Unit 13

Sequential Logic Constructs

Learning Outcomes

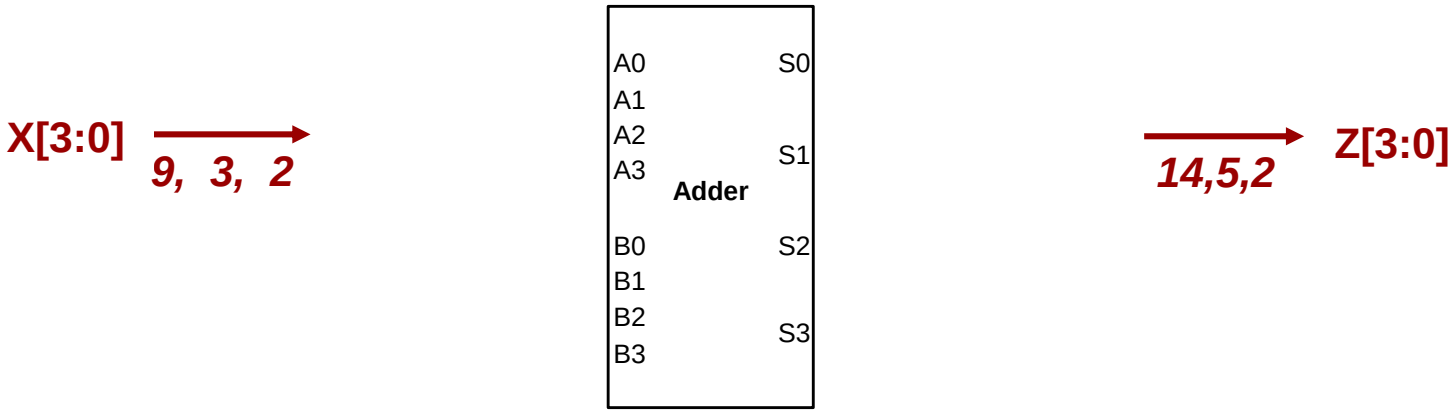
- I understand the difference between level-sensitive and edge-sensitive
- I understand how to create an edge-triggered FF from 2 latches

How sequential building blocks work

LATCHES AND FLIP-FLOPS

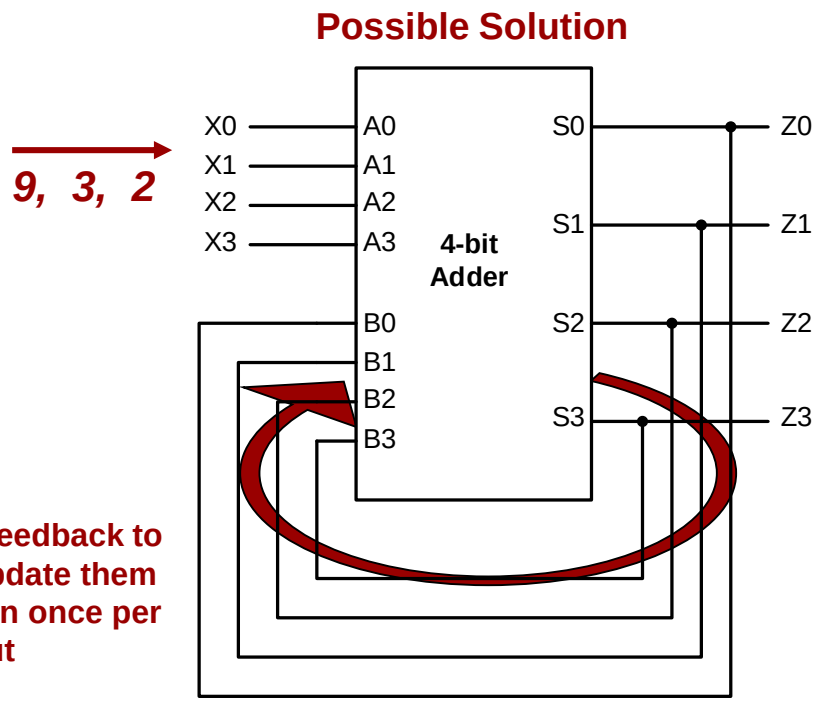
Sequential Logic

- Suppose we have a sequence of input numbers on X[3:0] that are entered over time that we want to sum up
- Possible solution: Route the outputs back to the inputs so we can add the current sum to the input X



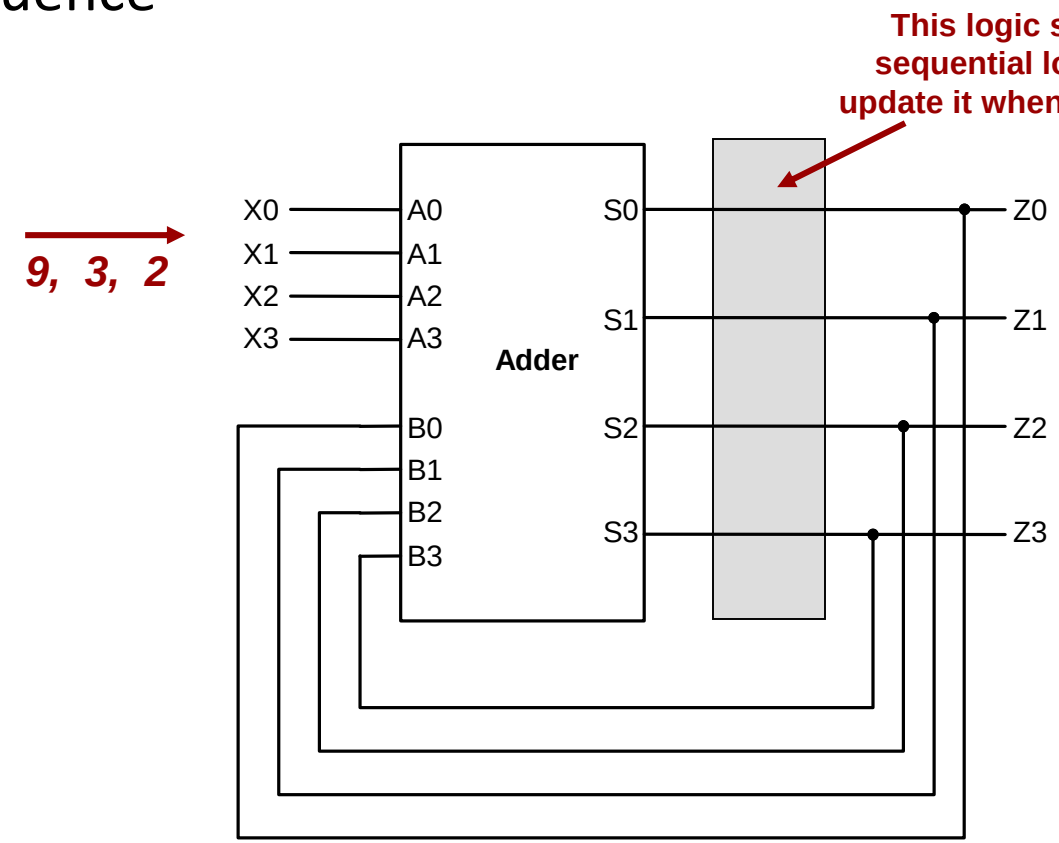
Sequential Logic

- Suppose we have a sequence of input numbers on X[3:0] that are entered over time that we want to sum up
- Possible solution: Route the outputs back to the inputs so we can add the current sum to the input X
- Problem 1: No way to initialize sum
- Problem 2: Outputs can race around to inputs and be added more than once per input number



Sequential Logic

- Add logic at outputs to just capture and remember the new sum until we're ready to input the next number in the sequence

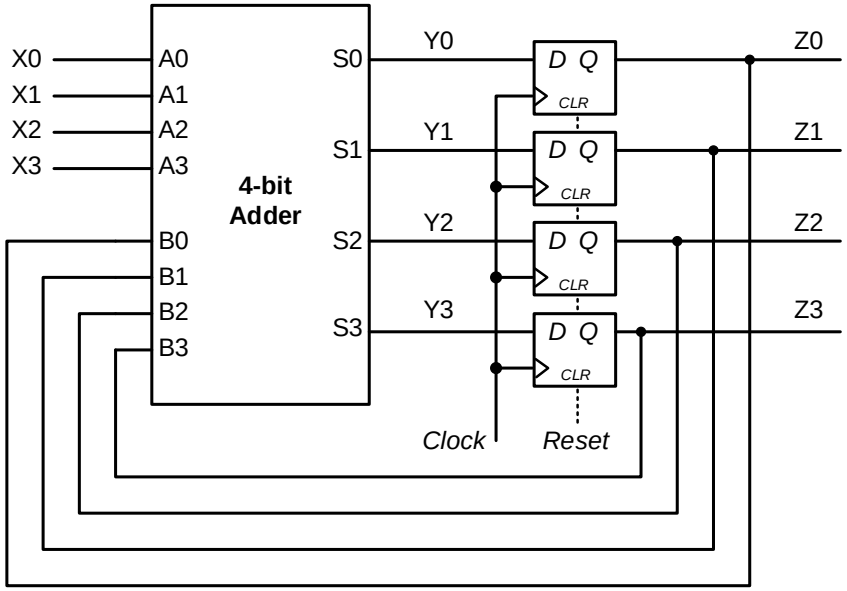


The data can still loop around and add up again (2+2=4) but if we just hold our output = 2 then the feedback loop will be broken

We remember initial sum of 2 until input 3 arrives at which point we'd capture & remember the sum 5.

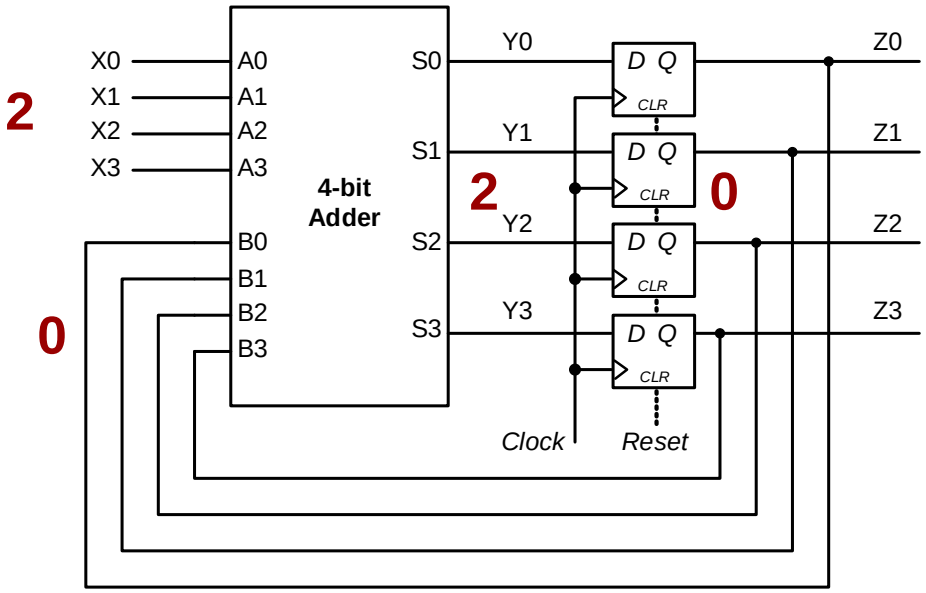
Sequence Adder

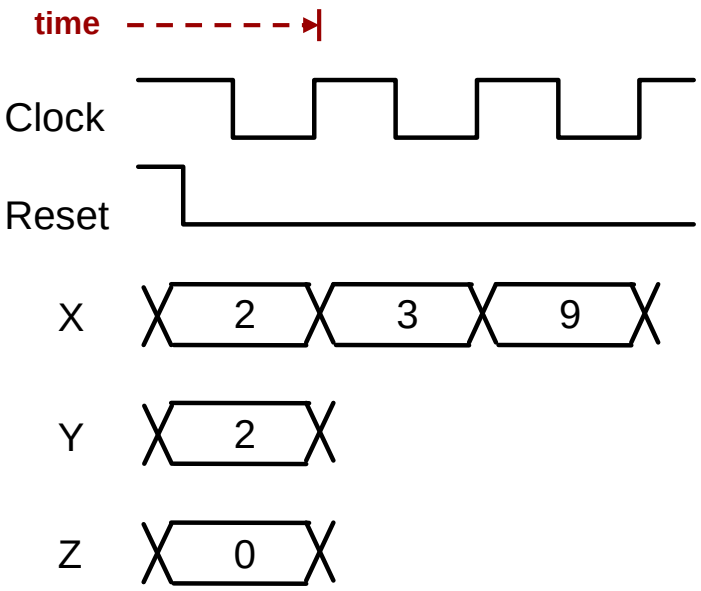
- If X changes once per cycle then Z should also change once per cycle
- That is why we will use a register (flip-flops) to ensure the outputs can only update once per cycle



Sequence Adder

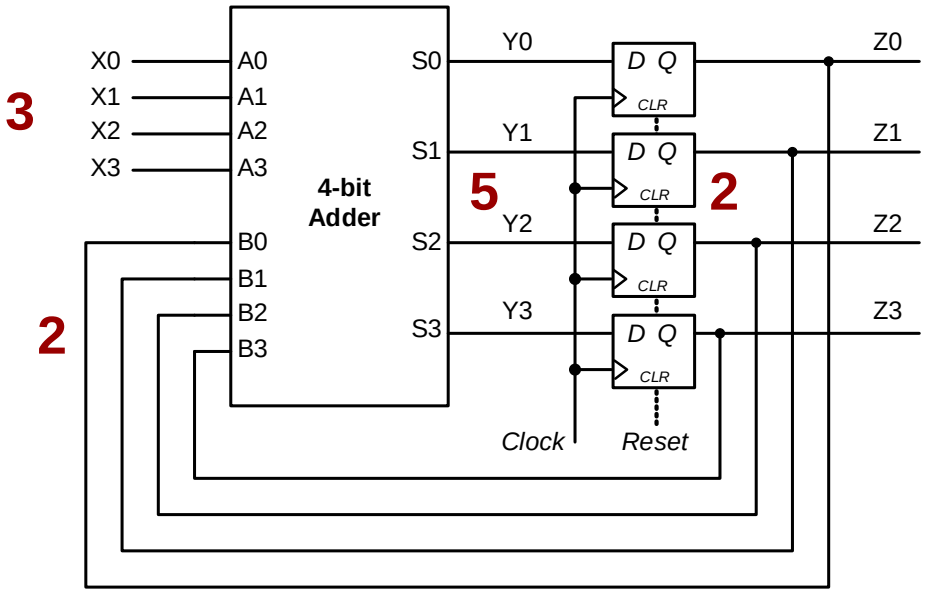
- The 0 on Clear will cause Z to be initialized to 0, but then Z can't change until the next positive edge
- That means we will just keep adding 0 + 2 = 2

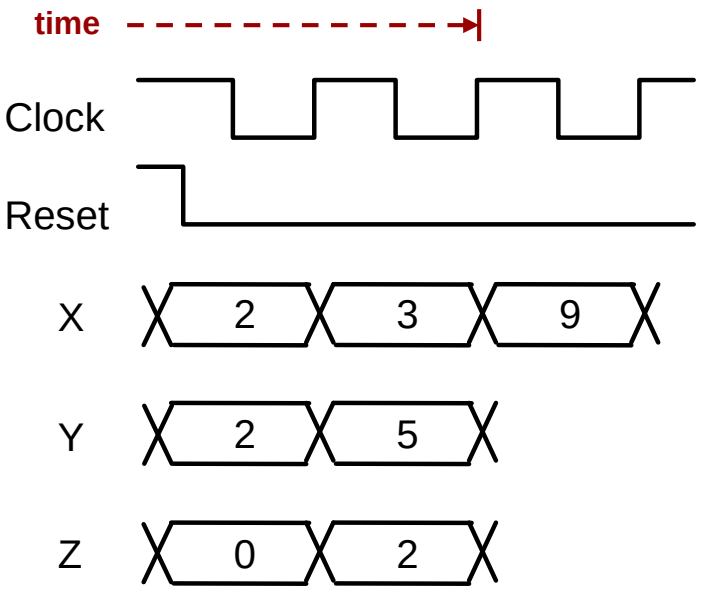




Sequence Adder

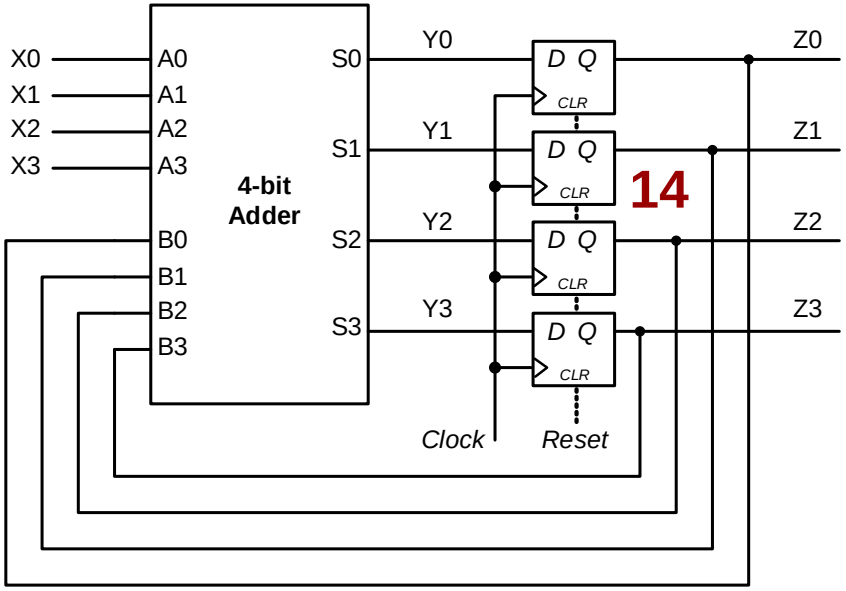
- At the edge the flip-flops will sample the D inputs and then remember 2 until the next positive edge
- That means we will just keep adding 3 + 2 = 5

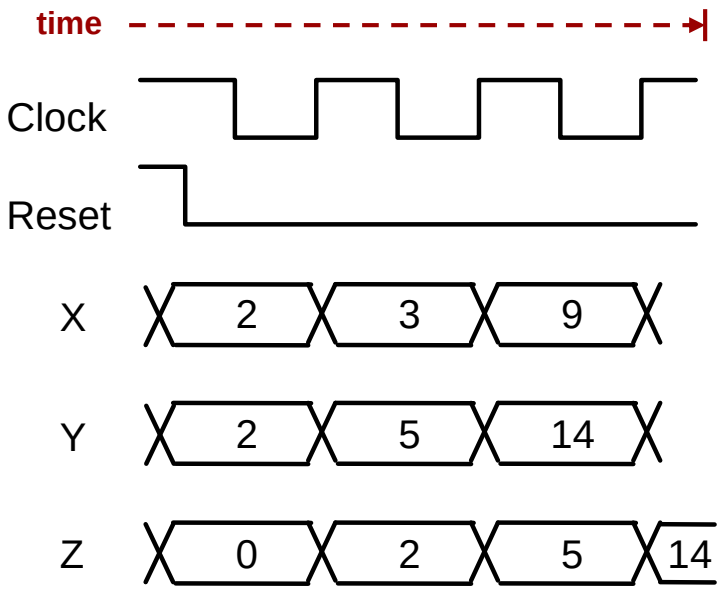




Sequence Adder

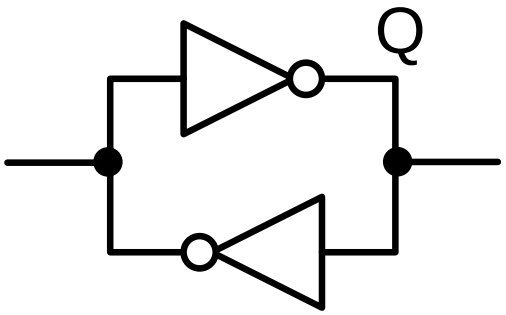
- Finally, at the positive edge the flip-flops will sample the D inputs and then remember 14



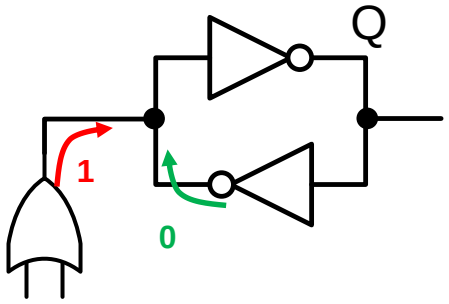


Sequential Logic

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

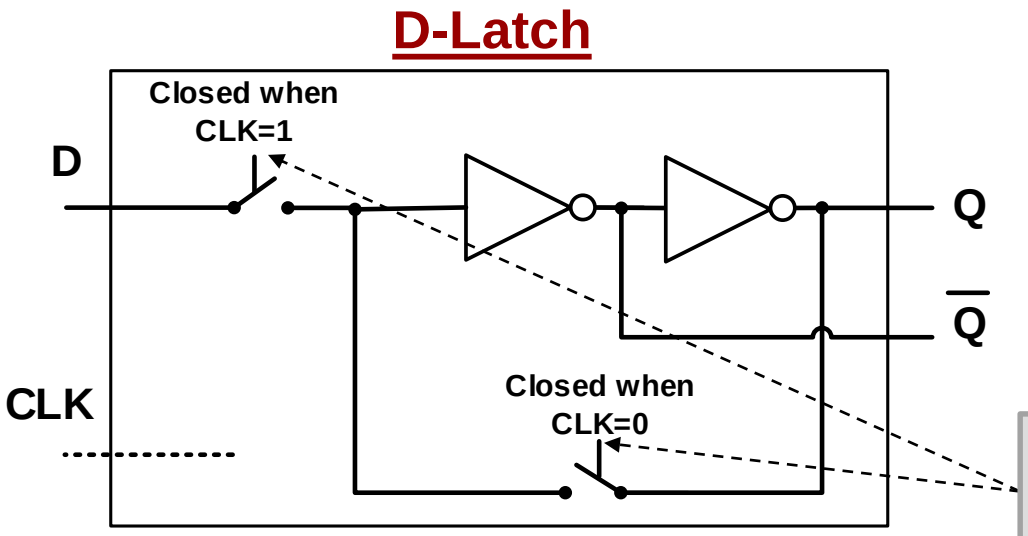


- But how do you change the input?
  - A signal should only have one driver



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)

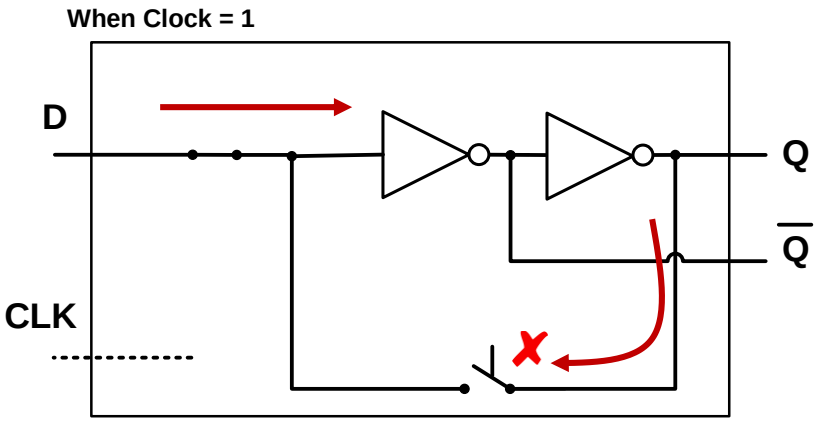


Transparent & Hold Mode of D-Latches

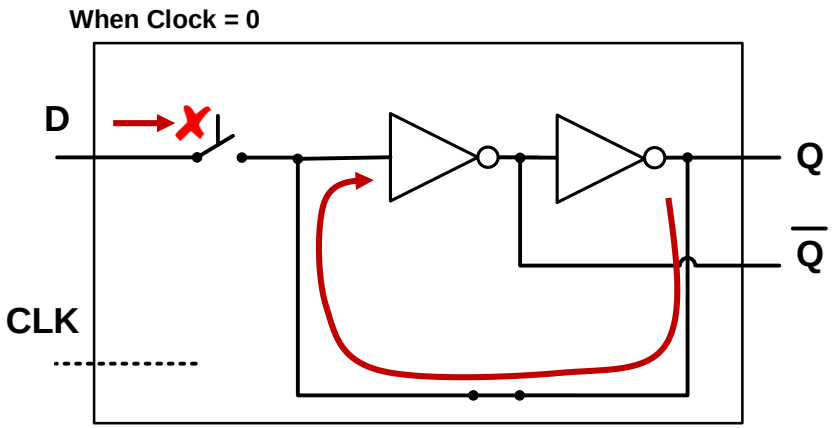
- The D-Latch operates in either transparent or hold mode based on the clock value

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

Function Table
Description of D-Latch



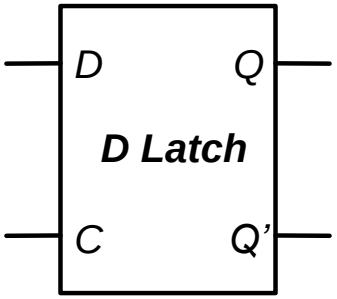
Transparent Mode
(Q=D when CLK=1)



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

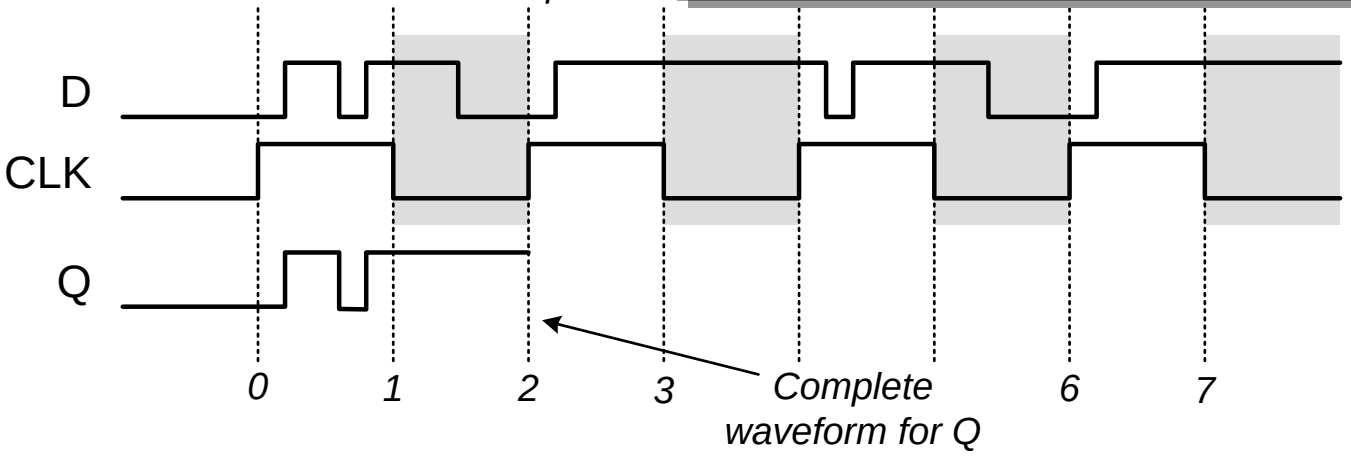
Hold Mode

Transparent Mode

D-LATCH
7475

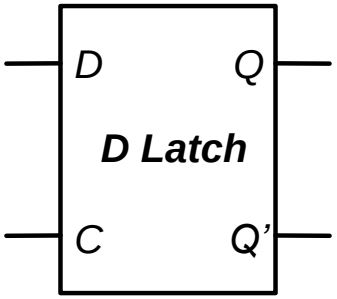
As clock is LOW, don't look at the D input

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

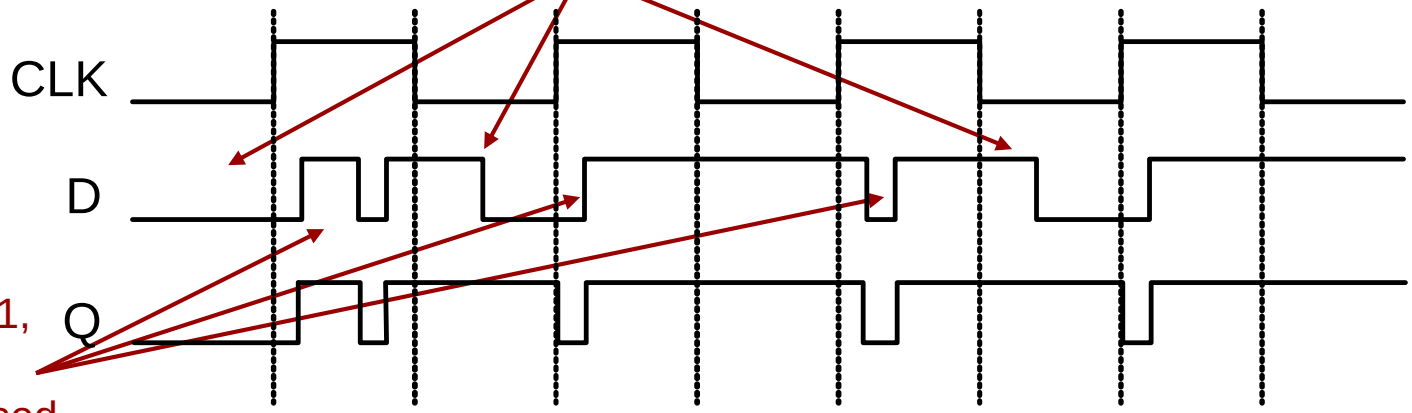


C	D	Q	Q'
0	x	Q_0	Q_0'
1	0	0	1
1	1	1	0

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

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

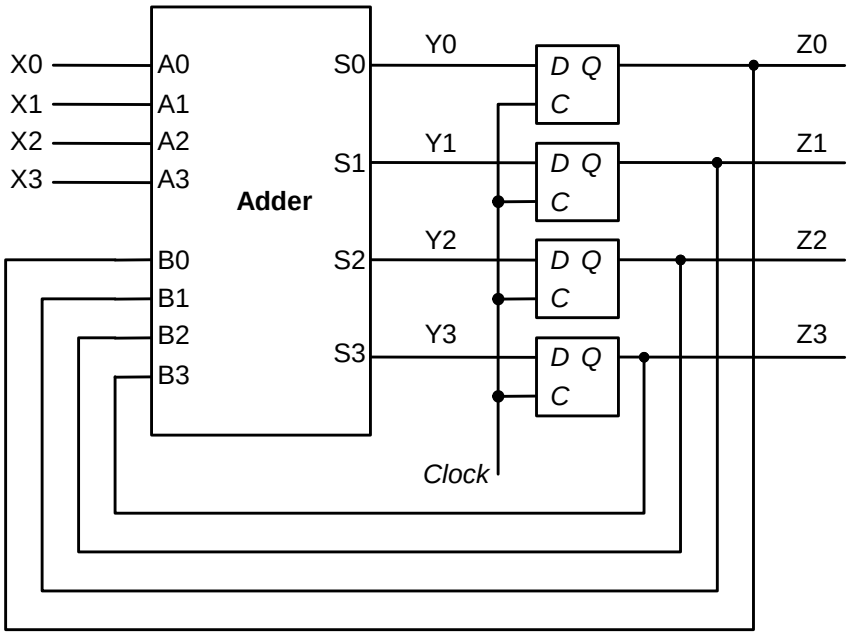
Current Value = Old Value

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

Next Value = Current Value

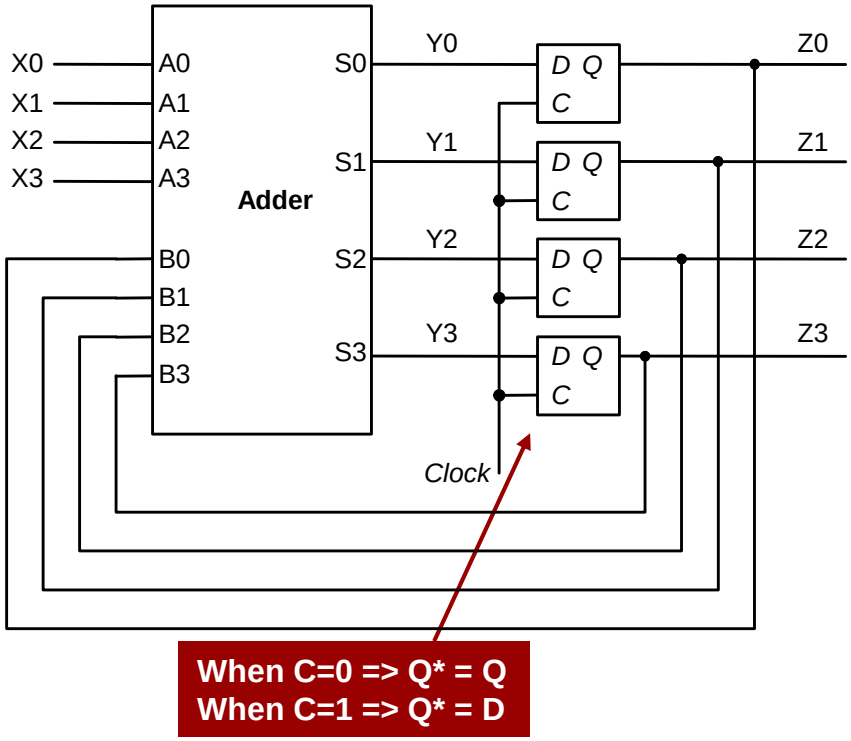
Adding a Sequence of Numbers

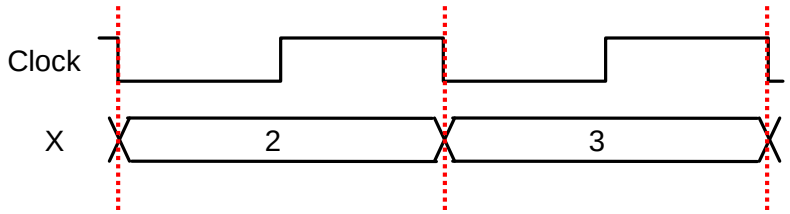
- What if we put D-Latches at the outputs



Adding a Sequence of Numbers

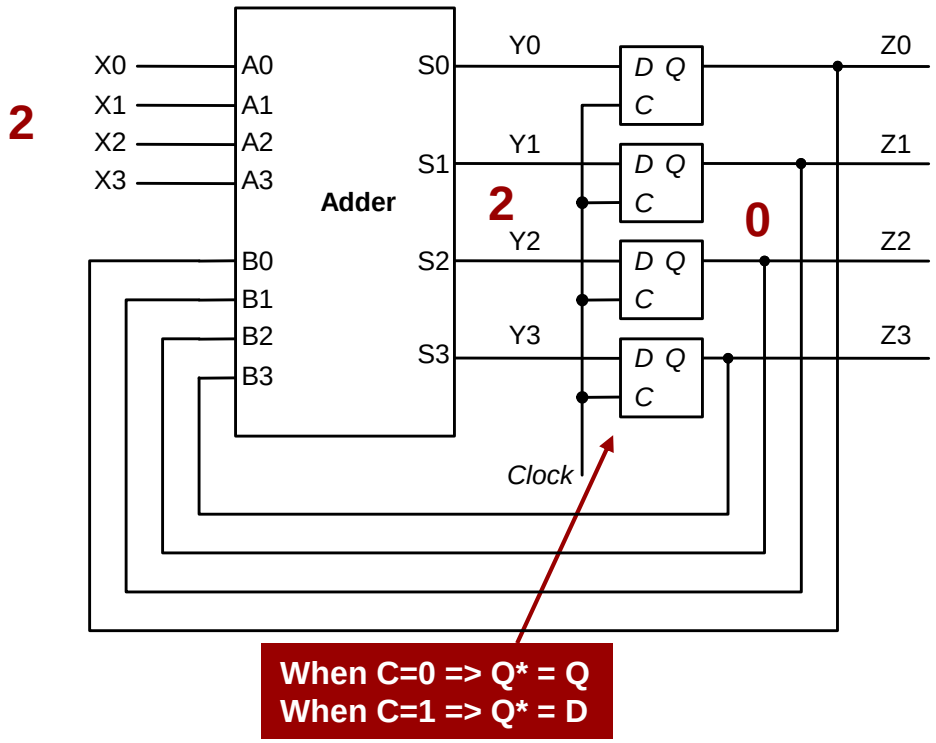
- We'll change X on every clock period

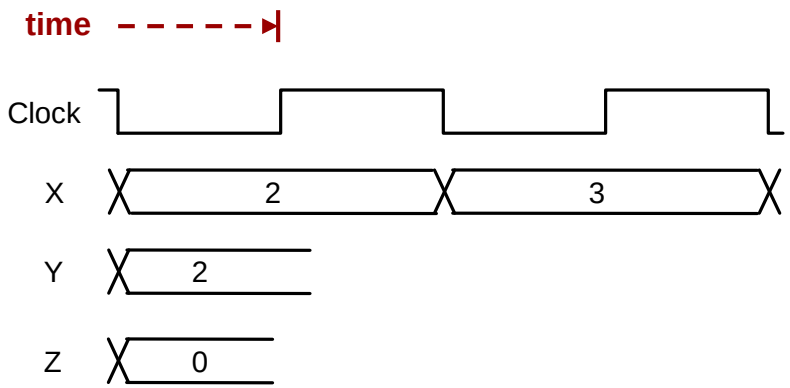




Adding a Sequence of Numbers

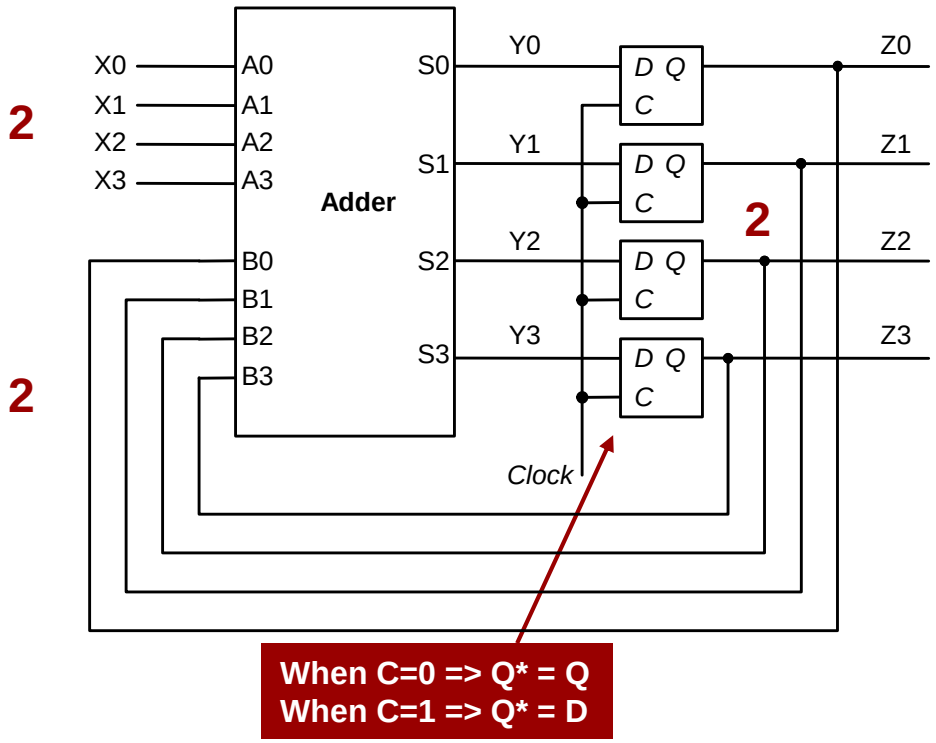
- Since the clock starts off low, the outputs of the latches can't change and just hold at 0

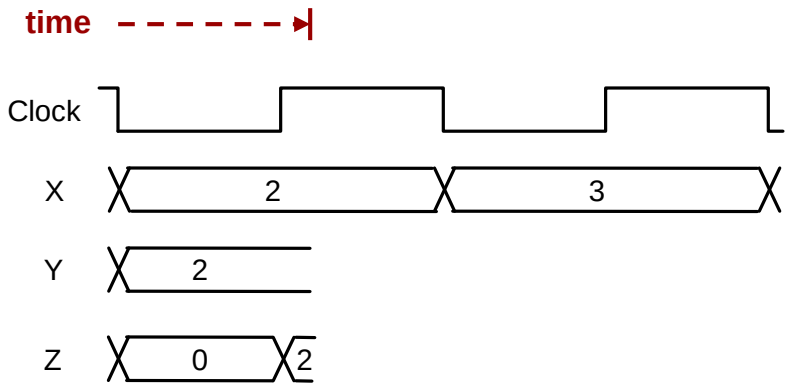




Adding a Sequence of Numbers

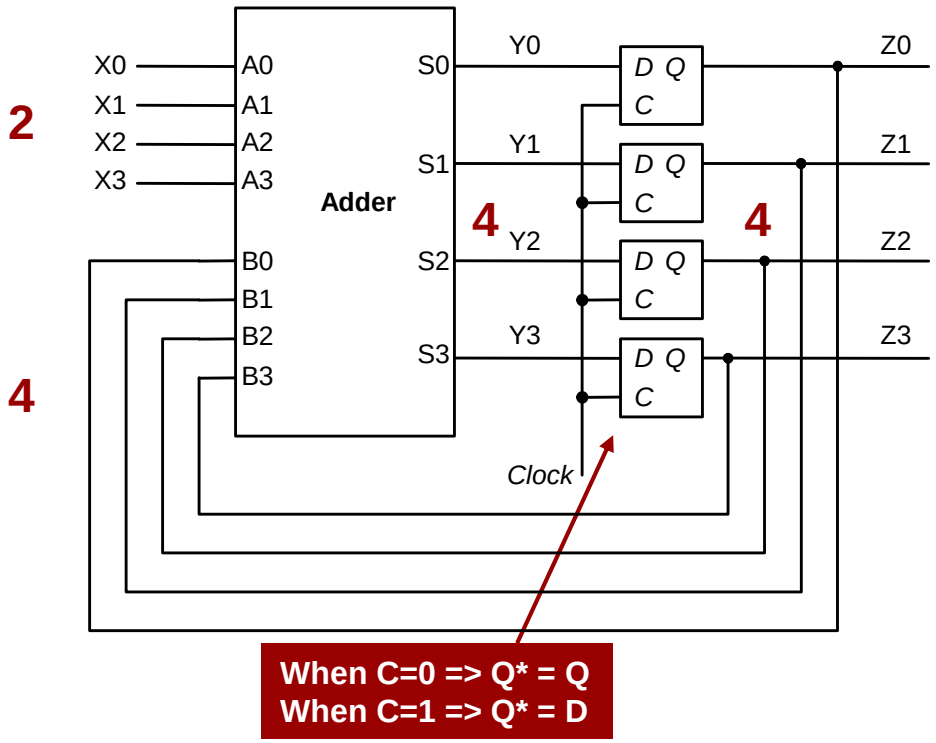
- When the clock goes high the D goes through to Q and is free to loop back around

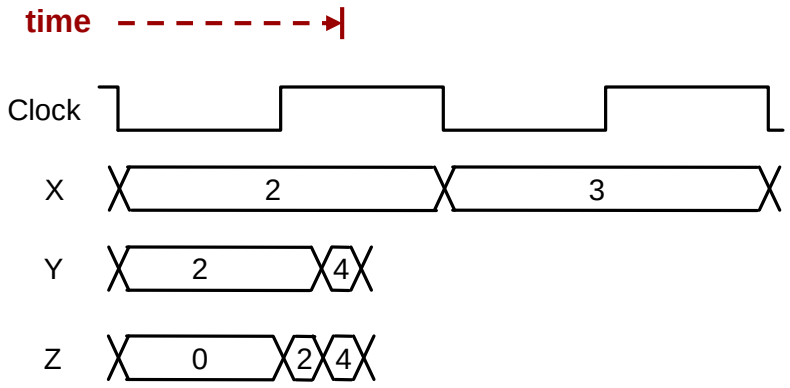




Adding a Sequence of Numbers

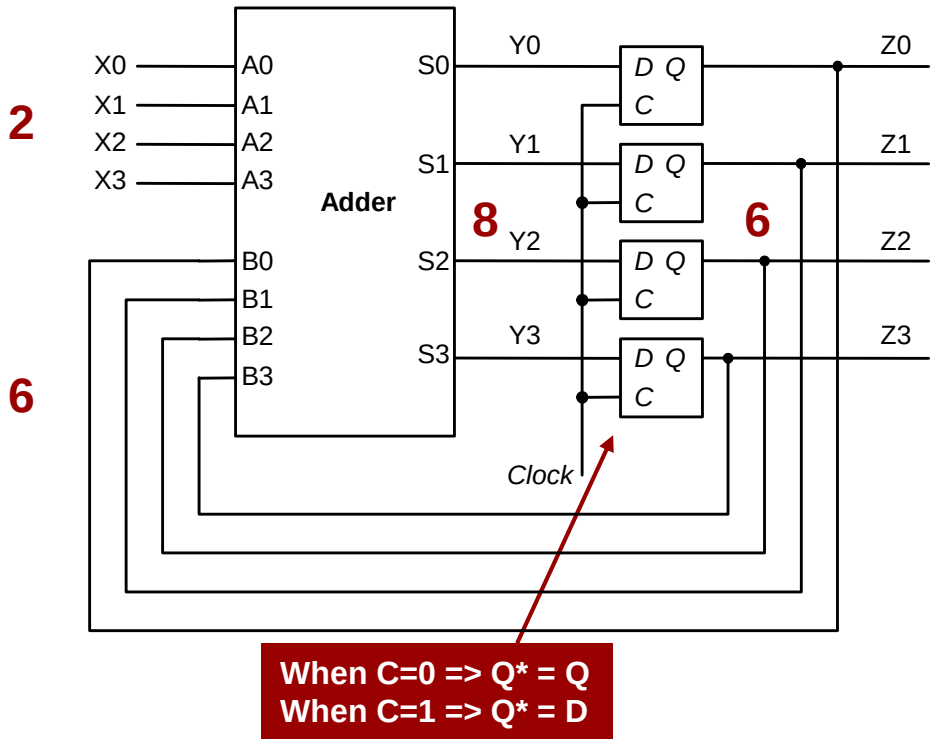
- Once it loops back around it will be added again, change the Y value and go through to Z and loop back around again

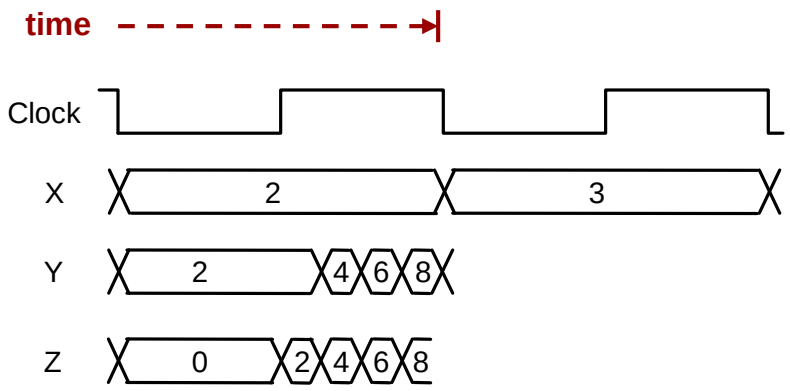




Adding a Sequence of Numbers

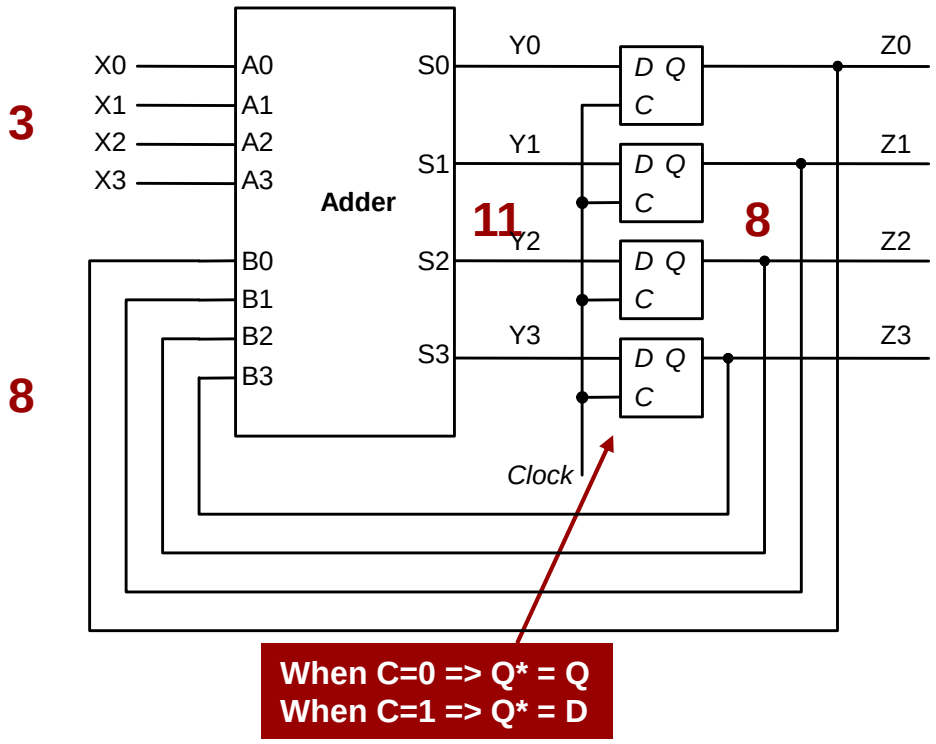
- This feedback loop continues until the clock goes low again

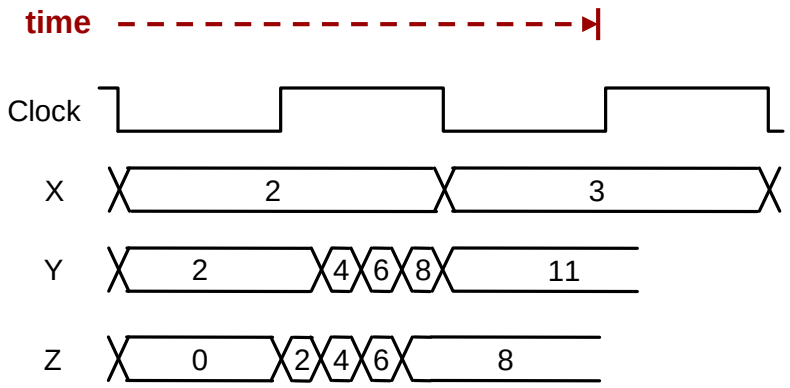




Adding a Sequence of Numbers

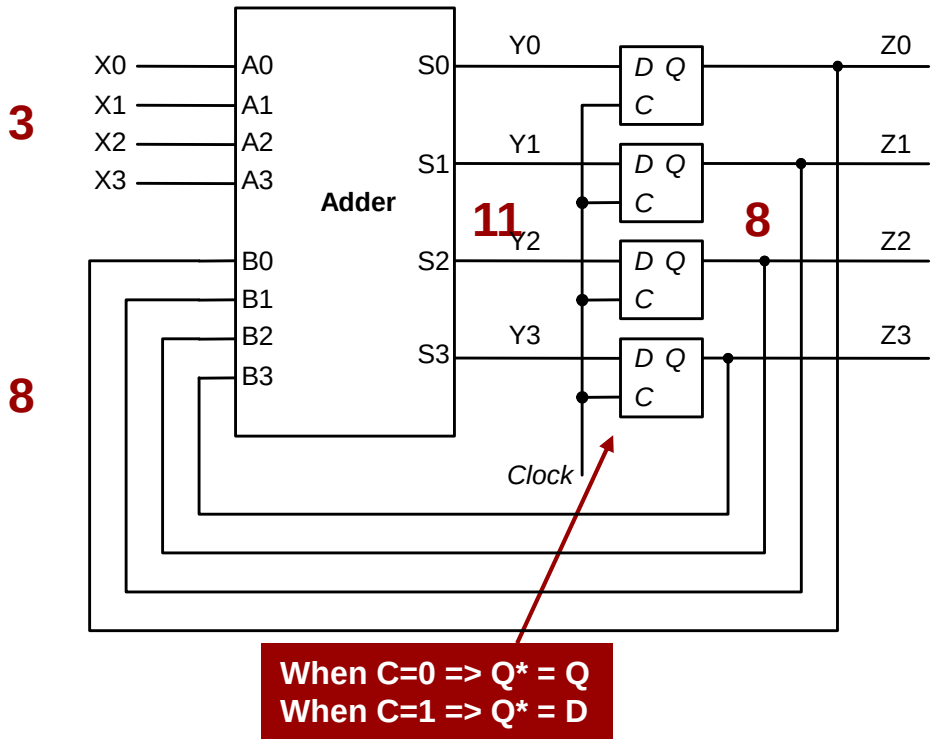
- When the clock goes low again, the outputs will hold at their current value 8 until the clock goes high

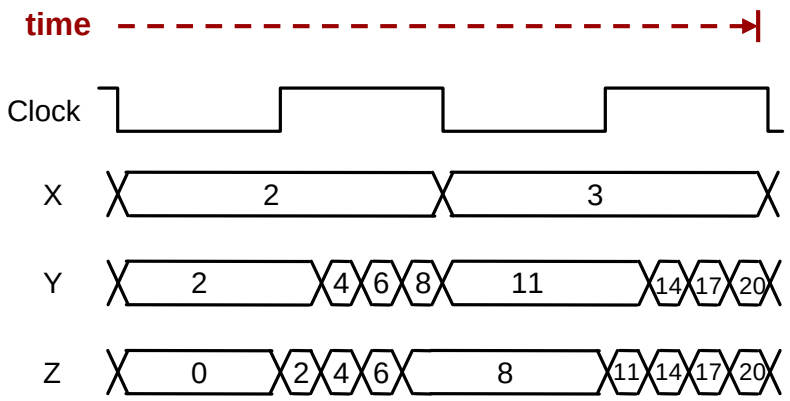




Adding a Sequence of Numbers

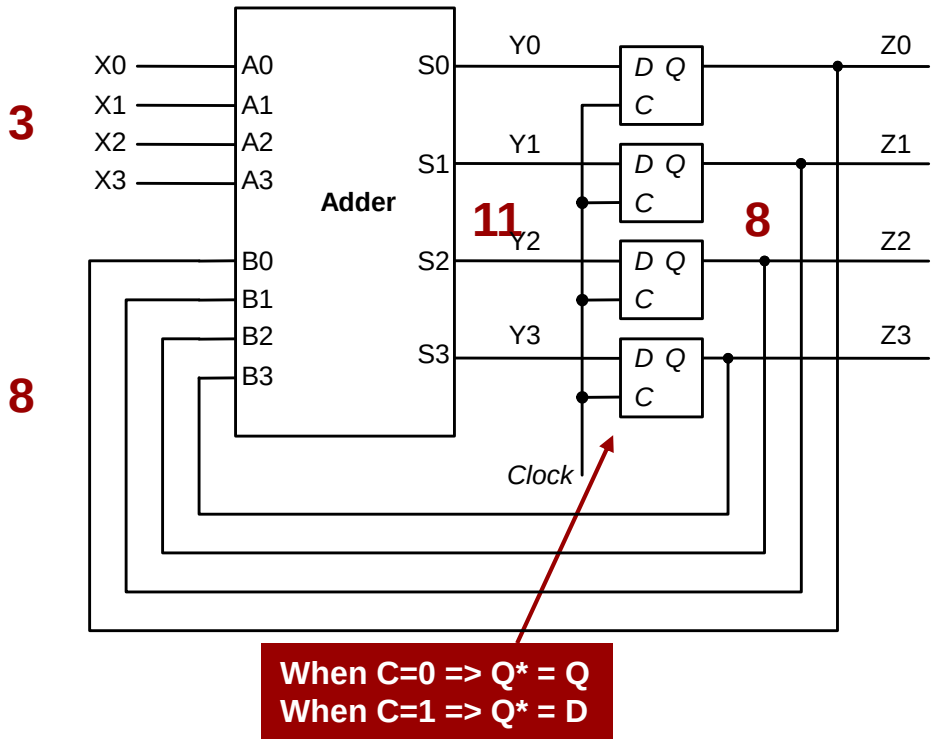
- When the clock goes high, the outputs will be free to change and we will get the feedback problem

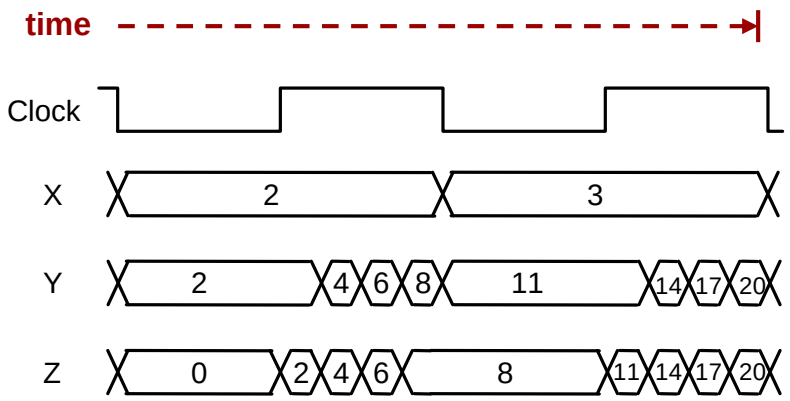




Adding a Sequence of Numbers

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





FLIP-FLOPS

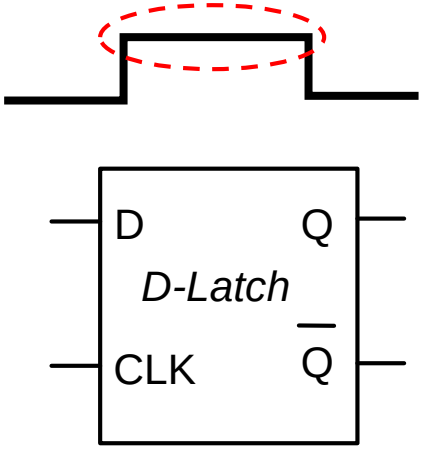
Flip-Flops vs. Latches

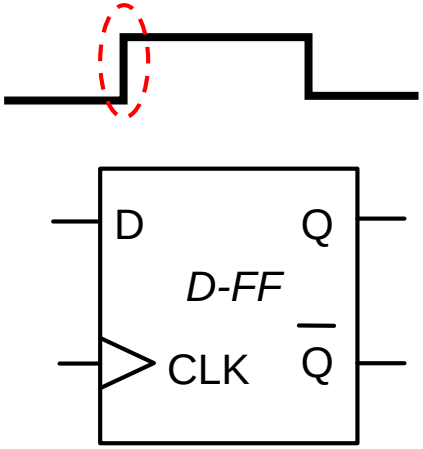
Latches

- Asynchronous
- Clock/Enable input
- Level Sensitive
  - Outputs can change anytime Clock = 1

Flip-Flops

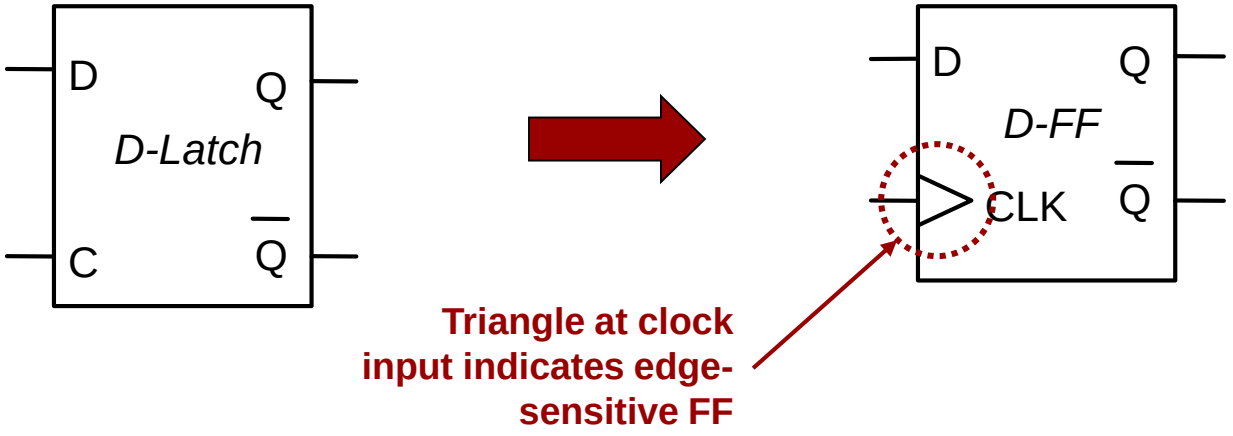
- Synchronous
- Clock Input
- Edge-Sensitive
  - Outputs change only on the positive (negative) edges





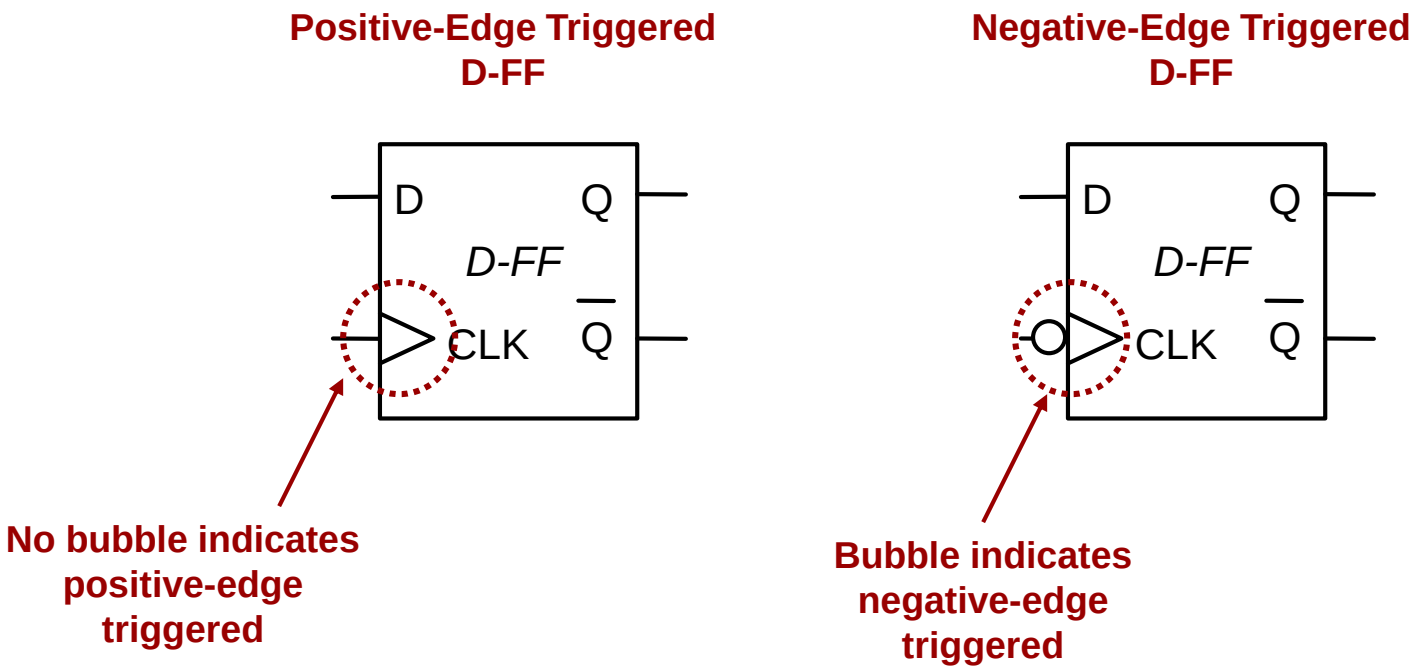
Flip-Flops

- Change D Latches to D Flip-Flops



Flip-Flops

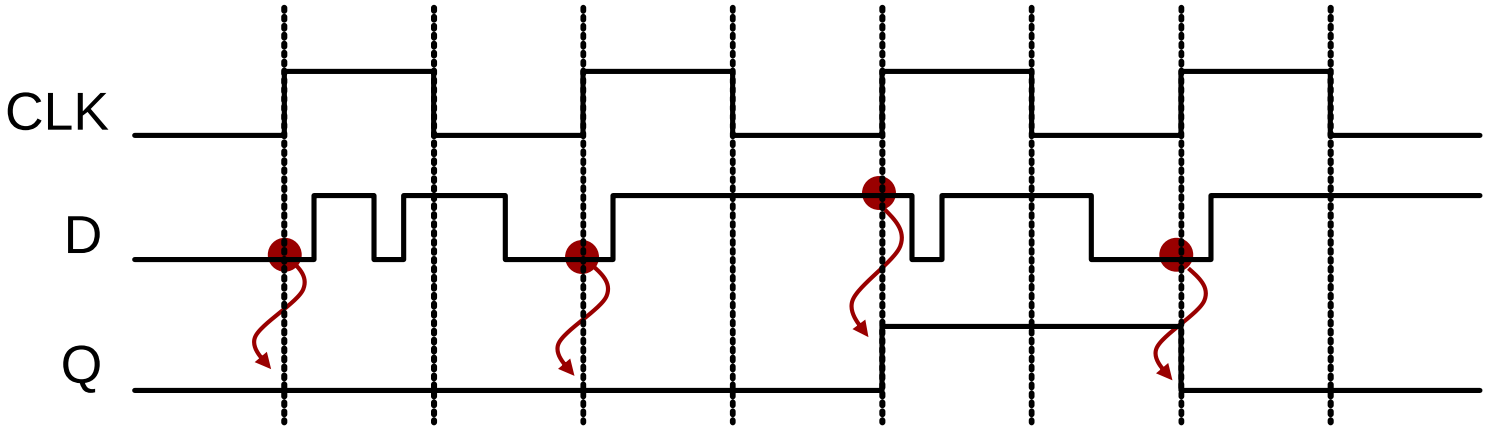
- To indicate negative-edge triggered use a bubble in front of the clock input



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

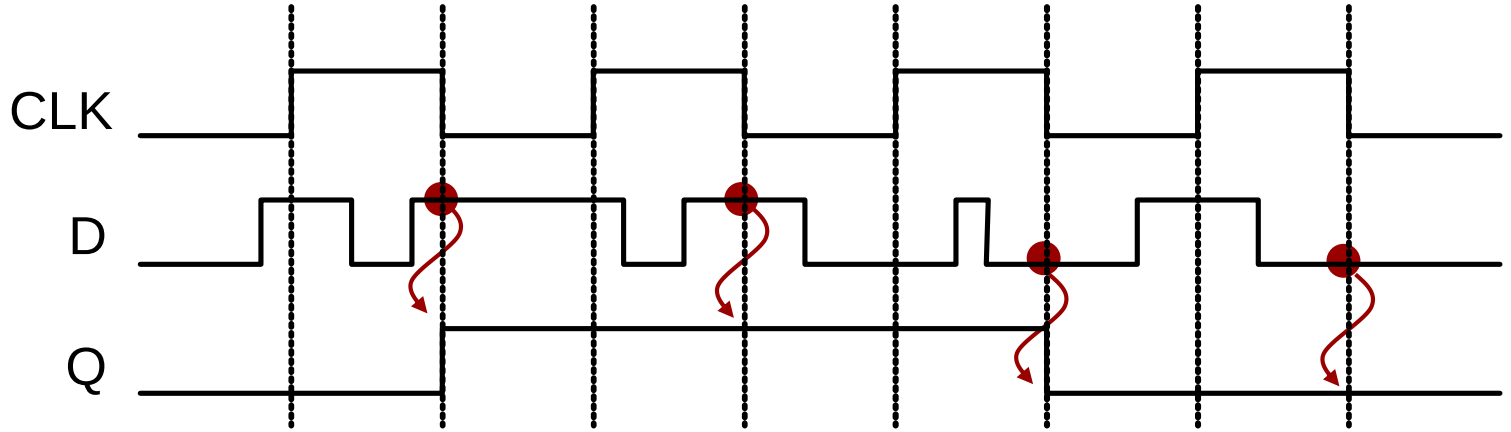


Q only samples D at the positive edges and then holds that value until the next edge

Negative-Edge Triggered D-FF

- Q looks at D only at the negative-edge

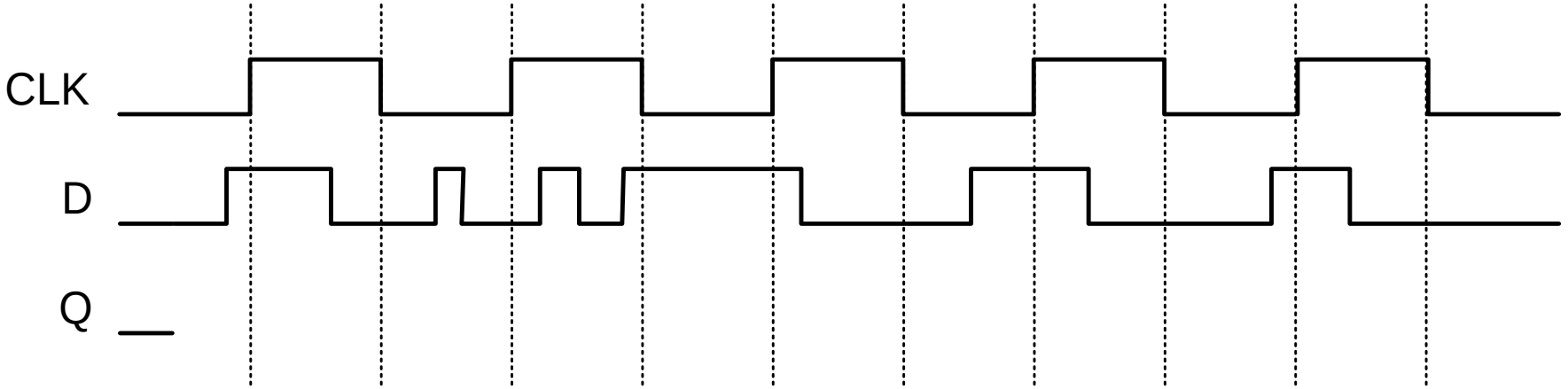
CLK	D	Q*	Q'*
0	x	Q	Q'
1	x	Q	Q'
↓	0	0	1
↓	1	1	0



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

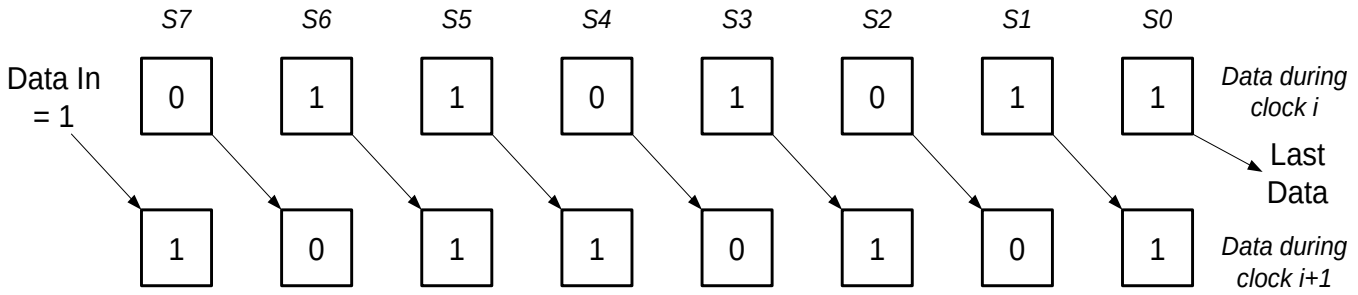
D FF Example

- Assume positive edge-triggered FF

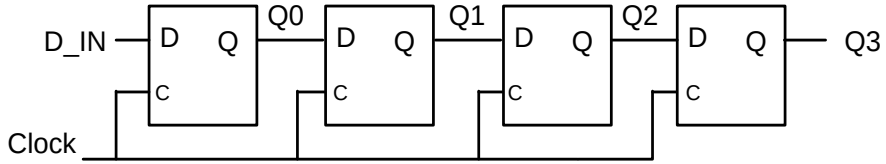


Shift Register

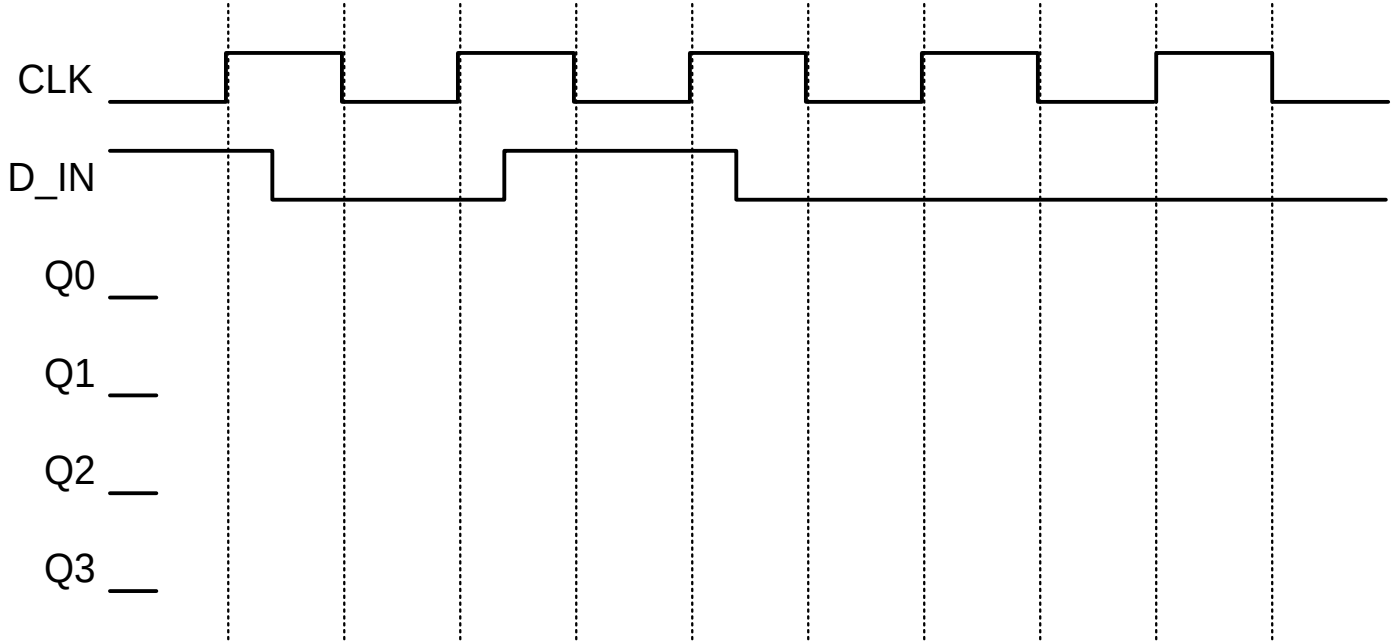
- A shift register is a device that acts as a ‘queue’ or ‘FIFO’ (First-in, First-Out).
- It can store n bits and each bit moves one step forward each clock cycle
  - One bit comes in the overall input per clock
  - One bit ‘falls out’ the output per clock



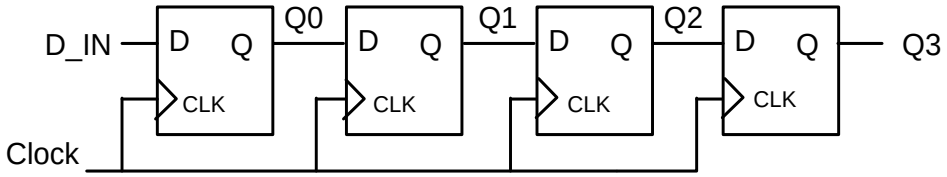
Shift Register



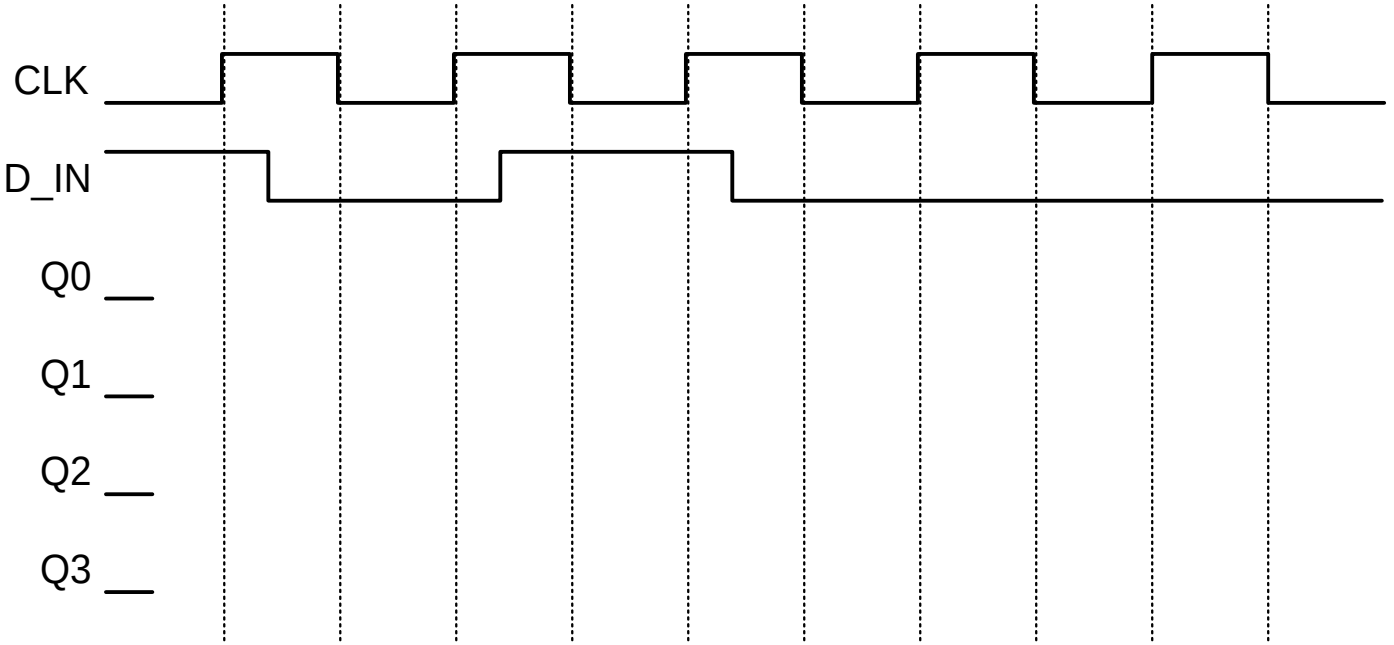
Shift Register w/ Latches



Shift Register



Shift Register w/ FF's

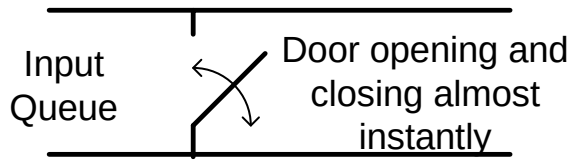


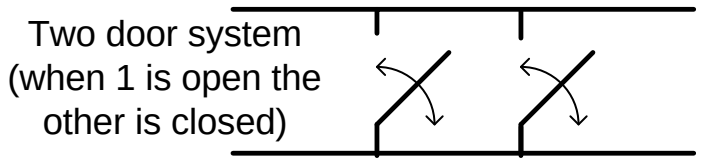
When we want to ensure an output updates only ONCE per clock, we need to use flip-flops (not latches or bistables)!

BUILDING A FLIP FLOP

Building an Edge-Triggered Device

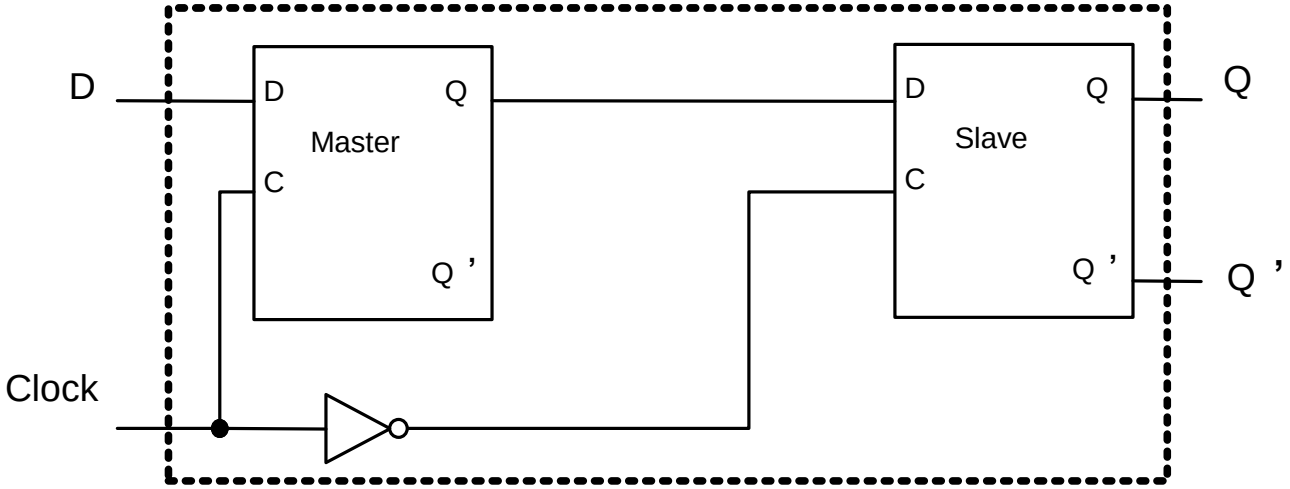
- We generally build FFs from latches
- To build a device that can only change at 1 instant (clock edge) we can:
  - Try to only enable 1 latch for a small instant in time
  - Use two latches running on opposite clock phases





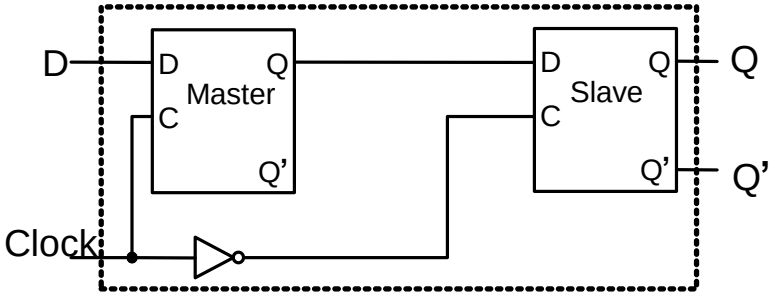
Master-Slave D-FF

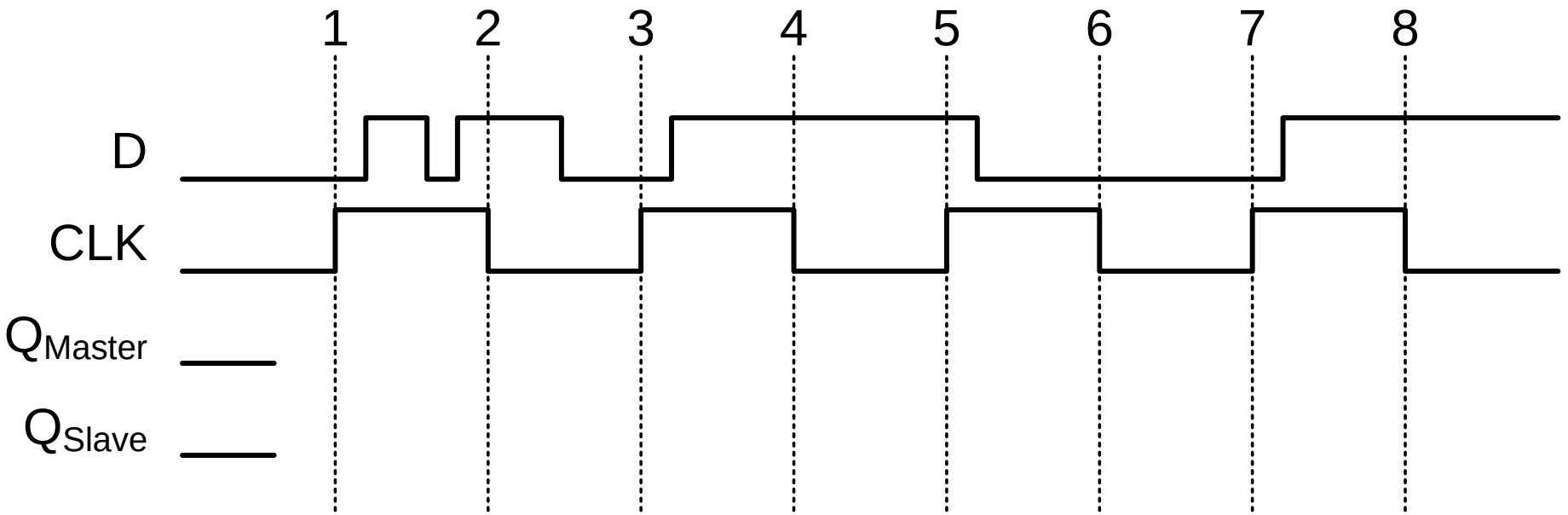
- To build an **edge-triggered** D-FF we can use two D-Latches



These 2 latches form a flip-flop

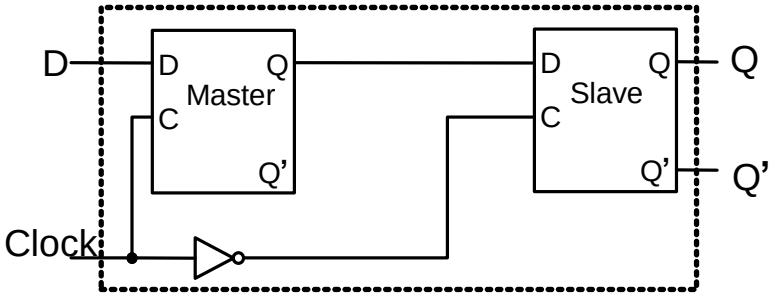
Complete the Waveform



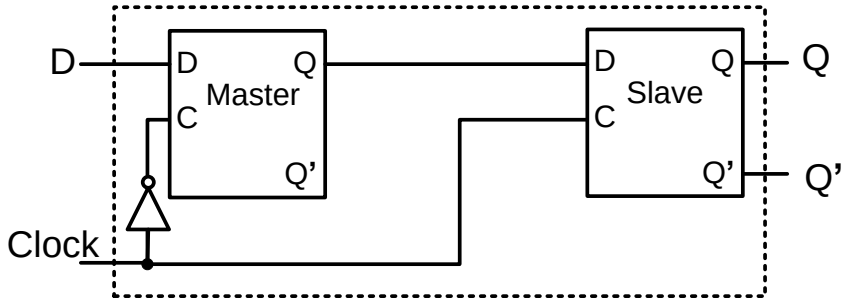


Master-Slave D-FF

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



Negative-Edge Triggered

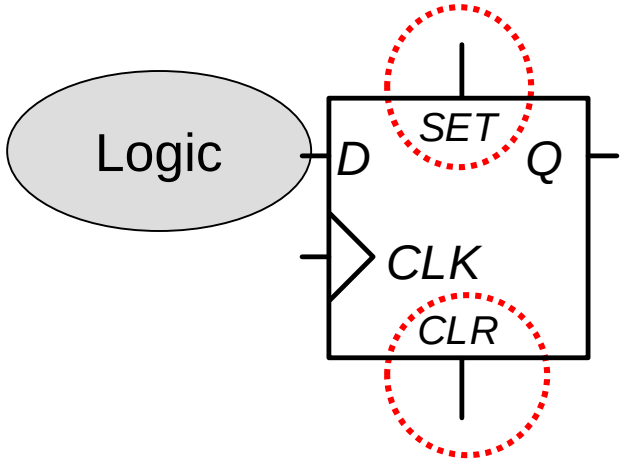


Positive-Edge Triggered

INITIALIZING OUTPUTS

Initializing Outputs

- Need to be able to initialize Q to a known value (0 or 1)
- FF inputs are often connected to logic that will produce values after initialization
- Two extra inputs are often included: (PRE)SET and CLEAR



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

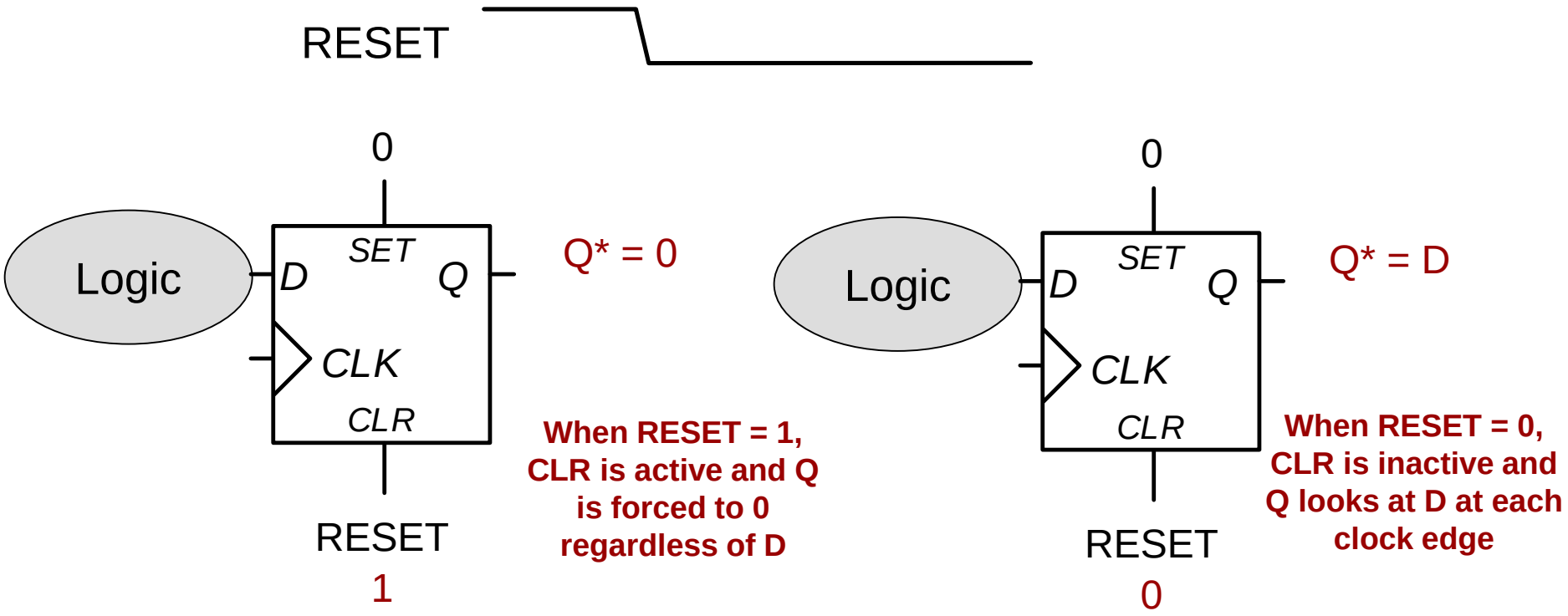
Note: CLR and SET have priority over normal FF inputs

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

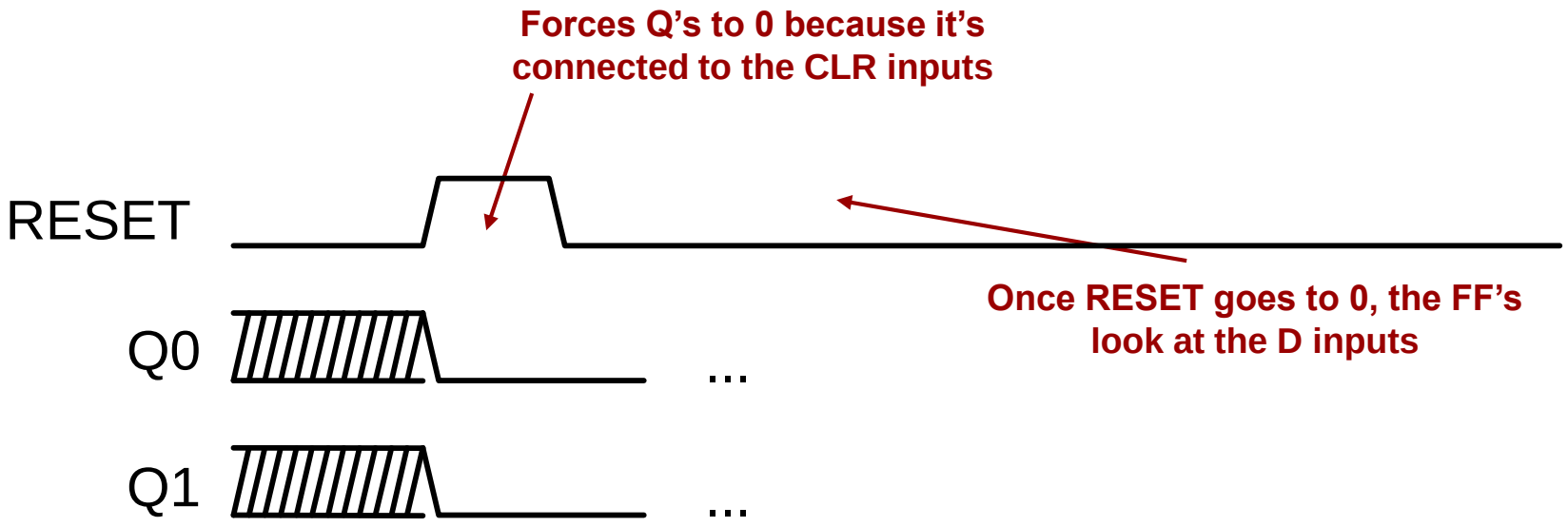
Initializing Outputs

- 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



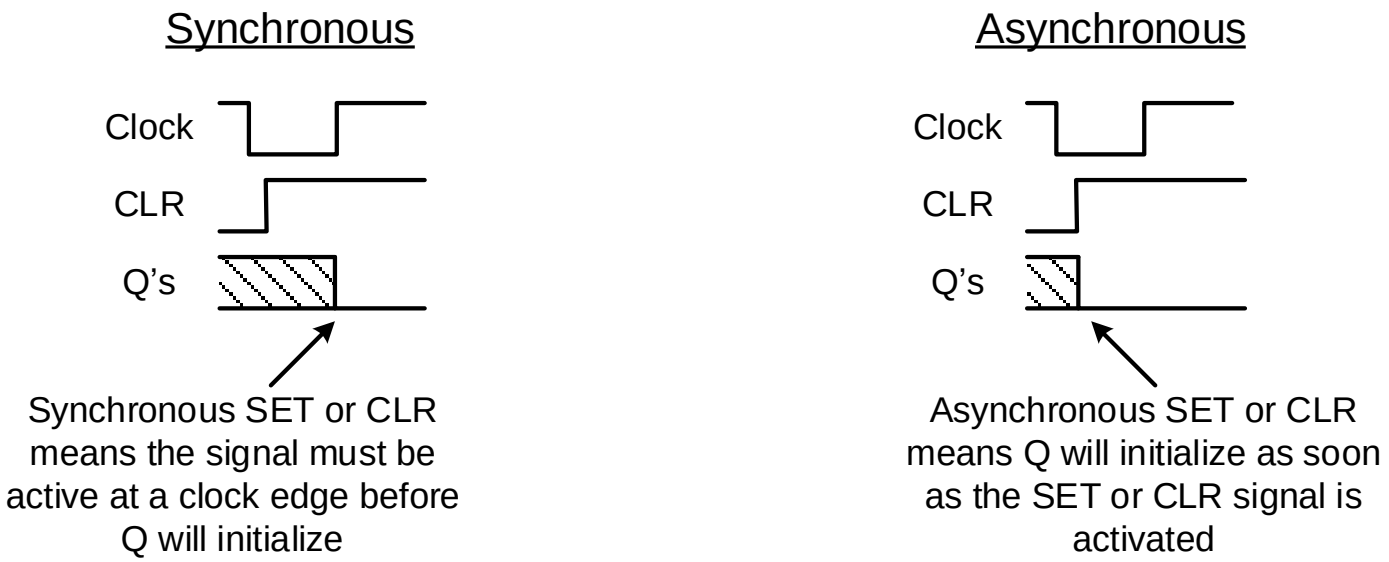
Implementing an Initial State

- When RESET is activated Q's initialize and then when it goes back to 1 the Q's look at the D inputs



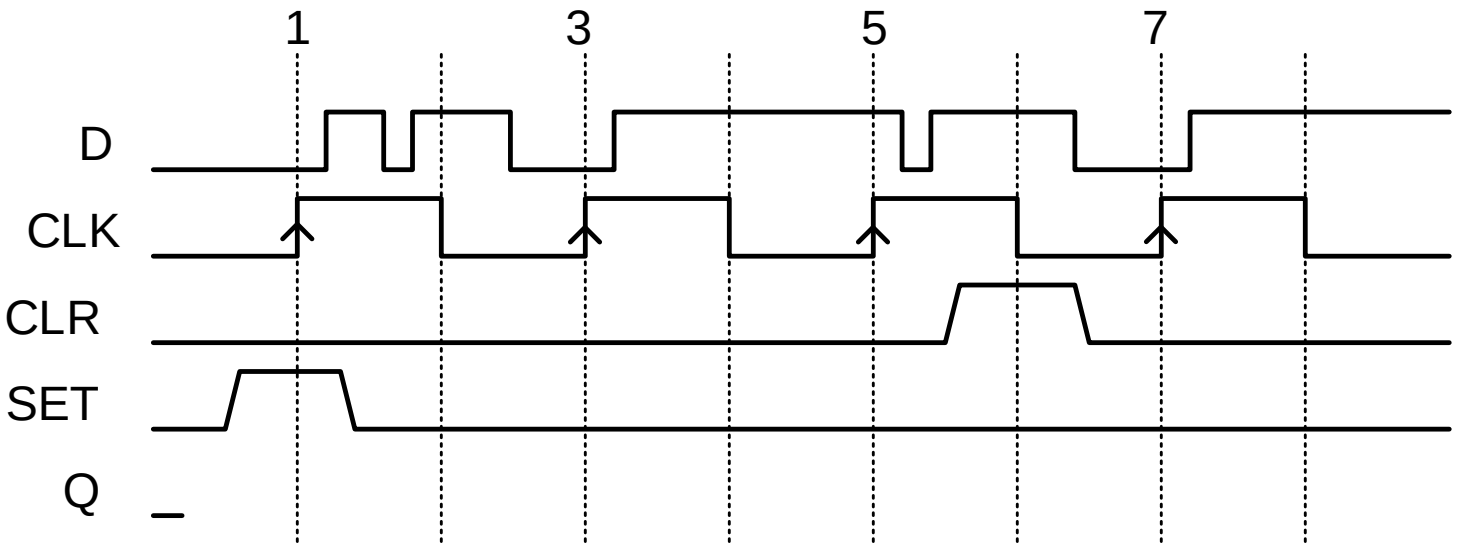
Synchronous vs. Asynchronous

- 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 when signal is activated
  - Synchronous…initialize at clock edge

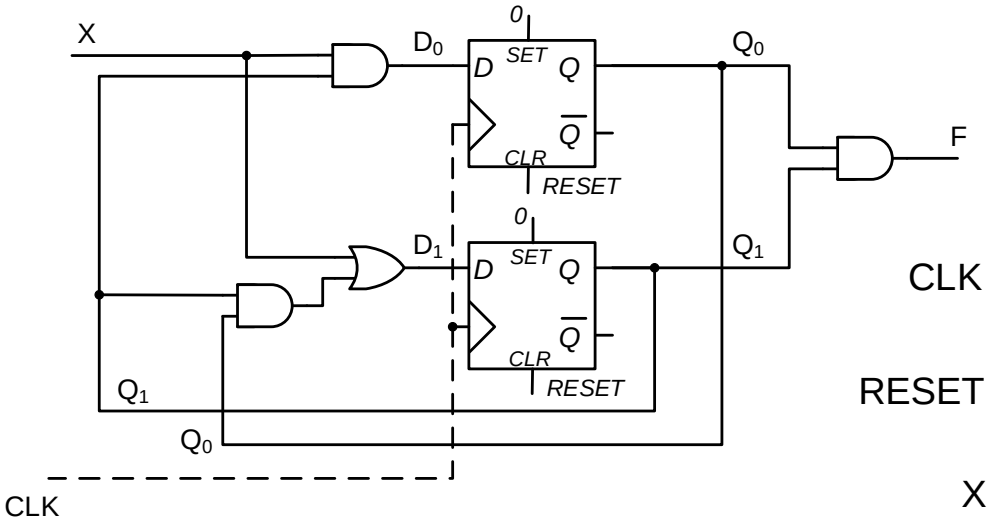


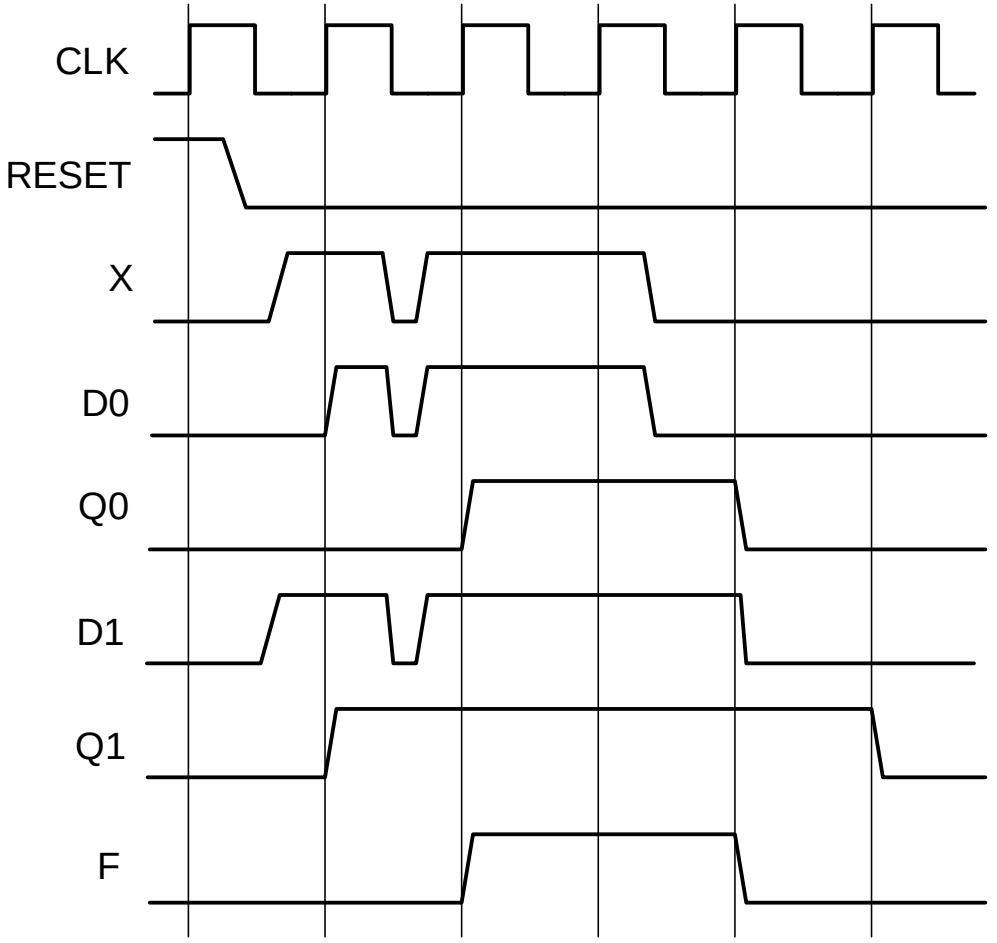
Set / Clear Example

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



Exercise


X
D0
0
SET
D
Q
Q
CLR
RESET
Q0
F
D1
0
SET
D
Q
Q
CLR
RESET
Q1
Q1
Q0
CLK


CLK
RESET
X
D0
Q0
D1
Q1
F

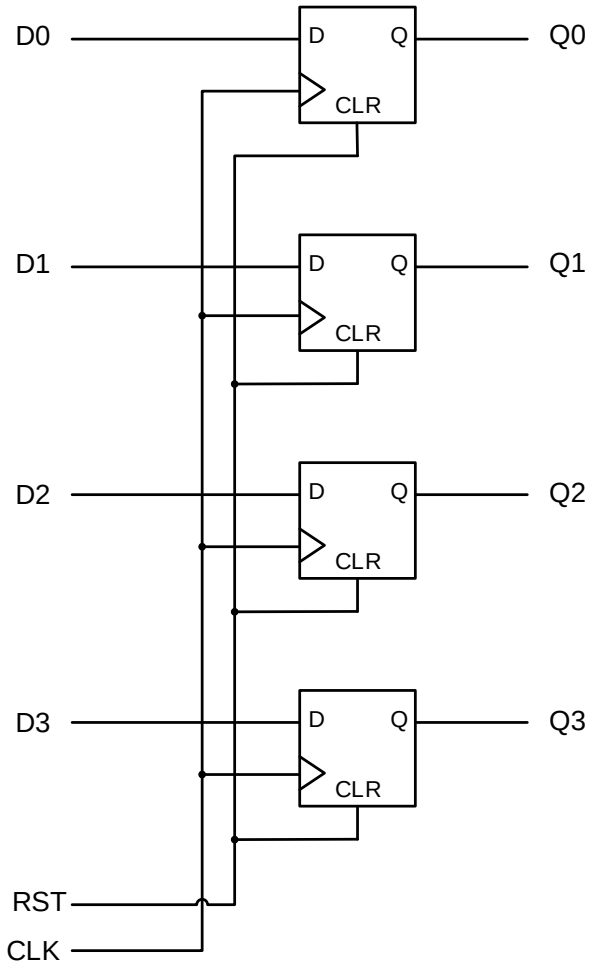
Using muxes to control when register save data

REGISTER WITH ENABLES

Register Resets/Clears

- When the power turns on the bit stored in a flip-flop will initialize to a random value
- Better to initialize it to a known value (usually 0's)
- Can use an asynchronous or synchronous "reset" to force the flip-flops to 0's

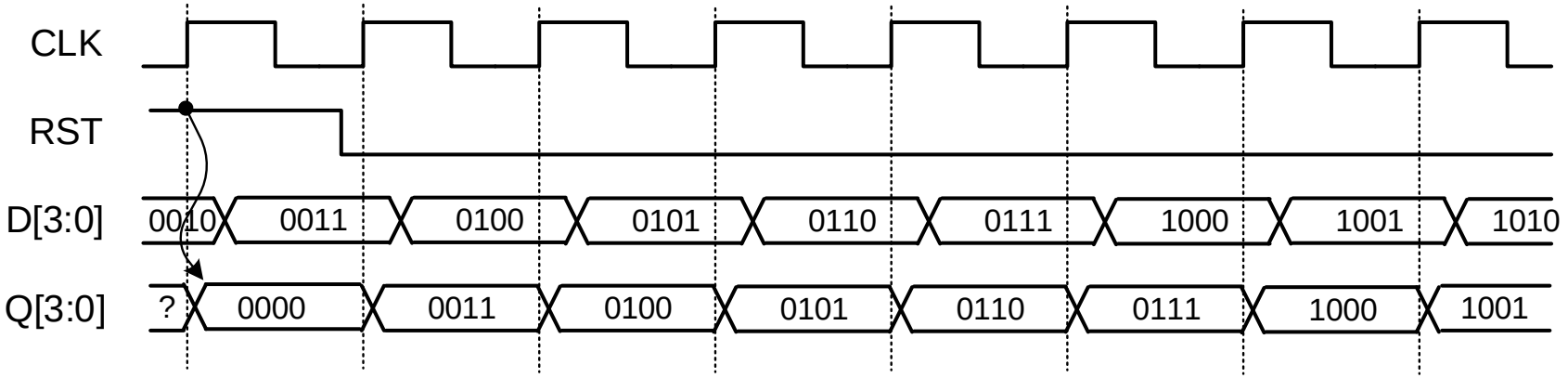
CLK	RST	D_i	Q_i*
1,0	X	X	Q_i
⇈	1	X	0
⇈	0	0	0
⇈	0	1	1



4-bit Register

Register Problem

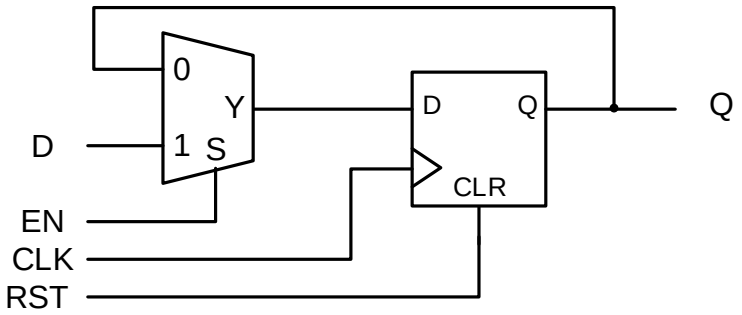
- Whatever the D value is at the clock edge is sampled and passed to the Q output until the next clock edge
- Problem: Register will save data on EVERY edge
  - Often we want the ability to save on one edge and then keep that value for many more cycles



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

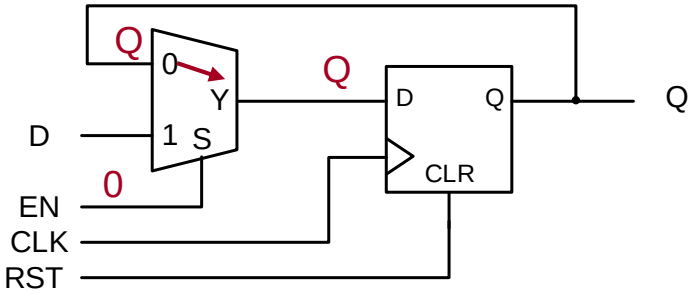


FF with Data Enable
(Always clocks, but selectively chooses old value, Q, or new value D)

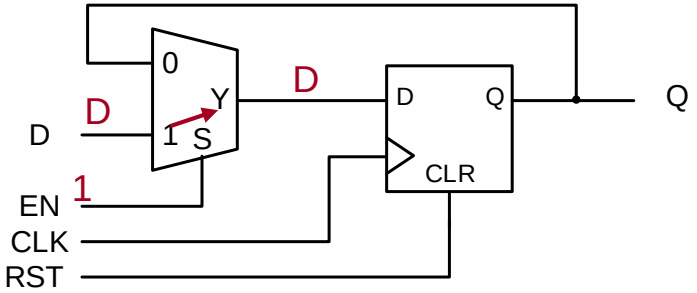
CLK	RST	EN	D_i	Q_i*
0,1	X	X	X	Q_i
⇈	1	X	X	0
⇈	0	0	X	Q_i
⇈	0	1	0	0
⇈	0	1	1	1

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 will change at the edge based on D



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

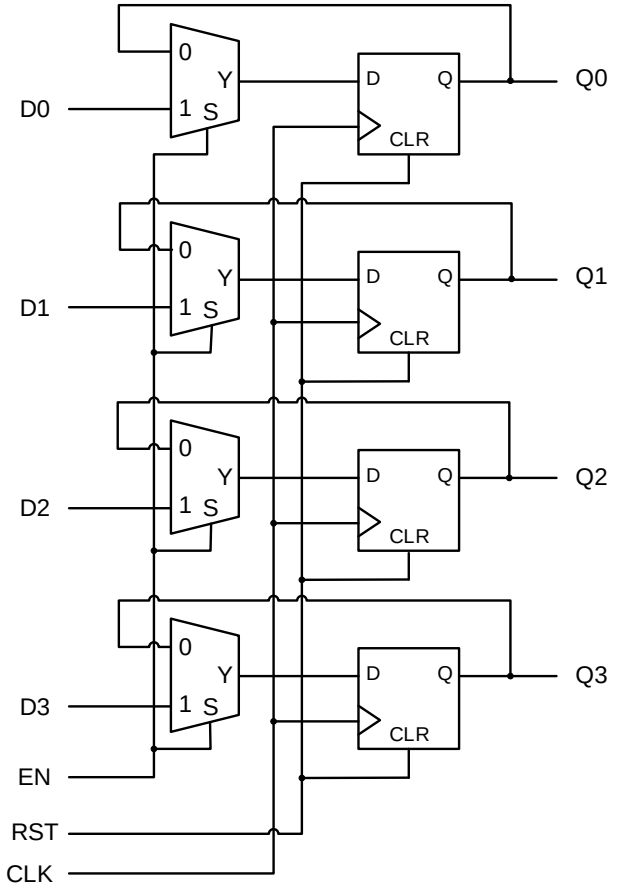


When EN=1, D input is passed to FF input

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

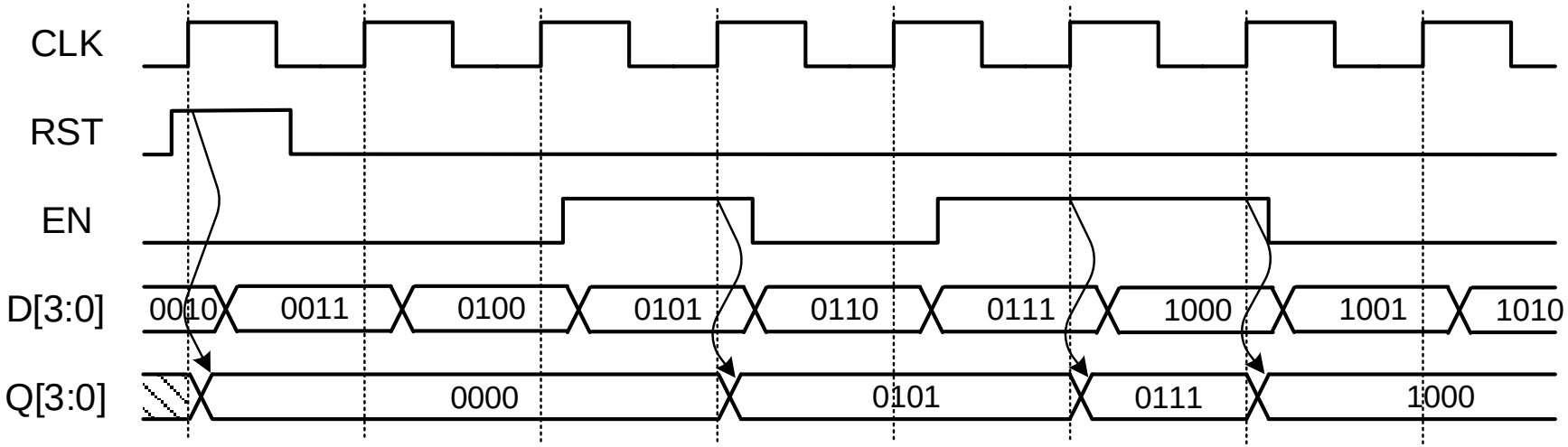
CLK	RST	EN	D_i	Q_i*
0,1	X	X	X	Q_i
⇈	1	X	X	0
⇈	0	0	X	Q_i
⇈	0	1	0	0
⇈	0	1	1	1



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

Registers w/ Enables

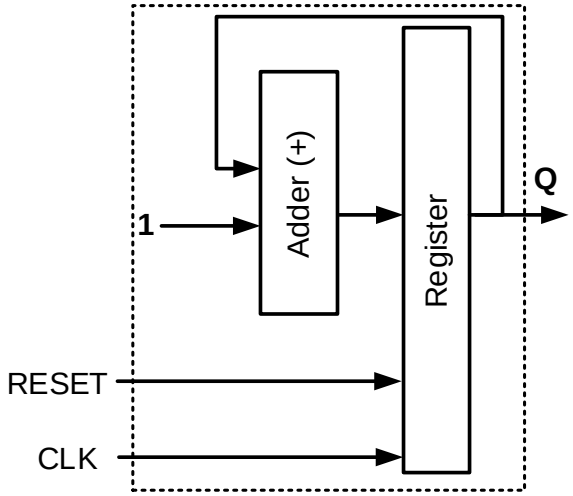
- The D value is sampled at the clock edge only if the enable is active
- Otherwise the current Q value is maintained



COUNTERS

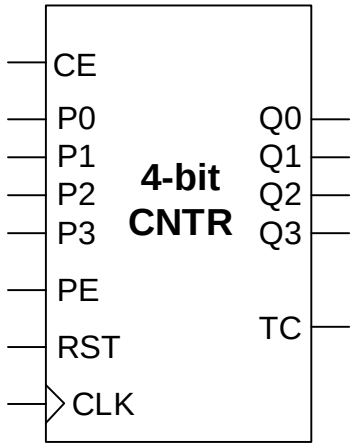
Counters

- 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
  - Enables: Will not count at edge if EN=0
  - Parallel Load Inputs: Can initialize count to a value P (i.e. Q* = P rather than Q+1)



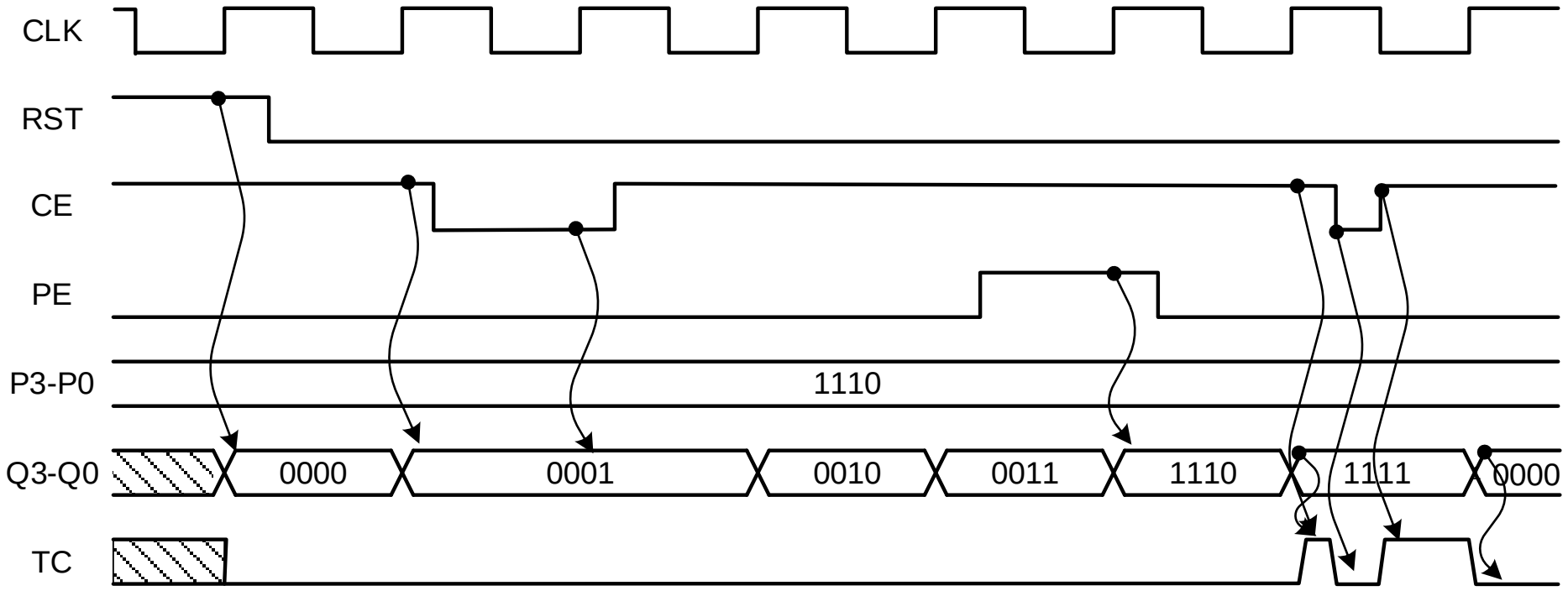
Sample 4-bit Counter

- 4-bit Up Counter
  - RST: synchronous reset input
  - PE and P_i inputs: loads Q with P when PE is active
  - CE: Count Enable
    - Must be active for the counter to count up
  - TC (Terminal Count) output
    - Active when Q=1111 AND counter is enabled
    - TC = EN•Q3•Q2•Q1•Q0
      - Mealy output
    - Indicates that on the next edge it will roll over to 0000



CLK	RST	PE	CE	Q*
0,1				
⇈				
⇈				
⇈				
⇈				

Counters



SR=active at clock edge, thus Q=0

Q*=Q+1

Enable = off, thus Q holds

Q*=Q+1

Q*=Q+1

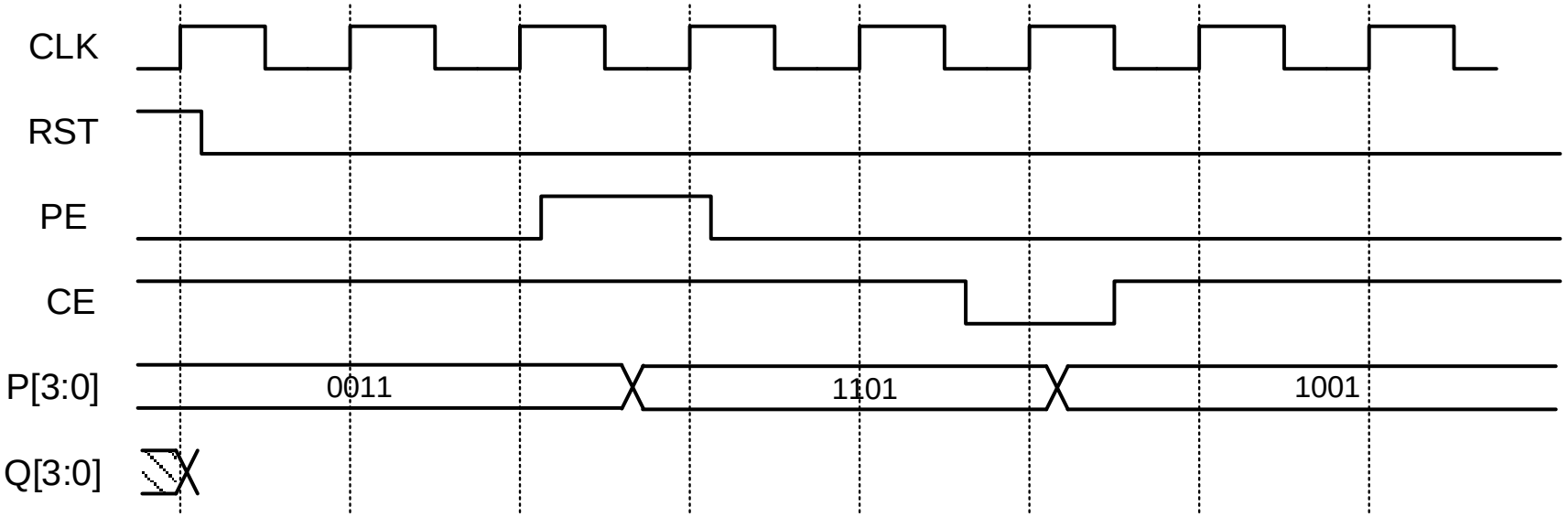
PE = active, thus Q=P

Q*=Q+1

Q*=Q+1

Mealy TC output: EN•Q3•Q2•Q1•Q0

Counter Exercise



Counter Design

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