- Q should only update once per clock cycle (time unit)
- That is why we will use a register (flip-flops) to ensure the outputs can only update once per cycle



Sequence Adder

- The Reset (aka Clear) input on the register will cause Q to be initialized to 0, but then Q can't change until the next positive edge
- That means we will just keep adding 0 + 1 = 1

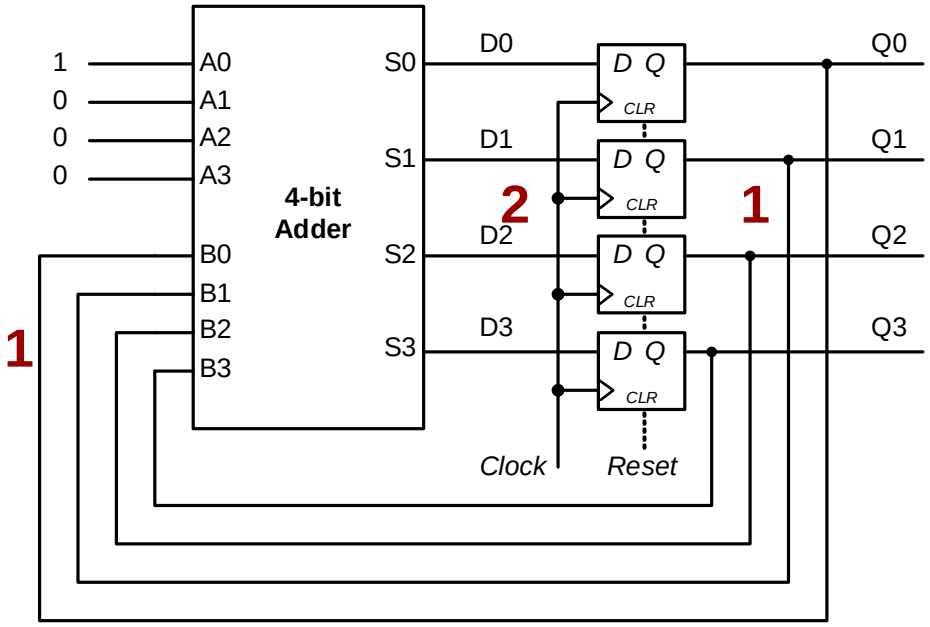


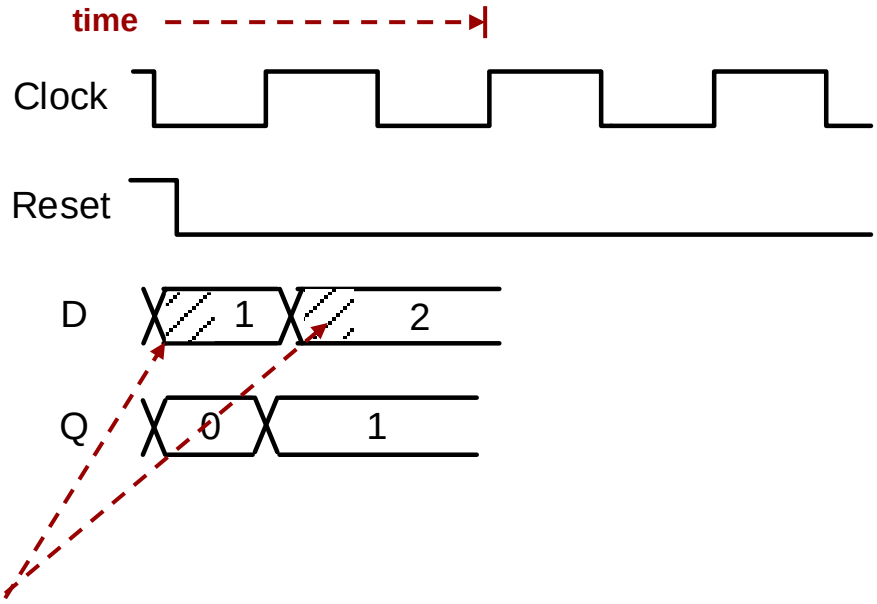


During this time the adder outputs are still being computed and the outputs may be arbitrary.

Sequence Adder

- At the edge the flip-flops will sample the D inputs and then remember 1 until the next positive edge
- The adder will then add 1+1=2

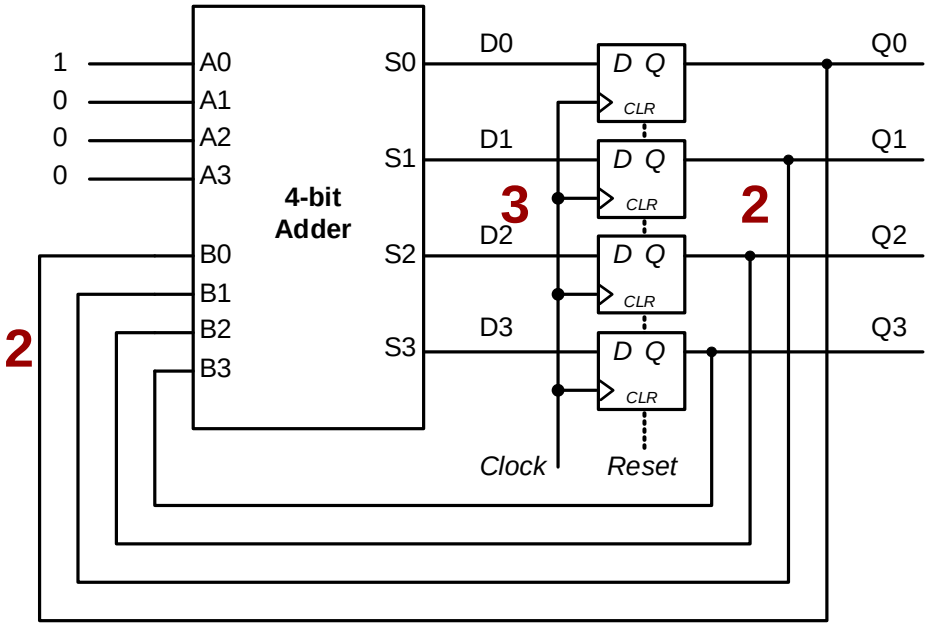


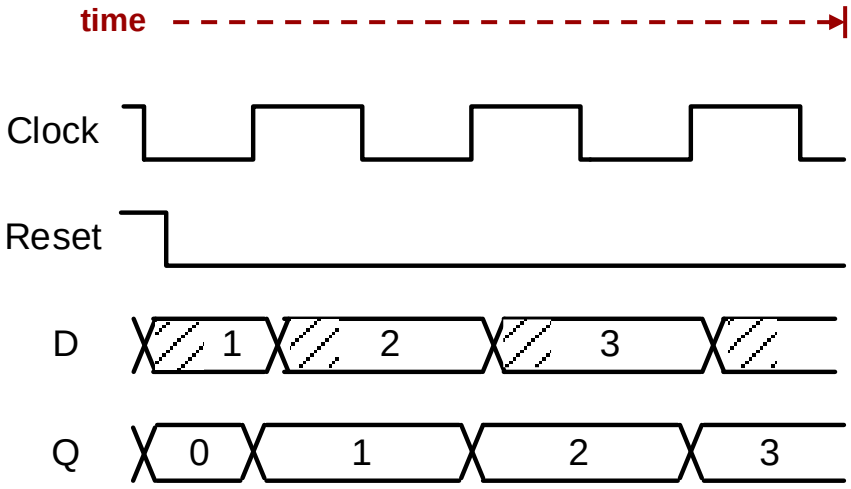


During this time the adder outputs are still being computed and the outputs may be arbitrary.

Sequence Adder

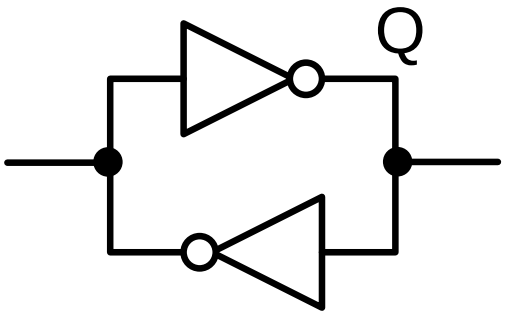
- The register will capture the adder output on each clock edge



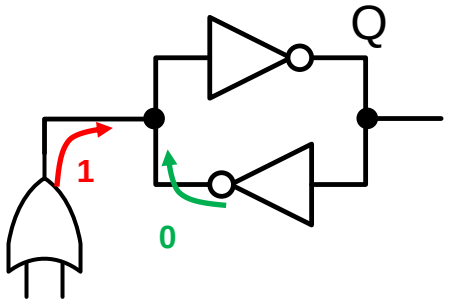


Sequential Logic

- But how do flip-flops work?
- Our first goal will be to design a circuit that can remember one bit of information
- Easiest approach…

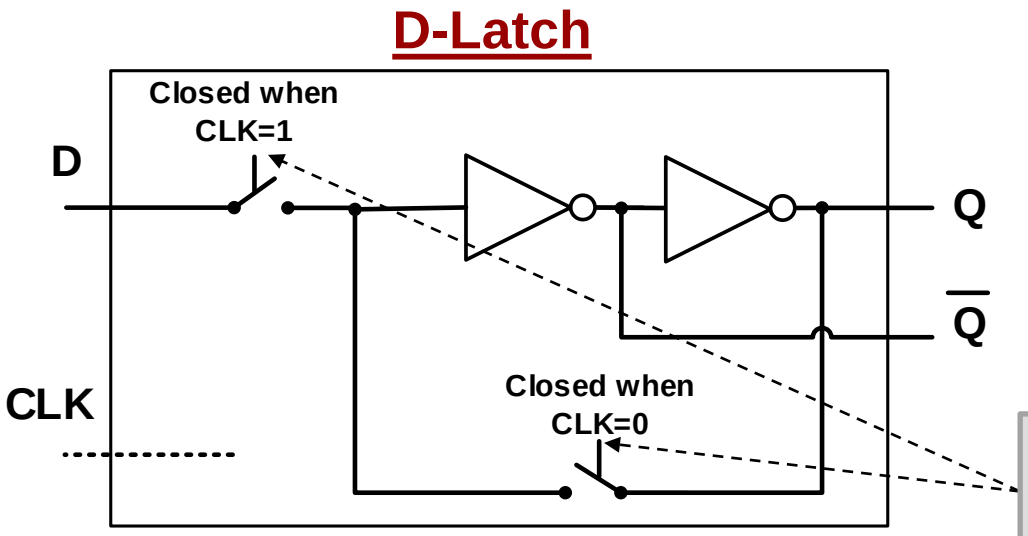


- But how do you change the input?
  - A signal should only have one driver



D-Latches

- The primary building block of sequential logic is a D-Latch
- D-Latches (Data latches) store/remember/hold data when the clock is low (CLK=0) and pass data when the clock is high (CLK=1)

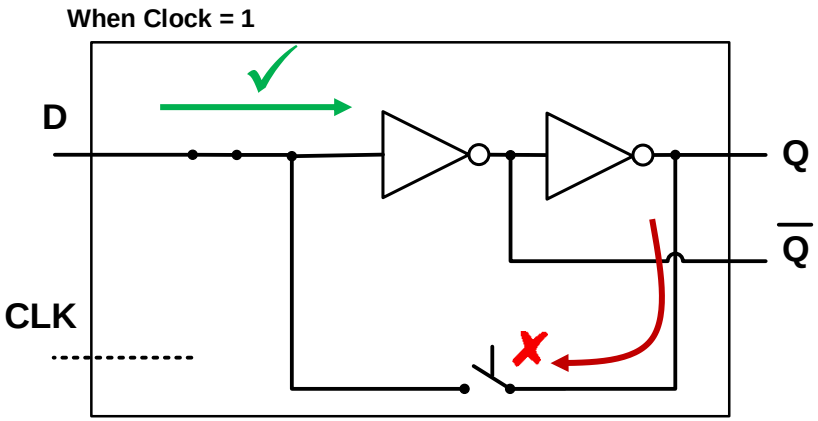


Transparent & Hold Mode of D-Latches

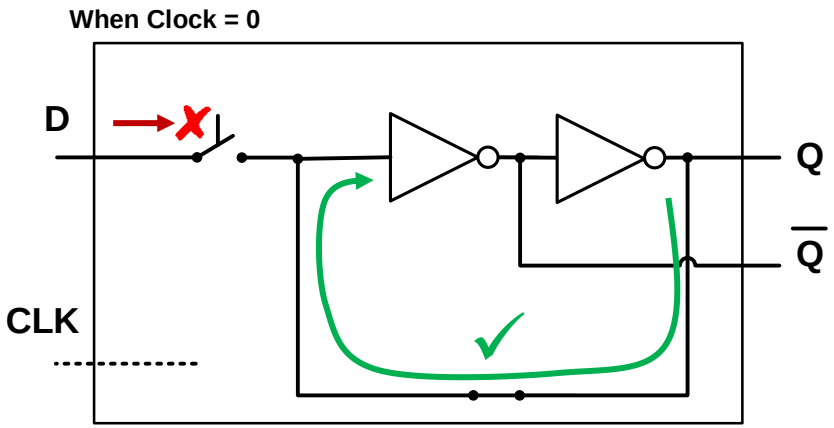
- The D-Latch operates in either transparent or hold mode based on the clock value

C	D	Q	Q'
0	x	Q_0	Q_0'
1	0	0	1
1	1	1	0

Function Table
Description of D-Latch



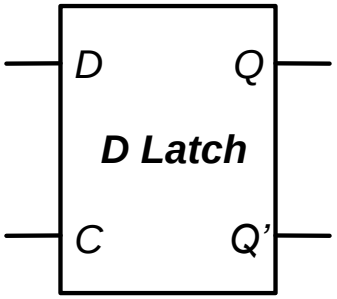
Transparent Mode
(Q=D when CLK=1)



Hold Mode
(Q=Q_0 when CLK=0)

D-Latches

Hold Mode



C	D	Q	Q'
0	x	Q_0	Q_0'
1	0	0	1
1	1	1	0

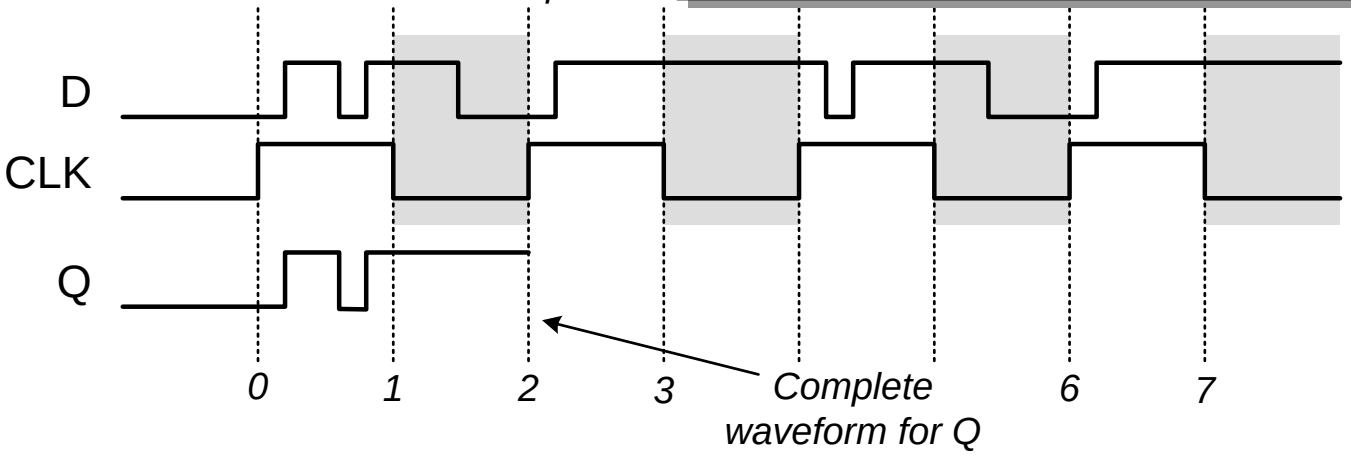
Hold Mode

Transparent Mode

D-LATCH
7475

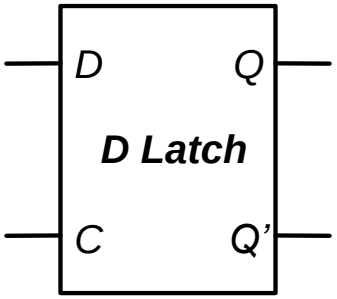
As clock is LOW, don't look at the D input

Triggering Rule: The Q output follow the D input (i.e. Q=D) when the clock or gate input is high (i.e. the latch is enabled). When the latch is disabled (Clock = LOW) the output remains put.



D-Latches

Hold Mode

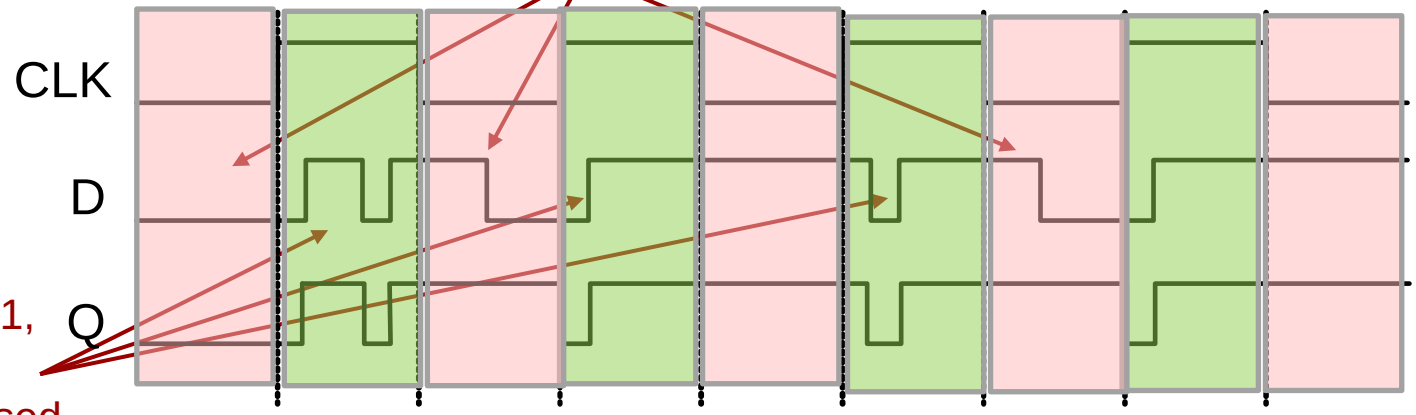


C	D	Q	Q'
0	x	Q_0	Q_0'
1	0	0	1
1	1	1	0

Hold Mode

Transparent Mode

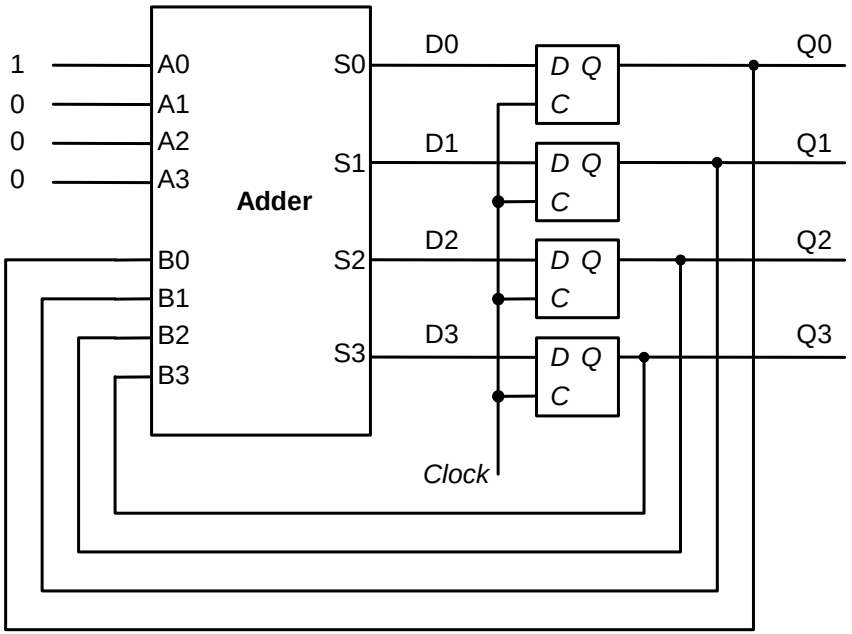
When C=0, outputs don't change no matter what the inputs do



When C=1, outputs change based on inputs

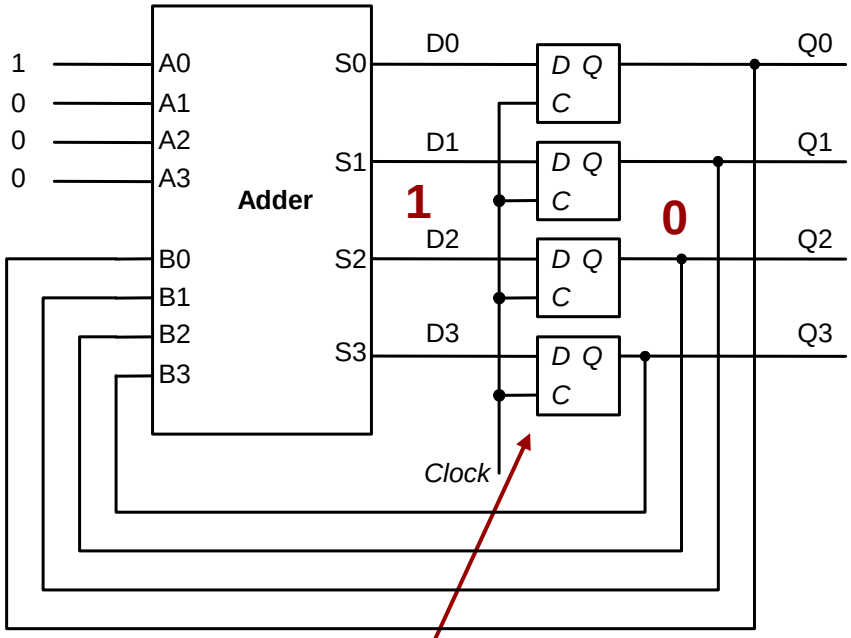
Adding a Sequence of Numbers

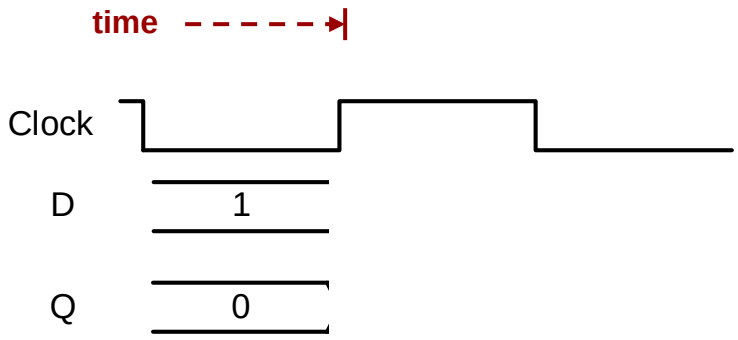
- What if we put D-Latches at the outputs



Adding a Sequence of Numbers

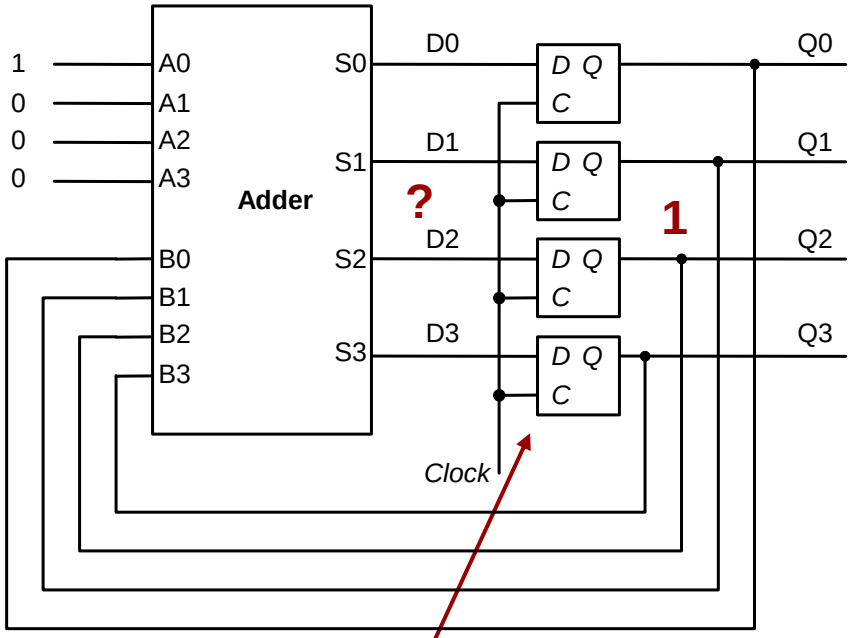
- Since the clock starts off low, the outputs of the latches can't change and just hold at 0
  - So far, so good. There is no uncontrolled feedback loop

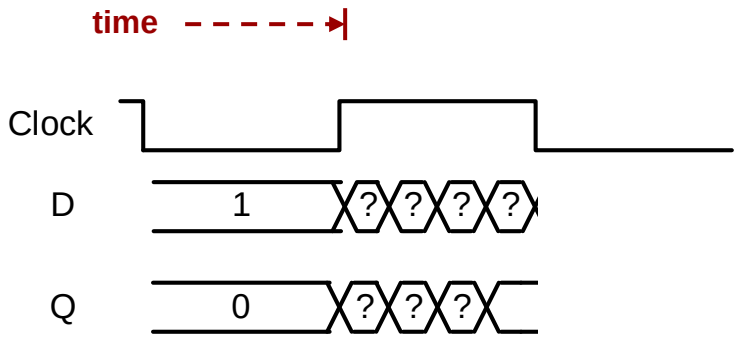




Adding a Sequence of Numbers

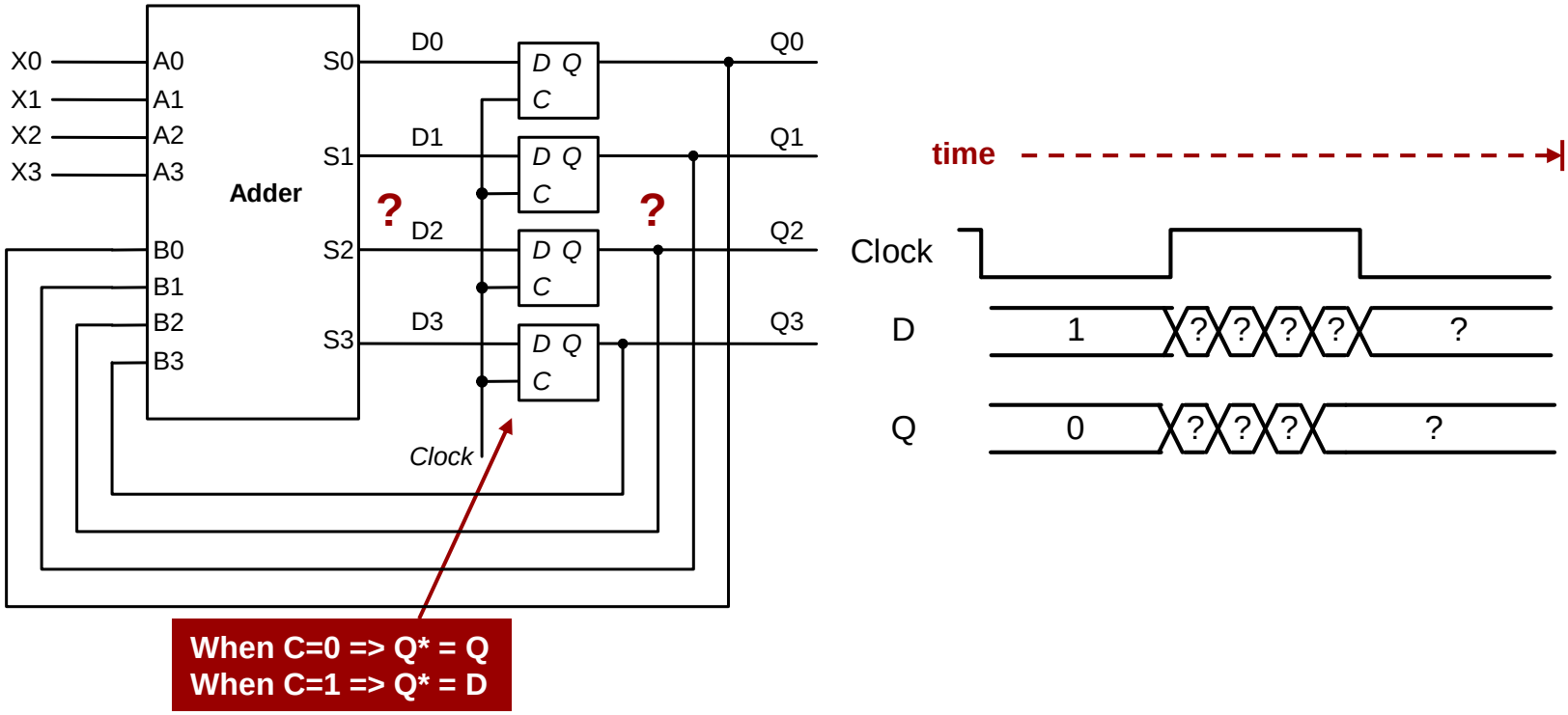
- When the clock goes high the D input is allowed to pass to Q which then loops back with the same arbitrary timing discussed earlier





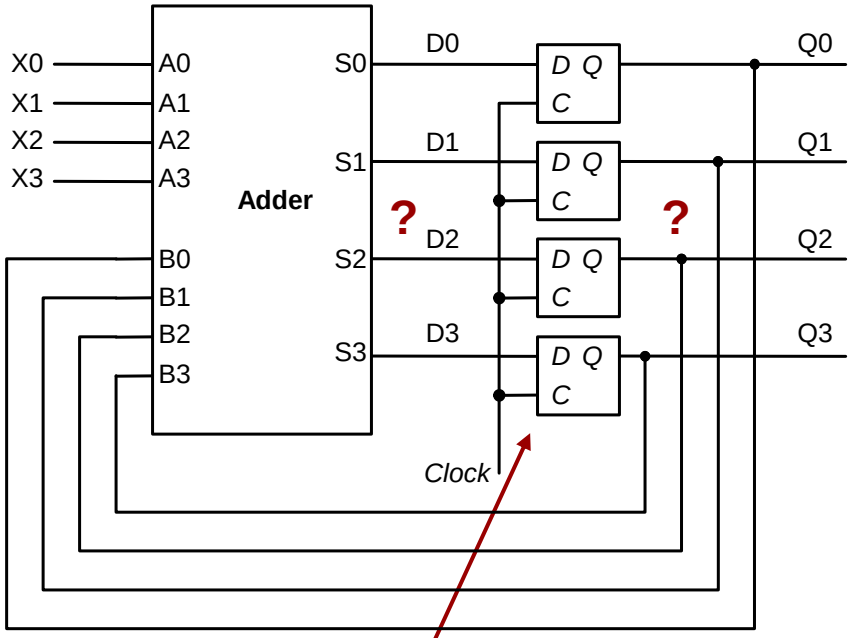
Adding a Sequence of Numbers

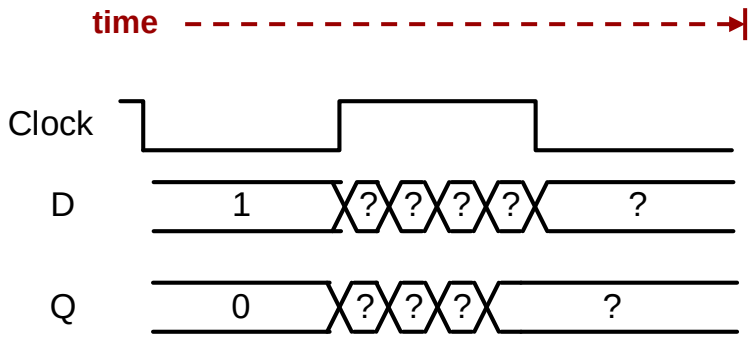
- When the clock goes low again, the outputs will stop changing but the value we are storing may be **arbitrary**



Adding a Sequence of Numbers

- Latches clearly don't work
- The goal should be to get ***one change of the outputs per clock period***

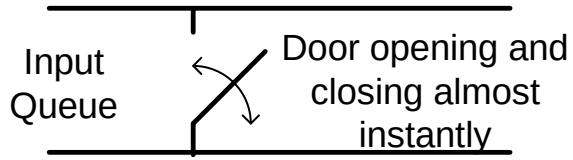


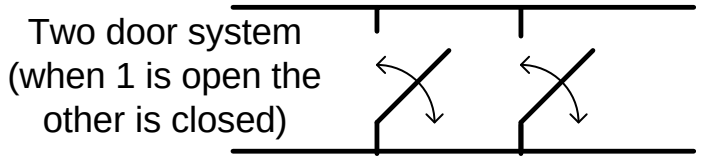


BUILDING A FLIP FLOP

Building an Edge-Triggered Device

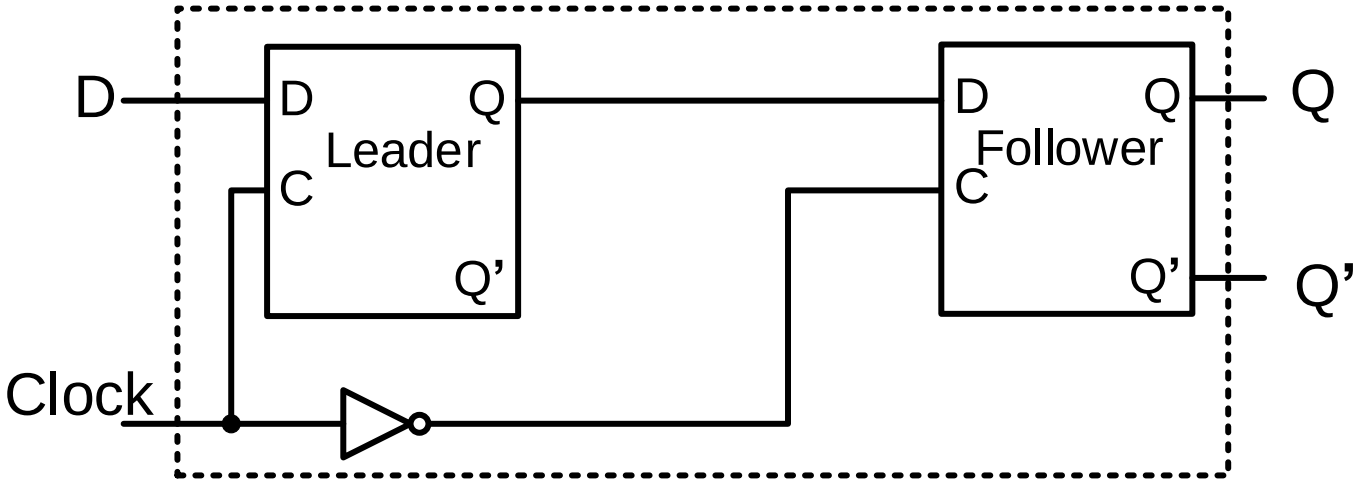
- We generally build FFs from latches
- To build a device that can only change at 1 instant (clock edge) we can:
  - Try to only enable 1 latch for a small instant in time
  - Use two latches running on opposite clock phases





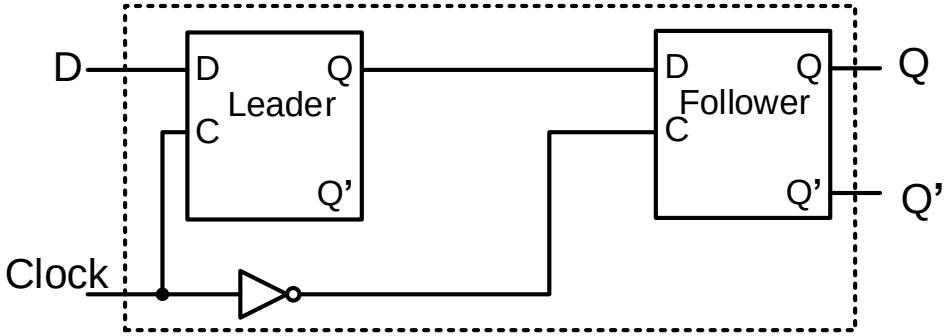
Leader-Follower D-FF

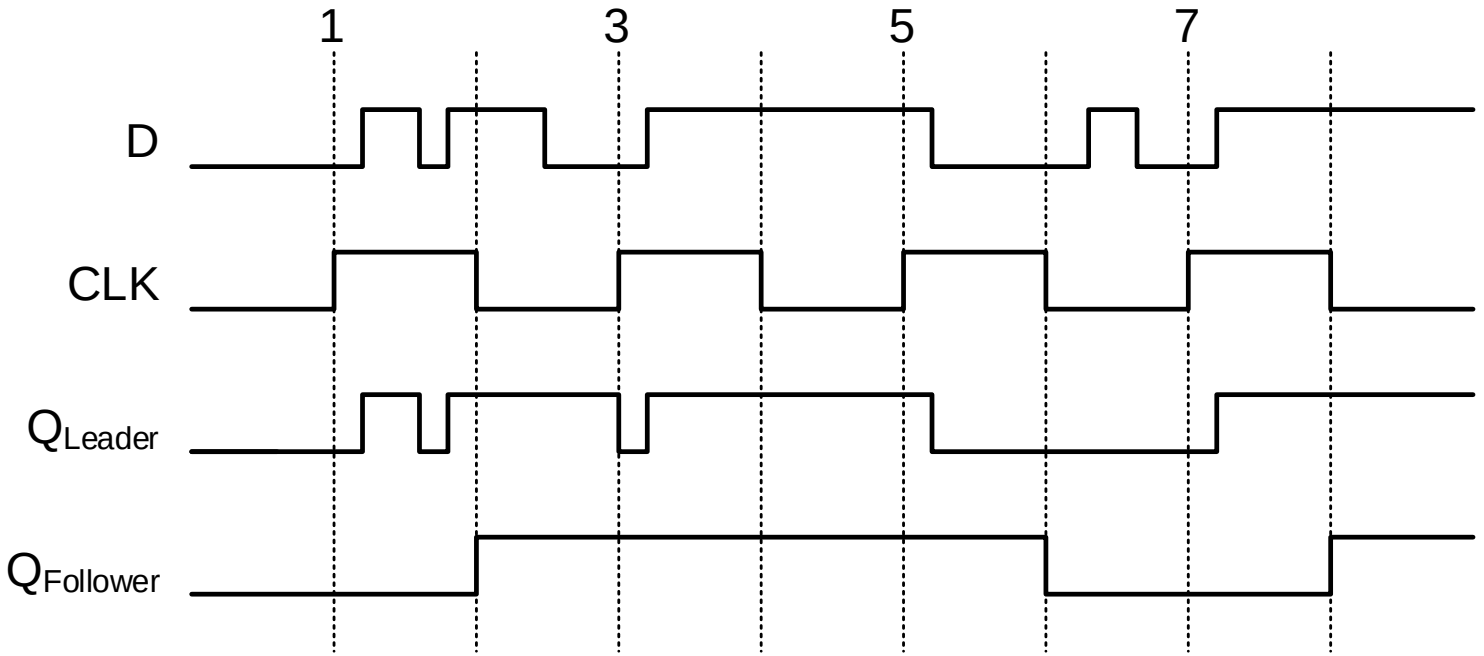
- To build an **edge-triggered** D-FF we can use two D-Latches
  - The configuration below forms a negative-edge triggered FF



These 2 latches form a flip-flop

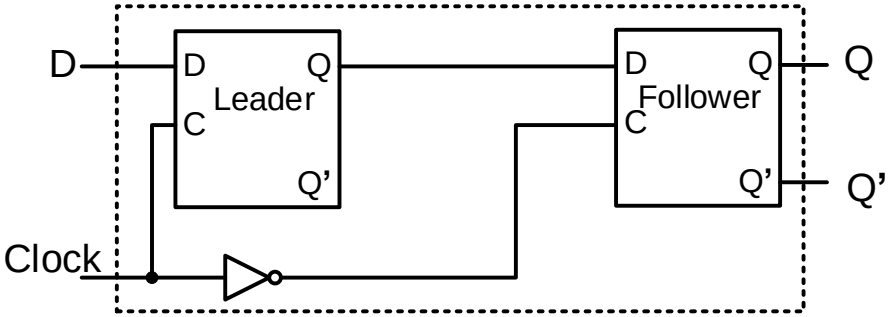
Complete the Waveform



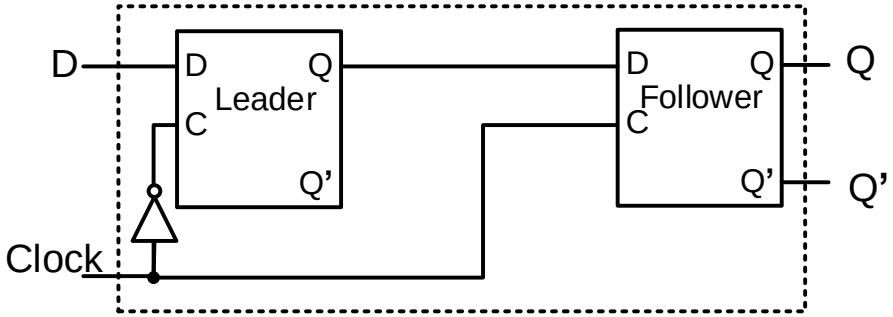


Leader-Follower D-FF

- To implement a positive edge-triggered D-FF change the clock inversion



Negative-Edge Triggered



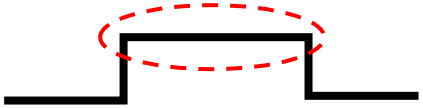
Positive-Edge Triggered

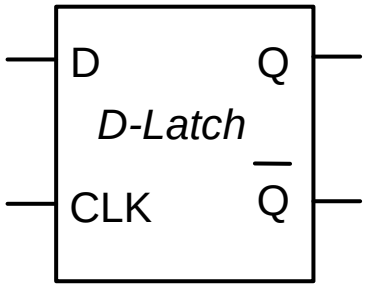
FLIP-FLOPS

Flip-Flops vs. Latches

Latches

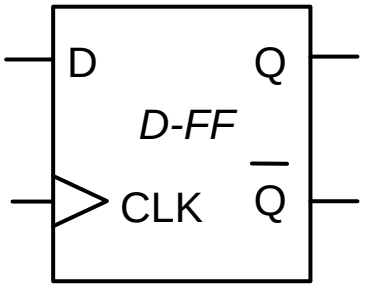
- Asynchronous
- Clock/Enable input
- **Level** Sensitive
  - Action of the device is dependent on the **level** of the clock
  - Outputs can change anytime Clock = 1





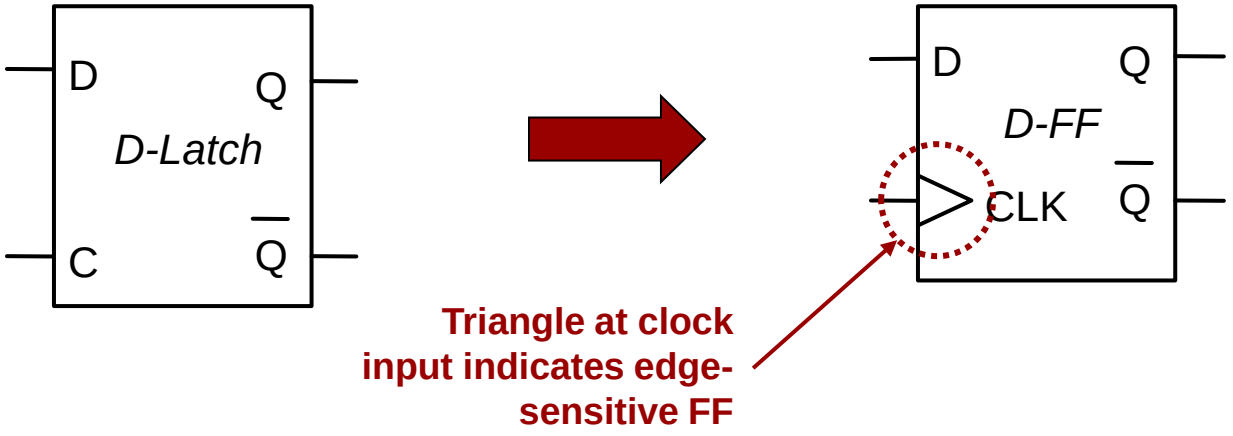
Flip-Flops

- Synchronous
- Clock Input
- **Edge** Sensitive
  - Outputs change only on the positive (negative) **edges**



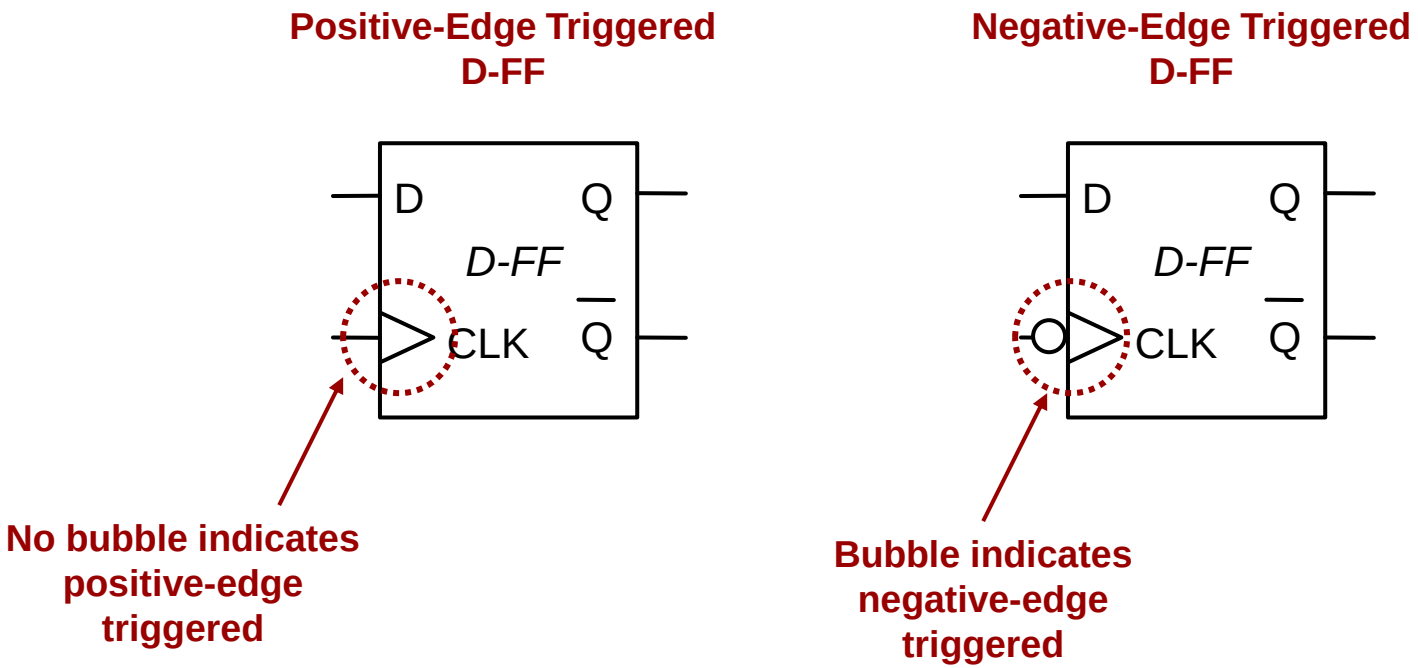
Flip-Flops

- Change D Latches to D Flip-Flops



Flip-Flops

- To indicate negative-edge triggered use a bubble in front of the clock input



Notation

- To show that Q remembers its value we can put it in the past tense:
  - $Q = Q_0$ (Current Value of Q = Old Value of Q)
- OR put it in the future tense
  - $Q^* = Q$ (Next Value of Q = Current Value of Q)

Indicates "next-value" of Q

C	D	Q	Q'
0	x	Q_0	Q_0'
1	0	0	1
1	1	1	0

Current Value = Old Value

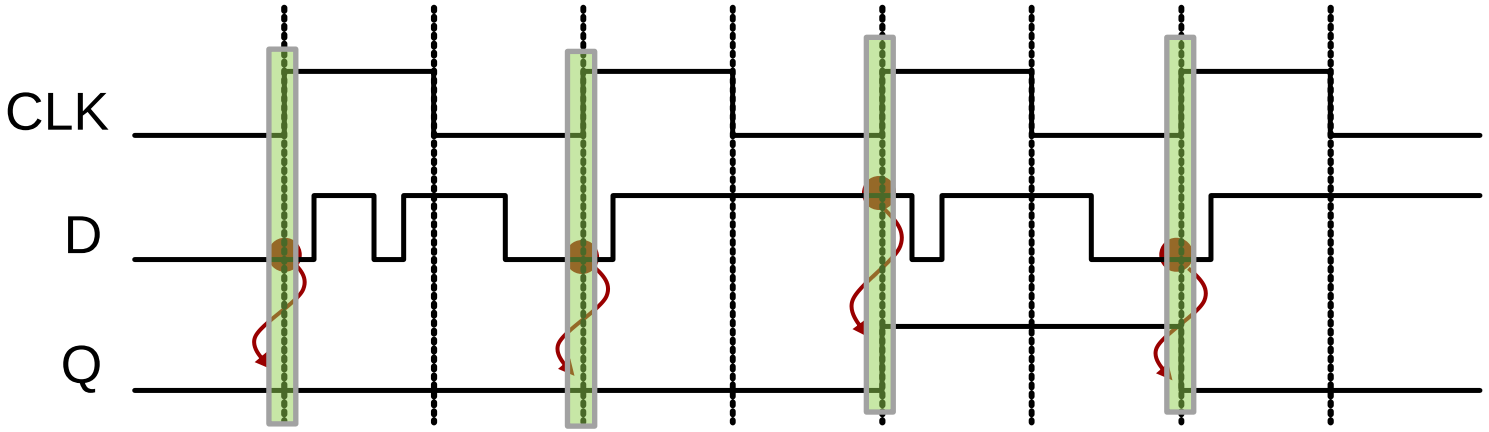
C	D	Q*	Q'*
0	x	Q	Q'
1	0	0	1
1	1	1	0

Next Value = Current Value

Positive-Edge Triggered D-FF

- Q looks at D only at the positive-edge

CLK	D	Q*	Q'*
0	x	Q	Q'
1	x	Q	Q'
↑	0	0	1
↑	1	1	0

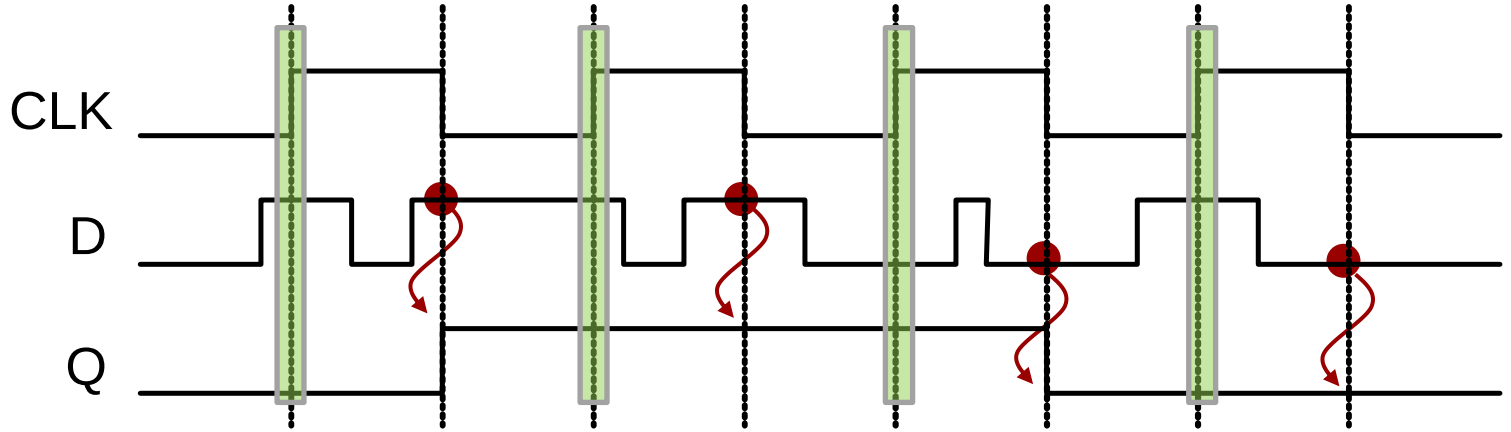


Q only samples D at the positive edges and then holds that value until the next edge

Negative-Edge Triggered D-FF

- Q looks at D only at the negative-edge

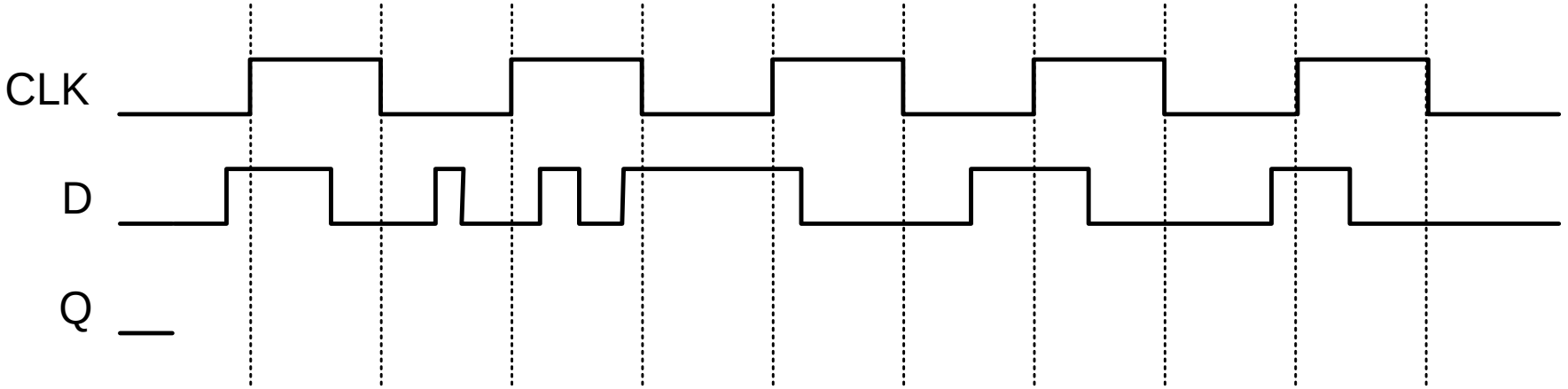
CLK	D	Q*	Q'*
0	x	Q	Q'
1	x	Q	Q'
↓	0	0	1
↓	1	1	0



Q only samples D at the negative edges and then holds that value until the next edge

D FF Example

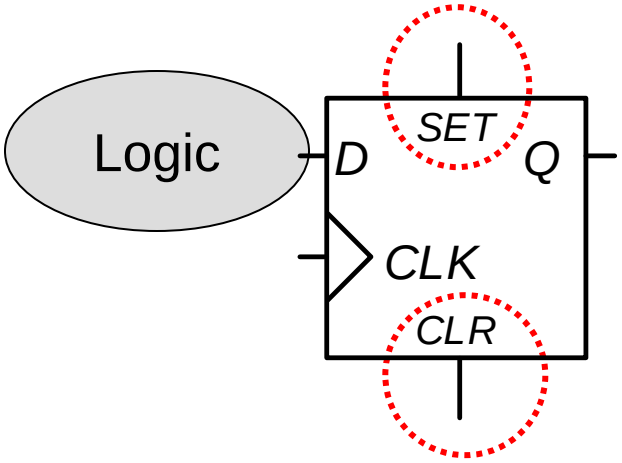
- Assume positive edge-triggered FF



INITIALIZING OUTPUTS

Initializing Outputs

- When the power is turned on, flip-flop outputs will be random values (0 or 1)
- We usually want to initialize the output to a known value (0 or 1)
- FF inputs are often connected to logic that will produce values after initialization (and, thus, may be hard to use for initialization)
- Two extra inputs are often included:
  - (PRE)SET (used if we want to initialize Q to 1)
  - CLEAR (used if we want to initialize Q to 0)



When CLEAR = on
Q*=0
When SET = on
Q*=1
When NEITHER are on
Normal FF operation

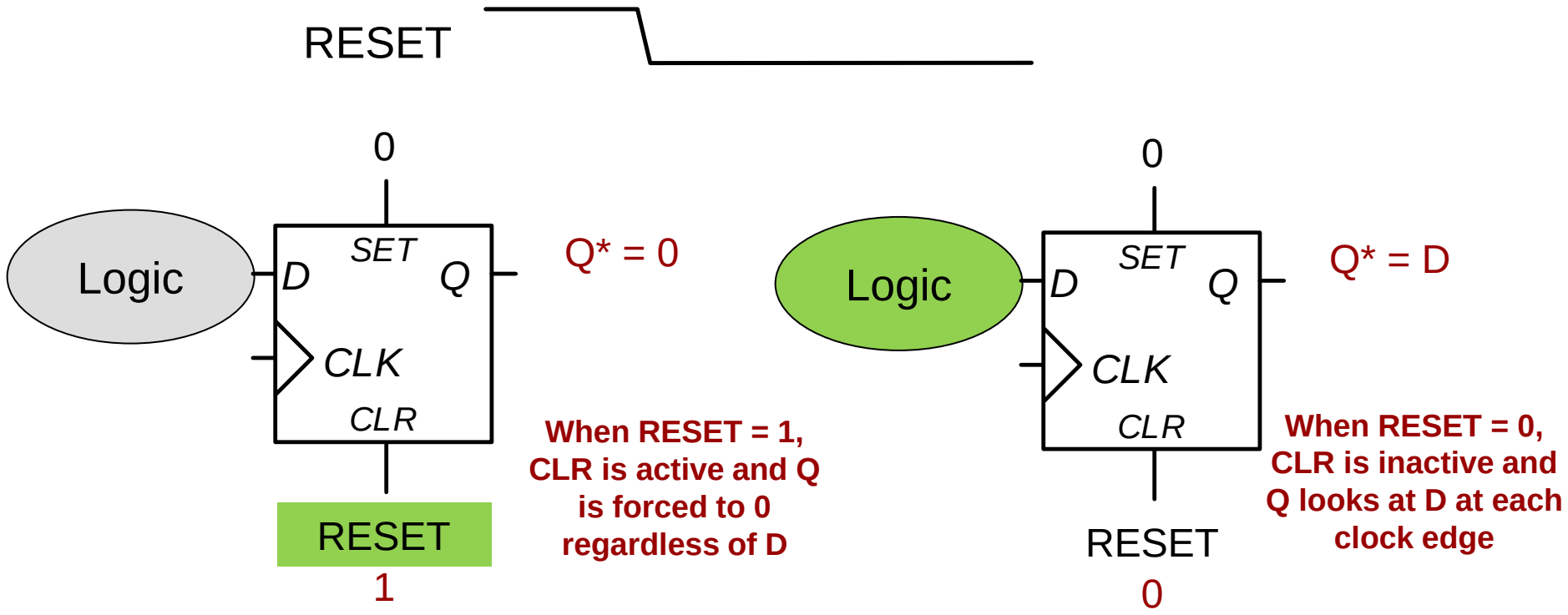
Note: CLR and SET have priority over normal FF inputs

Initializing Outputs

- To help us initialize our FF's use a RESET signal
  - Generally produced for us and given along with CLK
- It starts at *Active (1)* when power turns on and then goes to *Inactive (0)* for the rest of time
- When it's active, use it to initialize the FF's and then it will go inactive for the rest of time and the FF's will work based on their inputs

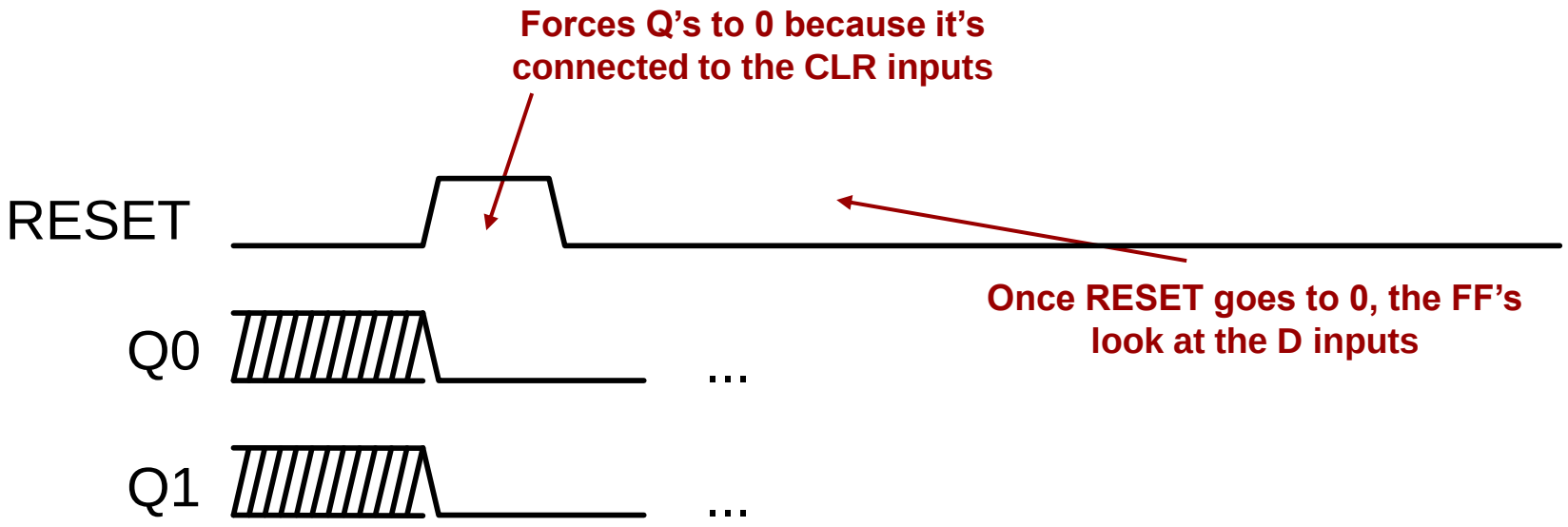
Initializing Outputs

- Suppose we want our FF to initialize to 0 when the power turns on
  - Connect RESET to the CLR input
  - Connect 0 (off) to the SET input



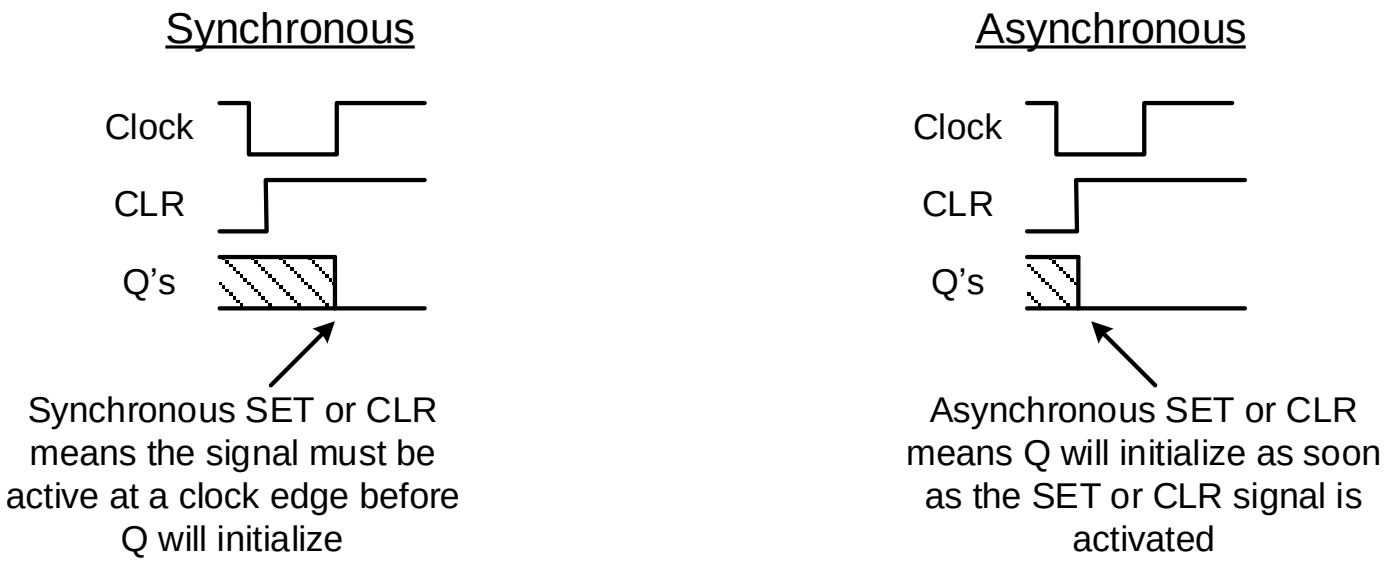
Implementing an Initial State

- When RESET is activated: Q's initialize to 0
- When RESET is deactivated: Q's look at the D inputs



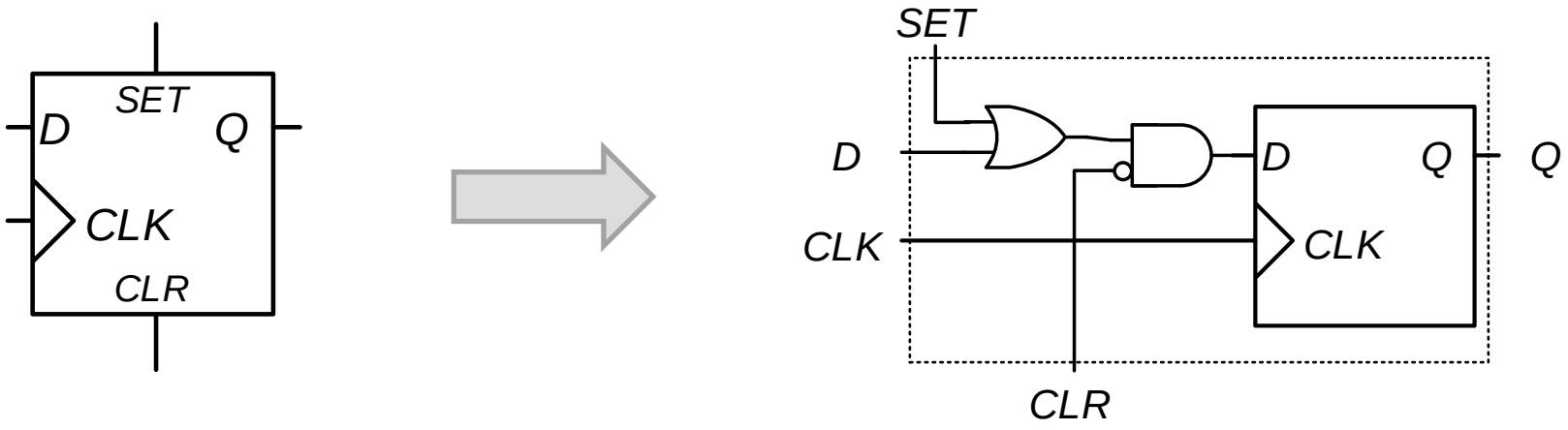
Synchronous vs. Asynchronous

- The new preset and clear inputs can be built to be ***synchronous*** or ***asynchronous***
- These terms refer to when the initialization takes place
  - Asynchronous…initialize as soon as signal is activated
  - Synchronous…initialize at clock edge



Implementing SET and CLEAR

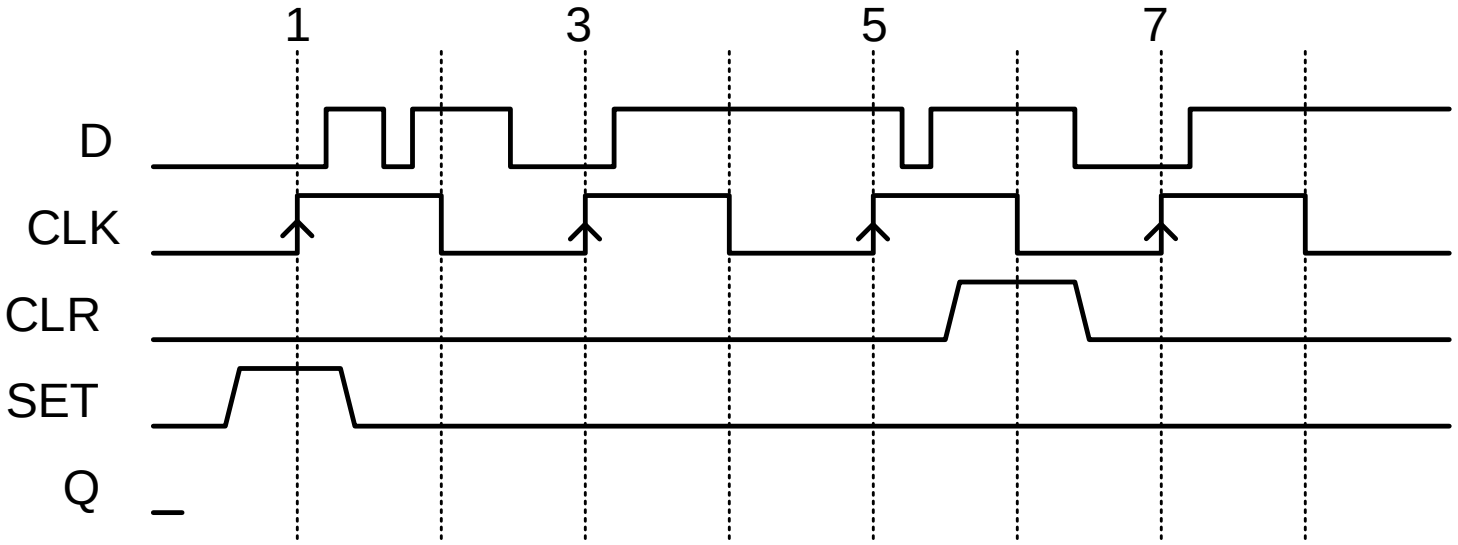
- Synchronous set and clear can be implemented through adding additional gates in front of the input
- Asynchronous set and clear are a bit more complicated due to their asynchronous nature and are not covered in this class



Implementation of SYNCHRONOUS SET and CLEAR

Set / Clear Example

- Complete the waveform for a D-FF with ***asynchronous*** SET and CLR



- What would change if these were ***synchronous*** SET/CLR
  - SET pulse at time 1 would not take effect until the clock edge
  - CLR pulse at time 6 would have no effect

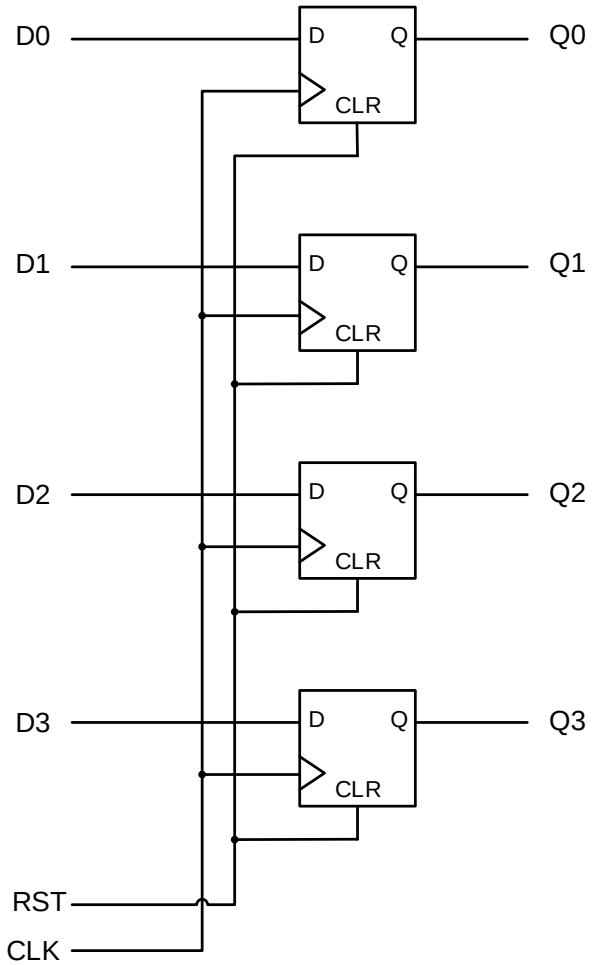
Groups of flip-flops

REGISTERS AND REGISTERS WITH ENABLES

Registers

- Registers are simply collections of flip-flops (**n-bit register = n flip flops**) that have a common clock and reset signal
- **Registers in HW** are analogous to **variables in SW** (used to store a value)
- Can use an asynchronous or synchronous "reset" to force the flip-flops to 0's
  - Which is shown in the table below? Synch.

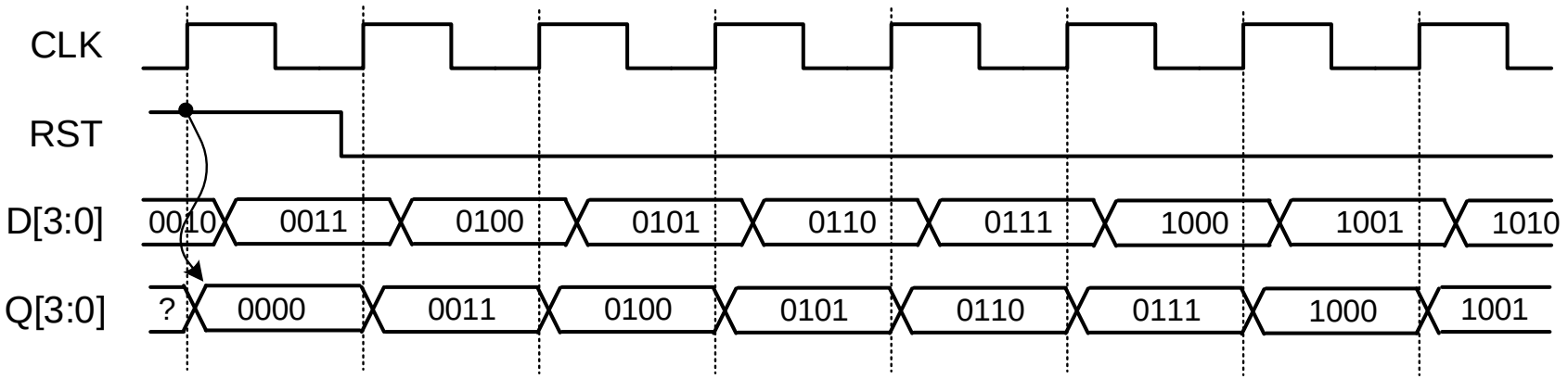
CLK	RST	D_i	Q_i*
1,0	X	X	Q_i
⇈	1	X	0
⇈	0	0	0
⇈	0	1	1



4-bit Register

Register Operation (and a Problem)

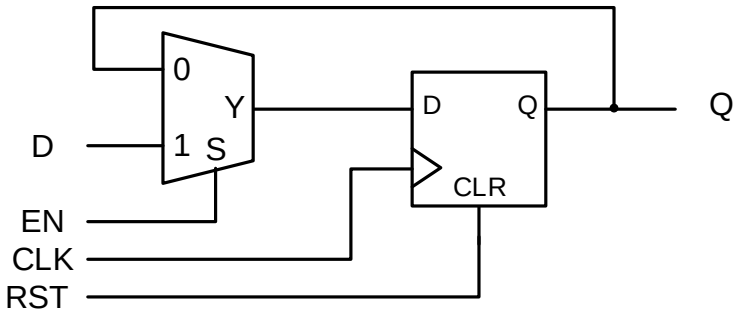
- The value on the D input is sampled at the clock edge and passed to the Q output and holds until the next clock edge
- Feature/Problem: Register saves data on EVERY edge
  - Often we want the ability to save on one edge and then keep that value for many more cycles



4-bit Register – On clock edge, D is passed to Q

Solution

- Registers (D-FF's) will sample the D bit every clock edge and pass it to Q
- Sometimes we may want to hold the value of Q and ignore D even at a clock edge
- We can add an enable input and some logic in front of the D-FF to accomplish this

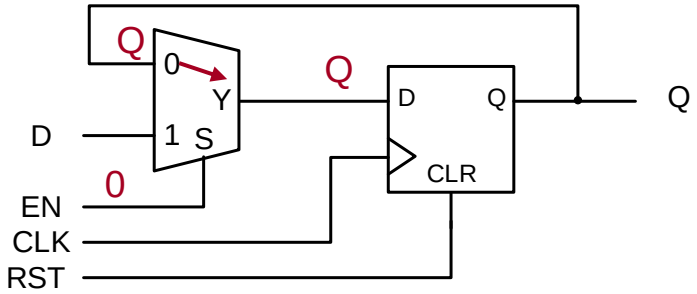


FF with Data Enable
(Always clocks, but selectively chooses old value, Q, or new value D)

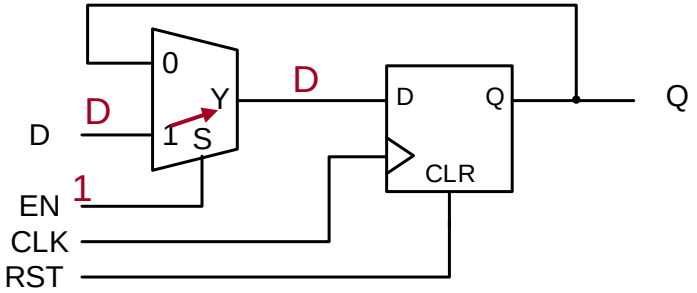
CLK	RST	EN	D_i	Q_i*
0,1	X	X	X	Q_i
⇈	1	X	X	0
⇈	0	0	X	Q_i
⇈	0	1	0	0
⇈	0	1	1	1

Registers w/ Enables

- When EN=0, Q value is passed back to the input and thus Q will maintain its value at the next clock edge
- When EN=1, D value is passed to the input and thus Q can change at the edge based on D



When EN=0, Q is recycled back to the input

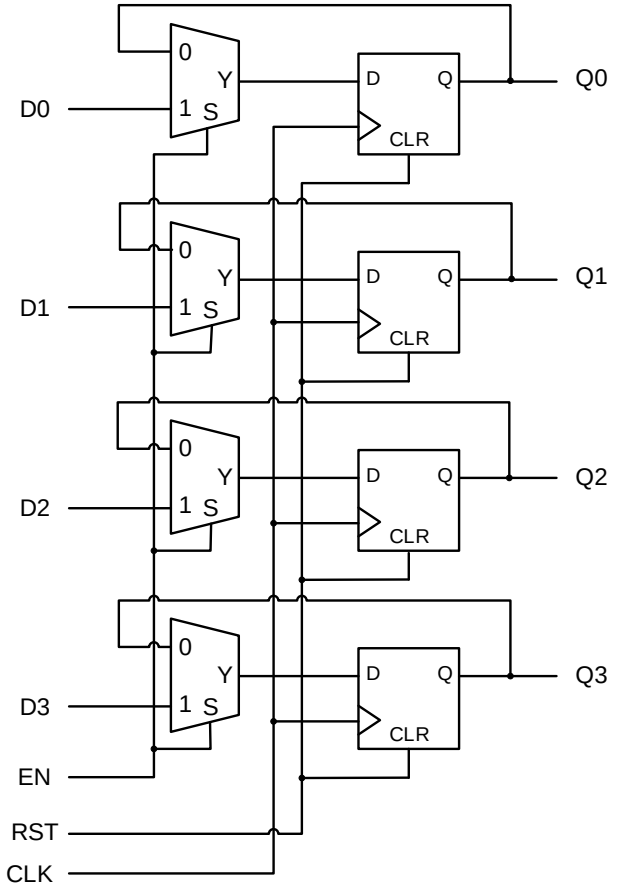


When EN=1, D input is passed to FF input

4-bit Register w/ Data (Load) Enable

- Registers (D-FF's) will sample the D bit every clock edge and pass it to Q
- Sometimes we may want to hold the value of Q and ignore D even at a clock edge
- We can add an enable input and some logic in front of the D-FF to accomplish this

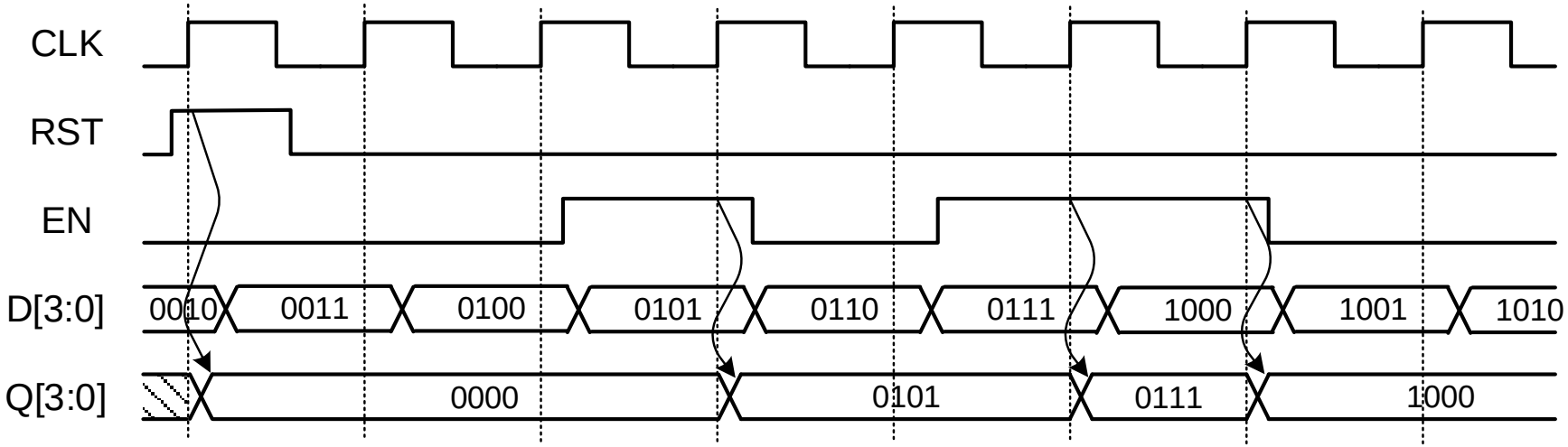
CLK	RST	EN	D_i	Q_i*
0,1	X	X	X	Q_i
⇈	1	X	X	0
⇈	0	0	X	Q_i
⇈	0	1	0	0
⇈	0	1	1	1



4-bit register with 4-bit wide 2-to-1 mux in front of the D inputs

Registers w/ Enables

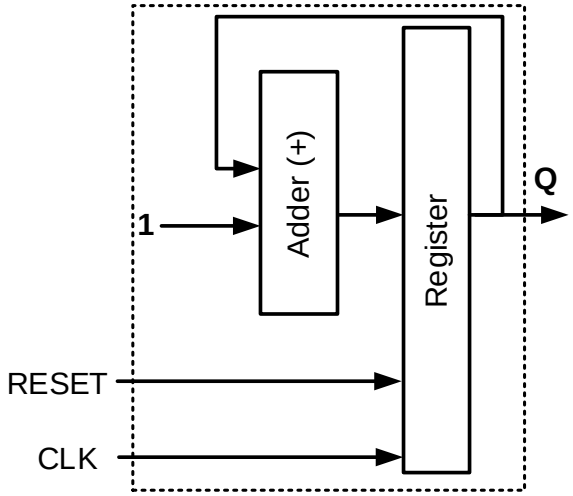
- The D value is sampled at the clock edge only if the enable is active
- Otherwise the current Q value is maintained



COUNTERS

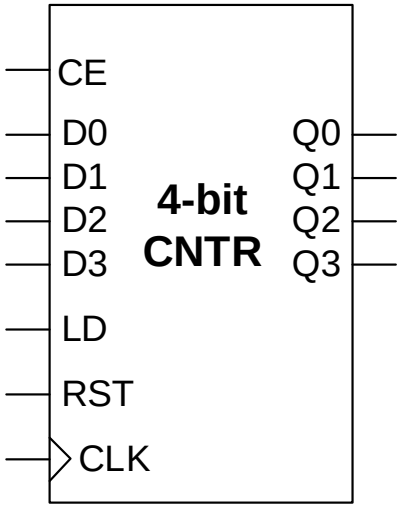
Counters

- Count (Add 1 to Q) at each clock edge
  - Up Counter: Q* = Q + 1
  - Can also build a down counter as well (Q* = Q – 1)
- Standard counter components include other features
  - Resets: Reset count to 0
  - Count Enables (CE): Will not count at edge if CE=0
  - Data Load Inputs: Can initialize count to a value D (i.e. Q* = D rather than Q+1)



Sample 4-bit Counter

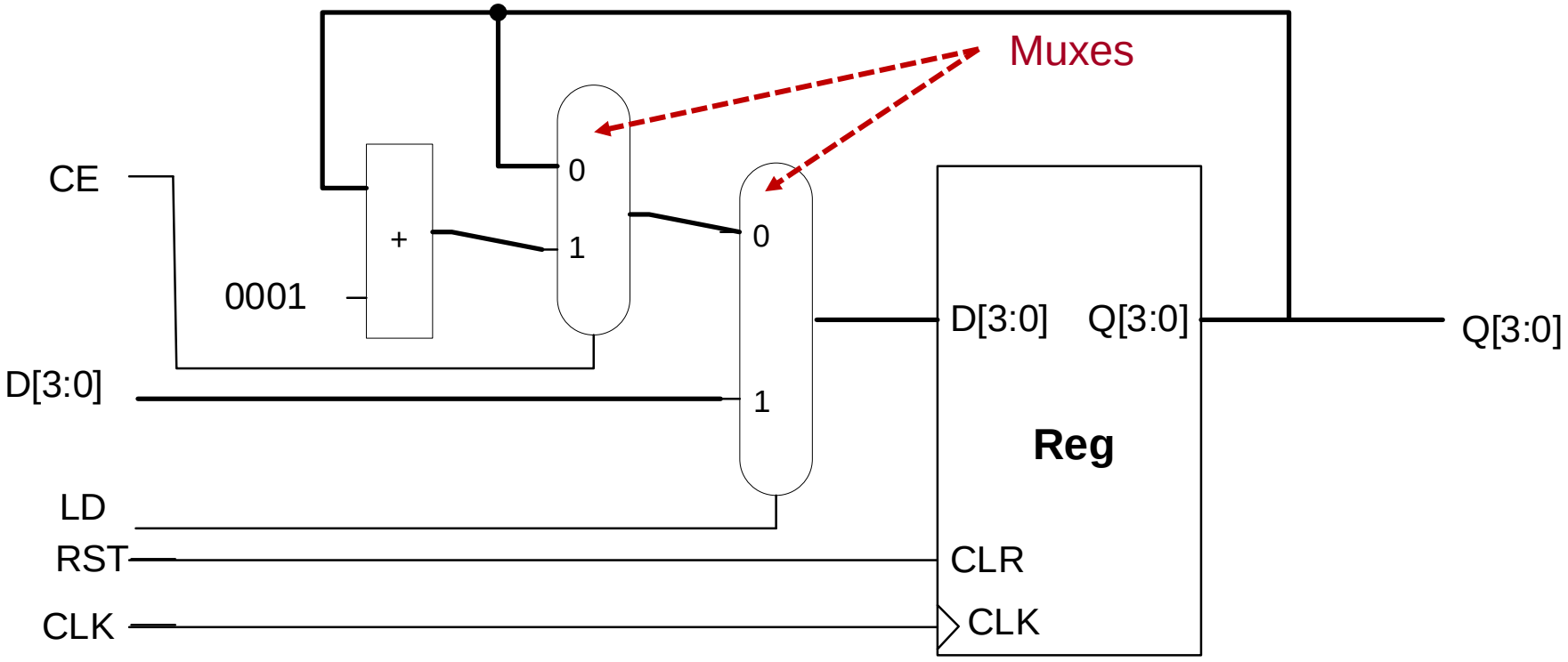
- 4-bit Up Counter
  - RST: synchronous reset input
  - LD and D_i inputs: loads Q with D when LD is active
  - CE: Count Enable
    - Must be active for the counter to count up



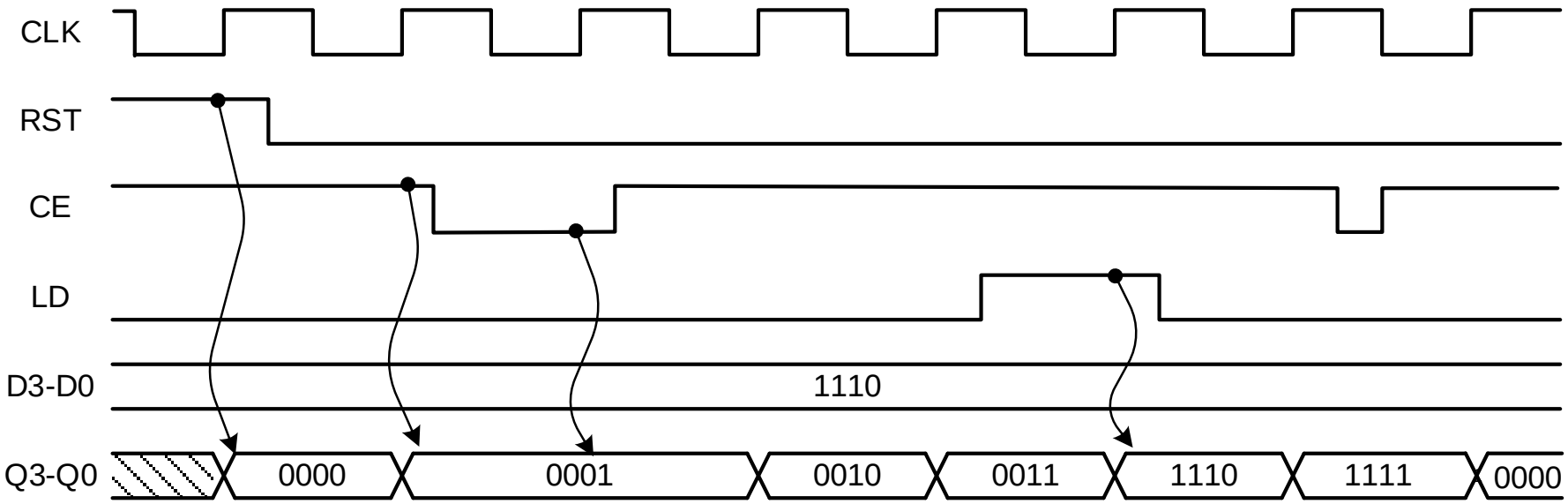
CLK	RST	LD	CE	Q*
0,1	X	X	X	Q
⇈	1	X	X	0
⇈	0	1	X	D[3:0]
⇈	0	0	1	Q+1
⇈	0	0	0	Q

Counter Design

- Sketch the design of the 4-bit counter presented on the previous slides



Counters



RST=active at clock edge, thus Q=0

Q*=Q+1

CE=0, thus Q holds

Q*=Q+1

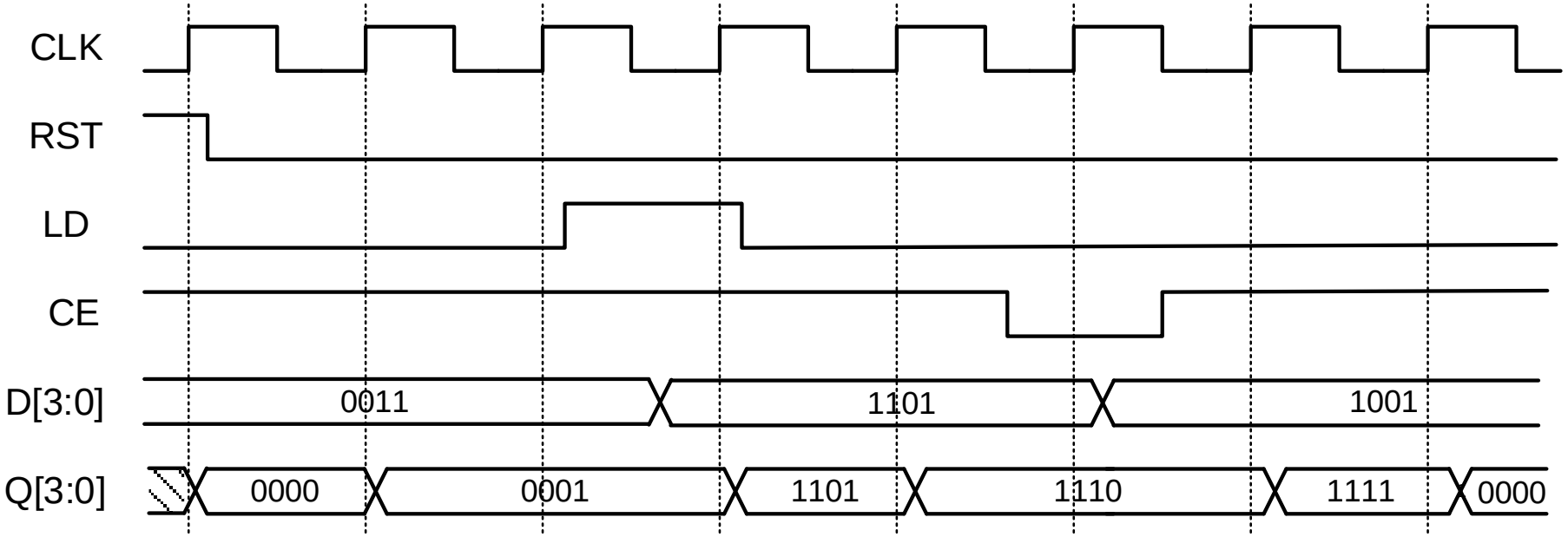
Q*=Q+1

LD = active, thus Q=P

Q*=Q+1

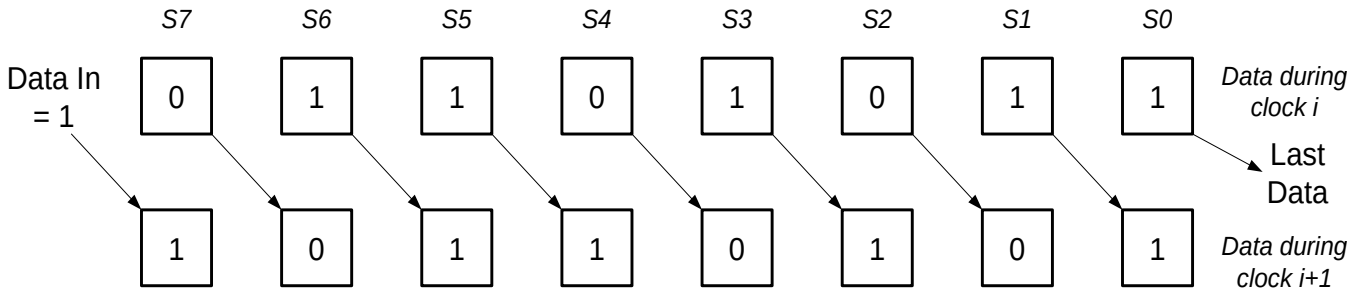
Q*=Q+1

Counter Exercise



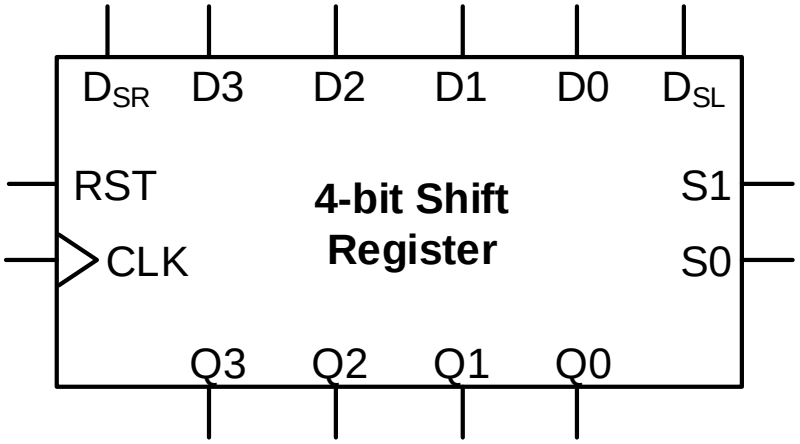
Shift Register

- A shift register is a device that acts as a ‘queue’ or ‘FIFO’ (First-in, First-Out).
- It can store n bits and each bit moves one step forward each clock cycle
  - One bit comes in the overall input per clock
  - One bit ‘falls out’ the output per clock



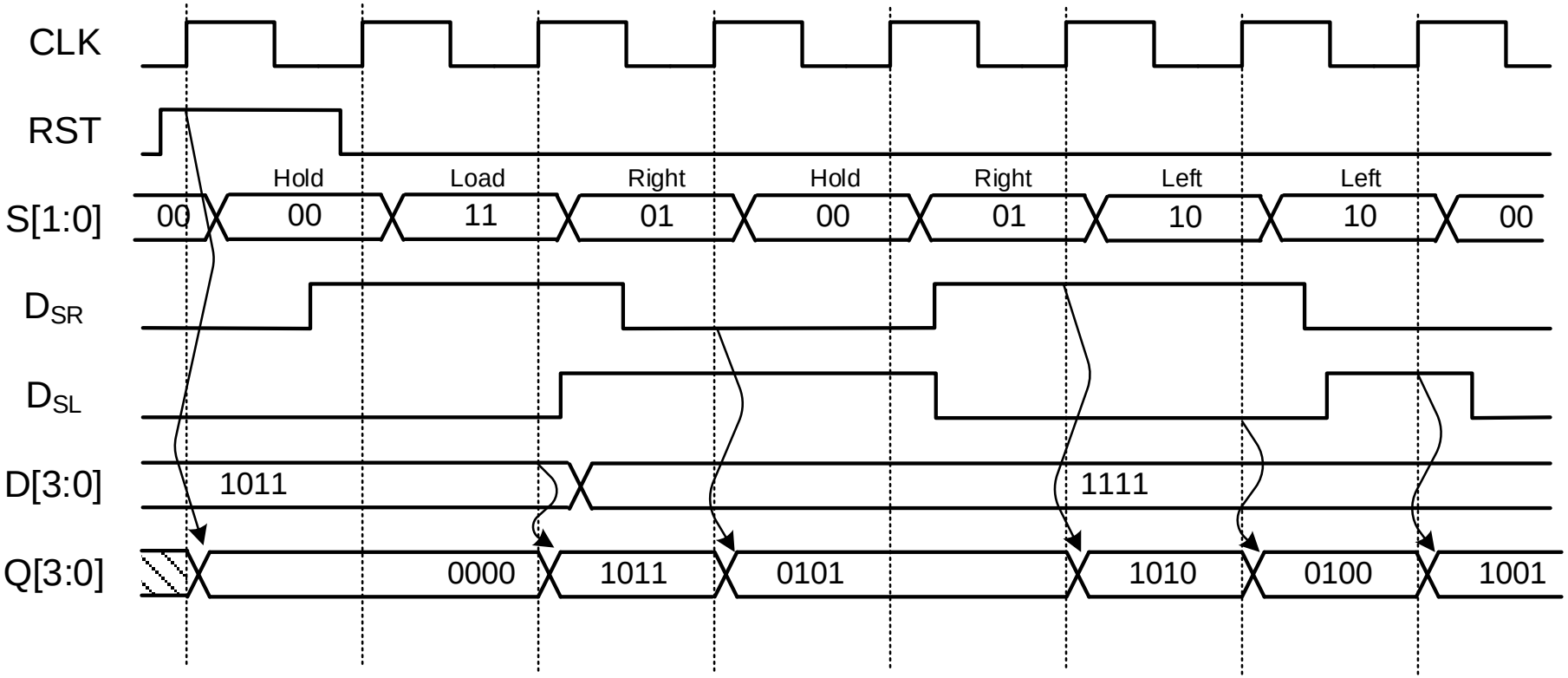
Sample Shift Register

- Shift registers come in many flavors, we'll just look at one example
- 4-bit Bi-directional Shift Register
  - RST: synchronous reset
  - S[1:0]: Hold, Right Shift, Left Shift, or Load
  - DSL and DSR
    - Data to shift in from left or right



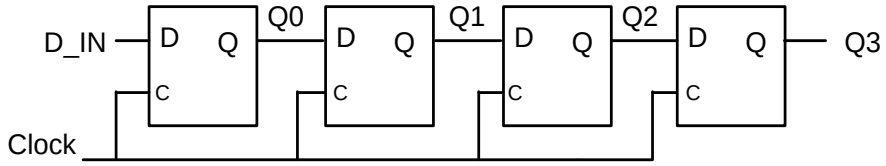
CLK	RST	S1	S0	Q*[3:0]	(case)
0,1	X	X	X	Q[3:0]	
⇈	1	X	X	0000	Reset
⇈	0	0	0	Q[3:0]	Hold
⇈	0	0	1	D_{SR},Q[3:1]	Right
⇈	0	1	0	Q[2:0],D_{SL}	Left
⇈	0	1	1	D[0:3]	Load

Shift Registers

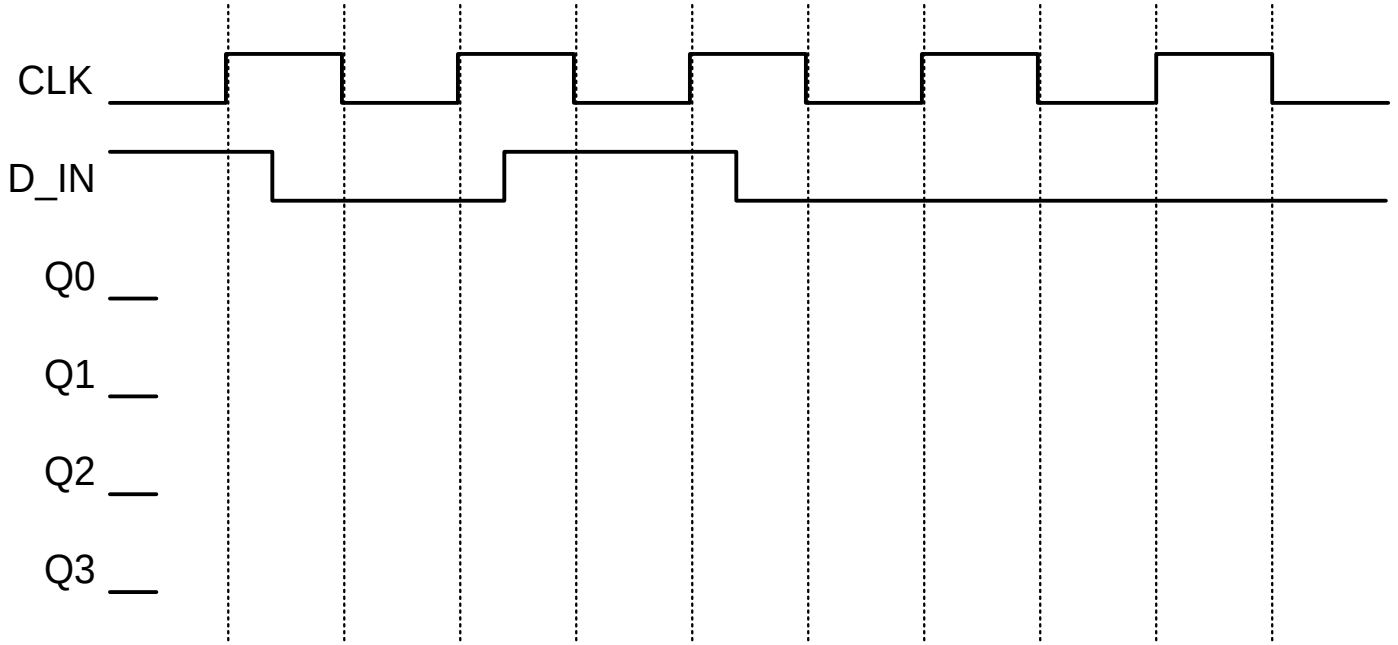


Shift Register

- Can we build a shift register from latches?

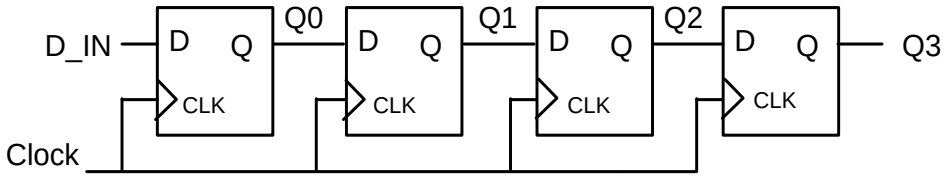


Shift Register w/ Latches

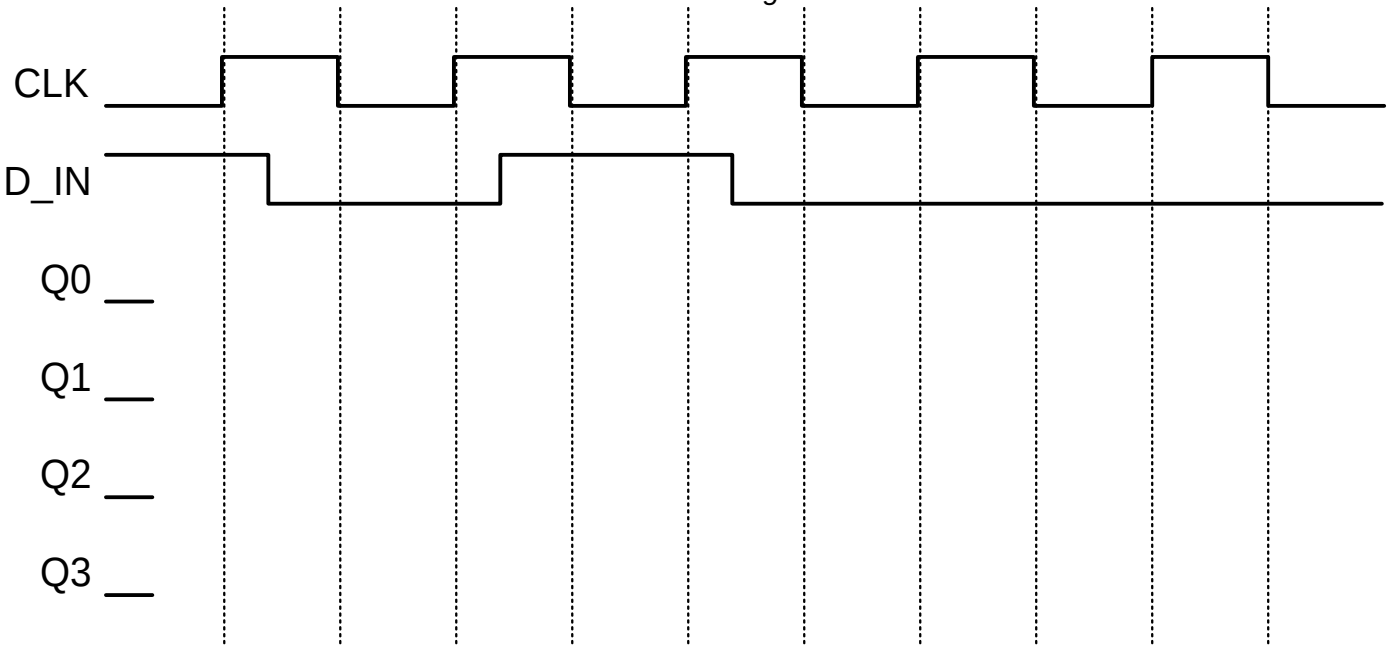


Shift Register

- Can we build a shift register from flip-flops?

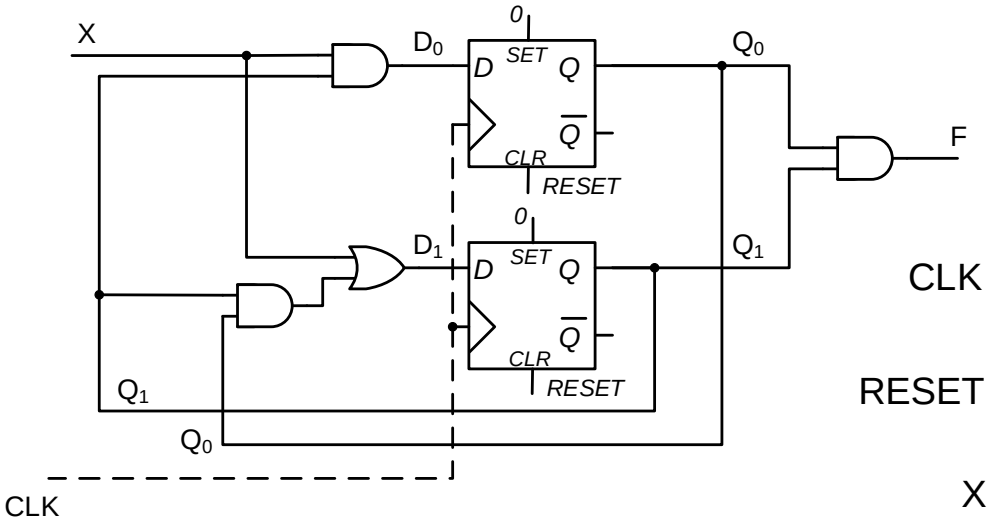


Shift Register w/ FF's

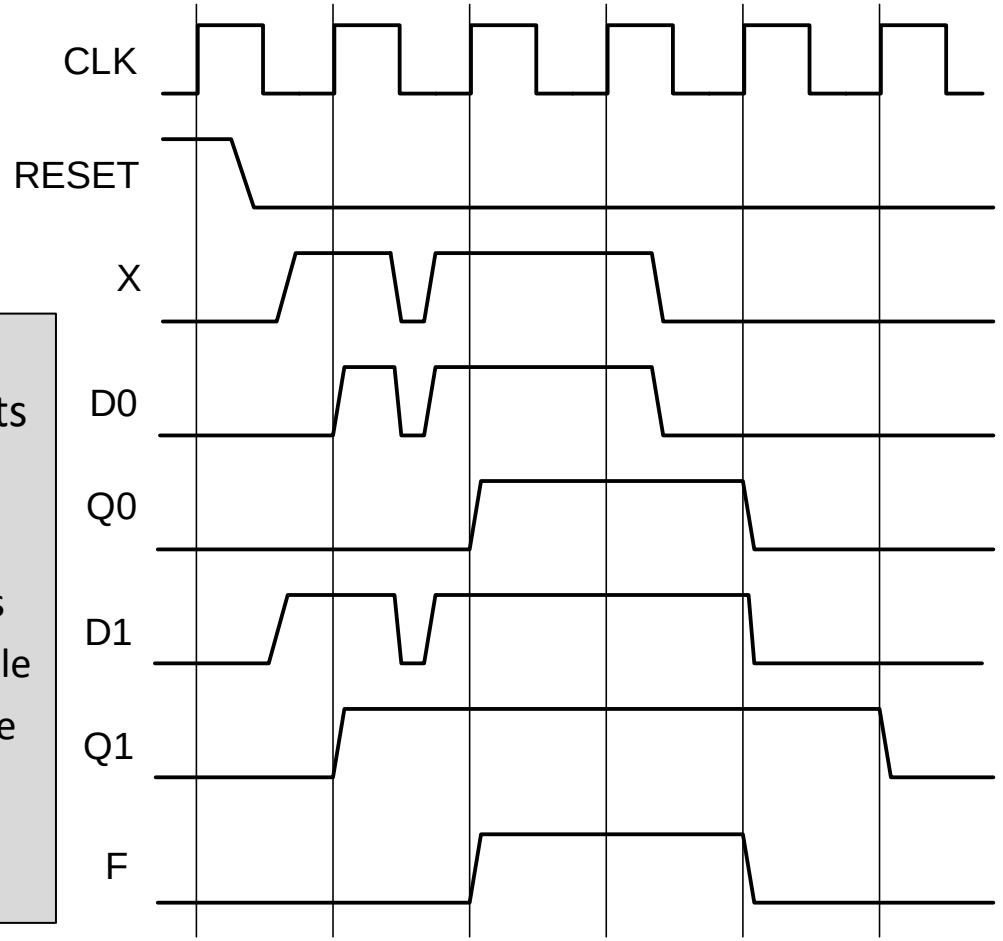


When we want to ensure an output updates only ONCE per clock, we need to use flip-flops (not latches or bistables)!

Exercise 1



- D0 = X•Q1
- D1 = X + Q1•Q0
- F = Q1•Q0



- Q outputs change on an edge while D0, D1, and F can change anytime the inputs change (since they are combinational)
- Process for solving:
  - **Step 0**: Write Boolean eqns for D inputs
  - **Step 1**: Hold Q values steady for full cycle
  - **Step 2**: Use Q values to solve for D value
  - **Step 3**: Use D value at the next edge to determine next Q values
  - Go back to step 1

Exercise 2

- Complete the waveform for the ouput of the 3 registers: X, Y, Z