ELECTRONICS SYSTEM DESIGN

SECTION-3

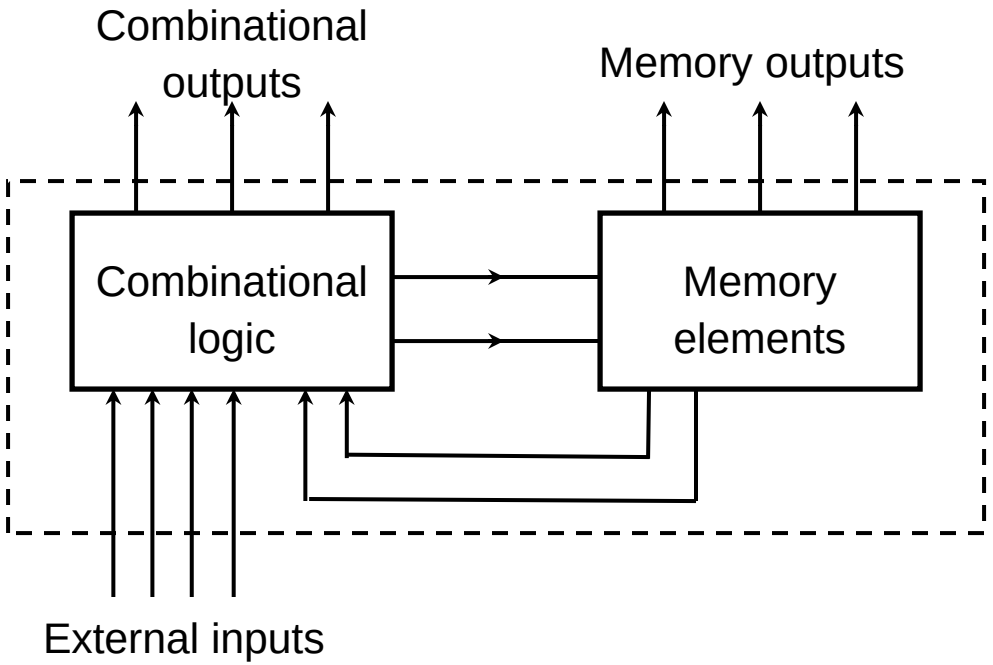
SEQUENTIAL MACHINES

Lecture 11: Sequential Logic Latches & Flip-flops

- Introduction
- Memory Elements
- Pulse-Triggered Latch
  - S-R Latch
  - Gated S-R Latch
  - Gated D Latch
- Edge-Triggered Flip-flops
  - S-R Flip-flop
  - D Flip-flop
  - J-K Flip-flop
  - T Flip-flop
- Asynchronous Inputs

Introduction

- A **sequential circuit** consists of a *feedback path*, and employs some *memory elements*.



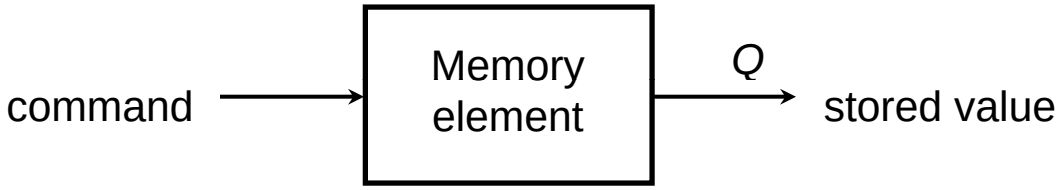
Sequential circuit = Combinational logic + Memory Elements

Introduction

- There are two types of sequential circuits:
  - *synchronous*: outputs change only at specific time
  - *asynchronous*: outputs change at any time
- *Multivibrator*: a class of sequential circuits. They can be:
  - *bistable* (2 stable states)
  - *monostable* or *one-shot* (1 stable state)
  - *astable* (no stable state)
- Bistable logic devices: *latches* and *flip-flops*.
- Latches and flip-flops differ in the method used for changing their state.

Memory Elements

- **Memory element**: a device which can remember value indefinitely, or change value on command from its inputs.



- Characteristic table:

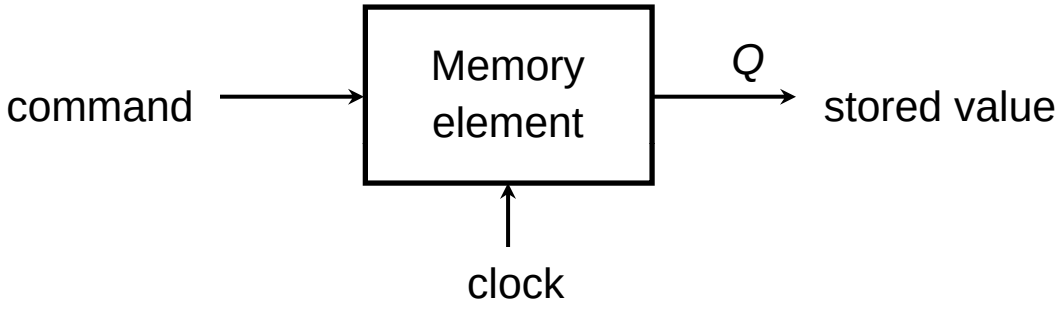
Command (at time *t*)	*Q(t)*	*Q(t+1)*
Set	X	1
Reset	X	0
Memorise / No Change	0	0
	1	1

Q(t): current state

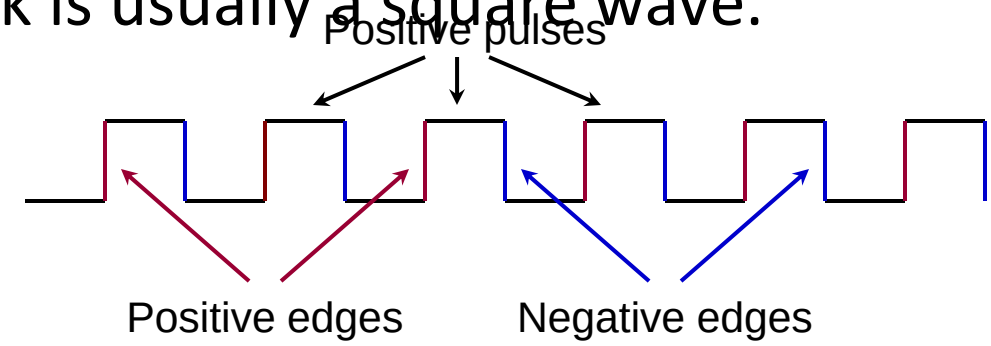
Q(t+1) or Q^+: next state

Memory Elements

- Memory element with clock. Flip-flops are memory elements that change state on clock signals.



- Clock is usually a square wave.



Memory Elements

- Two types of triggering/activation:
  - pulse-triggered
  - edge-triggered
- Pulse-triggered
  - latches
  - ON = 1, OFF = 0
- Edge-triggered
  - flip-flops
  - positive edge-triggered (ON = from 0 to 1; OFF = other time)
  - negative edge-triggered (ON = from 1 to 0; OFF = other time)

S-R Latch

- *Complementary* outputs: *Q* and *Q'*.
- When *Q* is HIGH, the latch is in *SET* state.
- When *Q* is LOW, the latch is in *RESET* state.
- For *active-HIGH input* S-R latch (also known as NOR gate latch),

  R=HIGH (and *S*=LOW) ⇨ RESET state
  S=HIGH (and *R*=LOW) ⇨ SET state
  both inputs LOW ⇨ no change
  both inputs HIGH ⇨ *Q* and *Q'* both LOW (invalid)!

S-R Latch

- For *active-LOW input* S'-R' latch (also known as NAND gate latch),

  R'=LOW (and *S'*=HIGH) ⇨ RESET state

  S'=LOW (and *R'*=HIGH) ⇨ SET state

  both inputs HIGH ⇨ no change

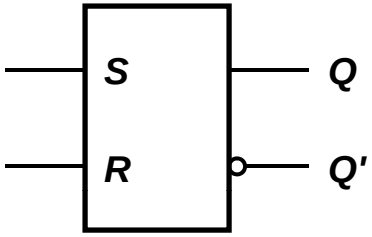
  both inputs LOW ⇨ *Q* and *Q'* both HIGH (invalid)!

- Drawback of S-R latch: invalid condition exists and must be avoided.

S-R Latch

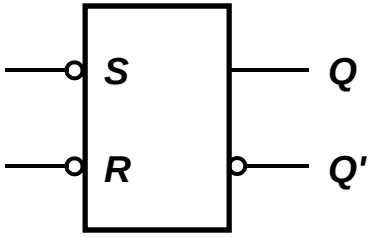
- Characteristics table for active-high input S-R latch:

S	*R*	*Q*	*Q′*	
0	0	NC	NC	No change. Latch remained in present state.
1	0	1	0	Latch SET.
0	1	0	1	Latch RESET.
1	1	0	0	Invalid condition.



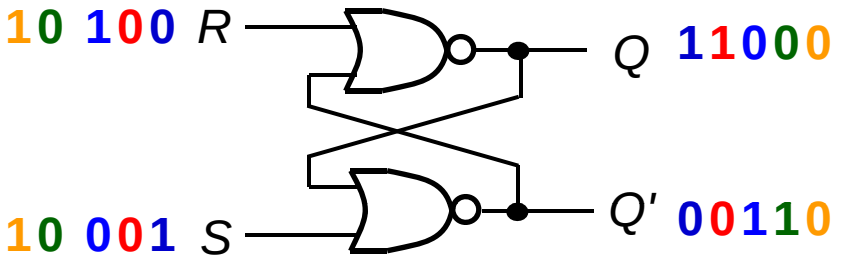
- Characteristics table for active-low input S'-R' latch:

S′	*R′*	*Q*	*Q′*	
1	1	NC	NC	No change. Latch remained in present state.
0	1	1	0	Latch SET.
1	0	0	1	Latch RESET.
0	0	1	1	Invalid condition.



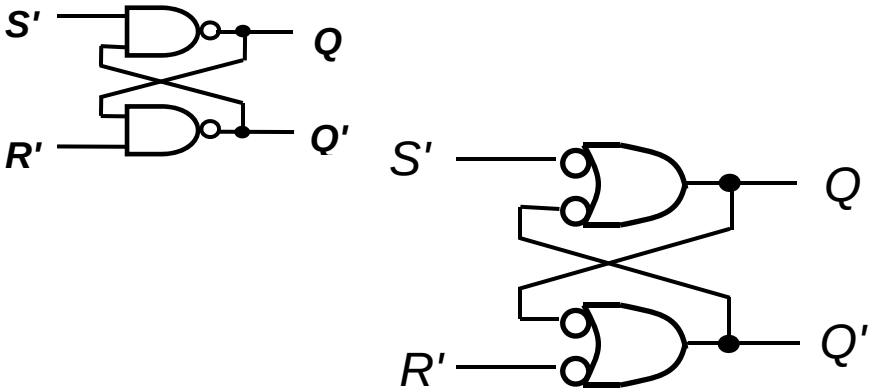
S-R Latch

- Active-HIGH input S-R latch



S	R	Q	Q′	
1	0	1	0	initial
0	0	1	0	(afer S=1, R=0)
0	1	0	1	
0	0	0	1	(after S=0, R=1)
1	1	0	0	invalid!

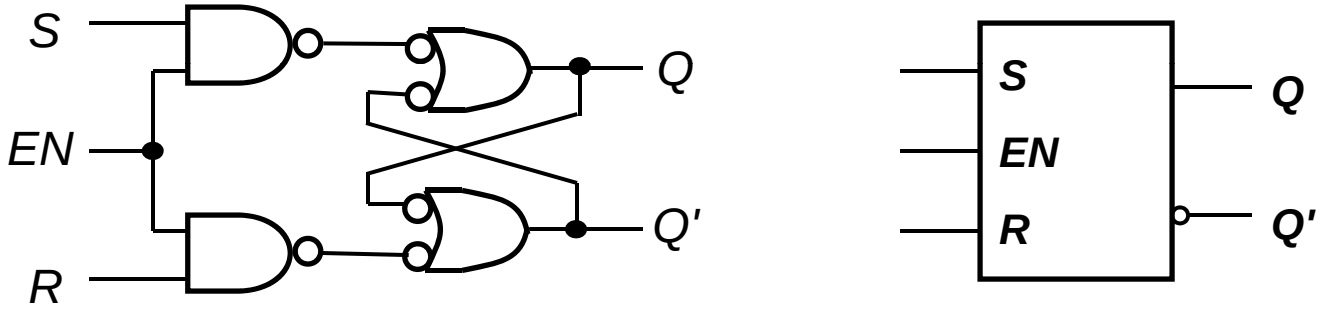
- Active-LOW input S'-R' latch



S′	R′	Q	Q′	
1	0	0	1	initial
1	1	0	1	(afer S′=1, R′=0)
0	1	1	0	
1	1	1	0	(after S′=0, R′=1)
0	0	1	1	invalid!

Gated S-R Latch

- S-R latch + *enable input* (*EN*) and 2 NAND gates → *gated S-R latch*.



Gated S-R Latch

- Outputs change (if necessary) only when *EN* is HIGH.
- Under what condition does the invalid state occur?
- Characteristic table:

***EN*=1**

Q(t)	*S*	*R*	*Q(t+1)*
0	0	0	0
0	0	1	0
0	1	0	1
0	1	1	indeterminate
1	0	0	1
1	0	1	0
1	1	0	1
1	1	1	indeterminate

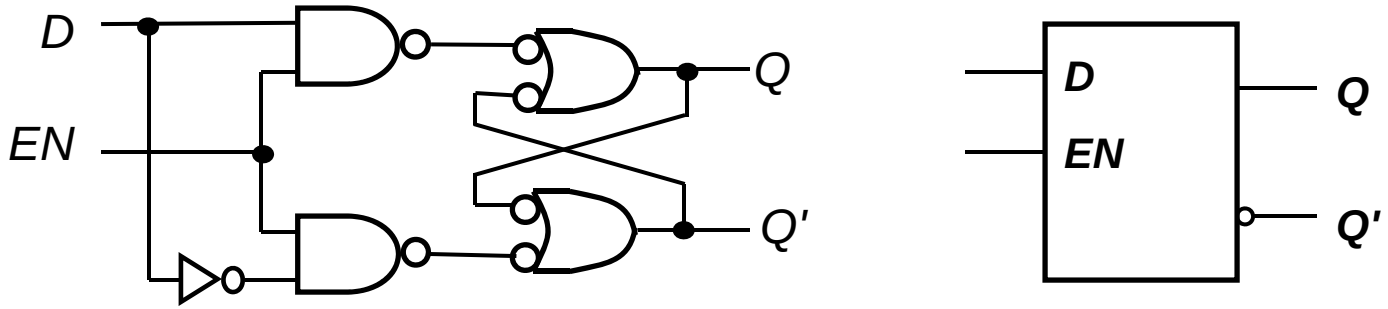
S	*R*	*Q(t+1)*	
0	0	*Q(t)*	No change
0	1	0	Reset
1	0	1	Set
1	1	indeterminate	

$Q(t+1) = S + R'.Q$

$S.R = 0$

Gated D Latch

- Make *R* input equal to *S'* → *gated D latch.*
- *D* latch eliminates the undesirable condition of invalid state in the *S-R* latch.



Gated D Latch

- When *EN* is HIGH,
  - *D*=HIGH $\rightarrow$ latch is SET
  - *D*=LOW $\rightarrow$ latch is RESET
- Hence when *EN* is HIGH, *Q* 'follows' the *D* (data) input.
- Characteristic table:

EN	*D*	*Q(t+1)*	
1	**0**	**0**	**Reset**
1	**1**	**1**	**Set**
0	**X**	***Q(t)***	**No change**

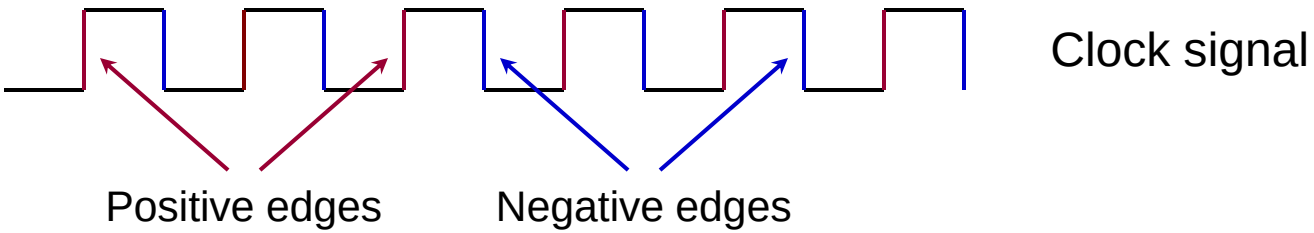
When *EN*=1, $Q(t+1) = D$

Latch Circuits: Not Suitable

- Latch circuits are not suitable in synchronous logic circuits.
- When the enable signal is active, the excitation inputs are gated directly to the output Q. Thus, any change in the excitation input immediately causes a change in the latch output.
- The problem is solved by using a special timing control signal called a *clock* to restrict the times at which the states of the memory elements may change.
- This leads us to the edge-triggered memory elements called *flip-flops*.

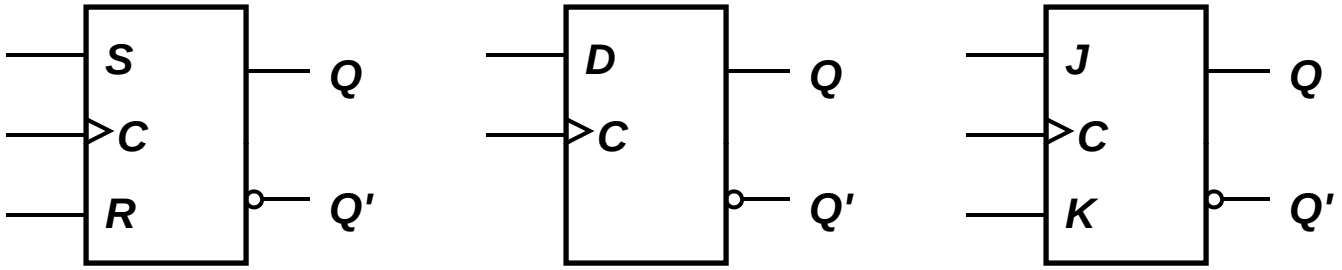
Edge-Triggered Flip-flops

- *Flip-flops*: synchronous bistable devices
- Output changes state at a specified point on a triggering input called the *clock*.
- Change state either at the *positive edge* (rising edge) or at the *negative edge* (*falling edge*) of the clock signal.

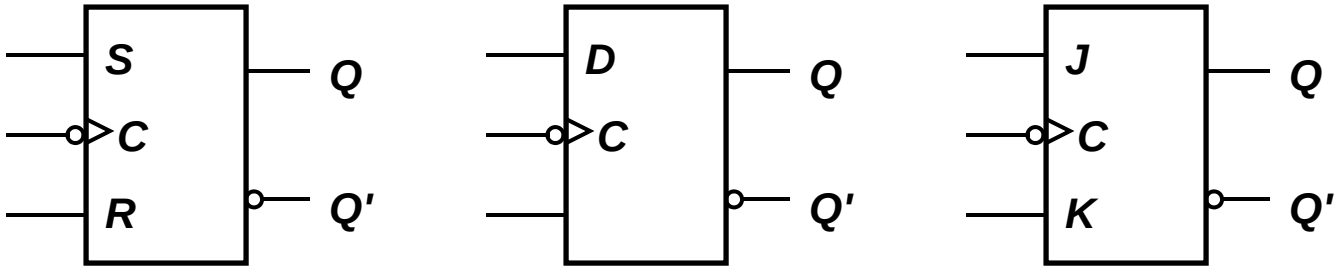


Edge-Triggered Flip-flops

- S-R, D and J-K edge-triggered flip-flops. Note the “>” symbol at the clock input.



Positive edge-triggered flip-flops



Negative edge-triggered flip-flops

S-R Flip-flop

- S-R flip-flop: on the triggering edge of the clock pulse,
  - ❖ *S*=HIGH (and *R*=LOW) ⇨ SET state
  - ❖ *R*=HIGH (and *S*=LOW) ⇨ RESET state
  - ❖ both inputs LOW ⇨ no change
  - ❖ both inputs HIGH ⇨ invalid
- Characteristic table of positive edge-triggered S-R flip-flop:

S	*R*	*CLK*	*Q(t+1)*	Comments
0	0	X	*Q(t)*	No change
0	1	↑	0	Reset
1	0	↑	1	Set
1	1	↑	?	Invalid

X = irrelevant (“don’t care”)

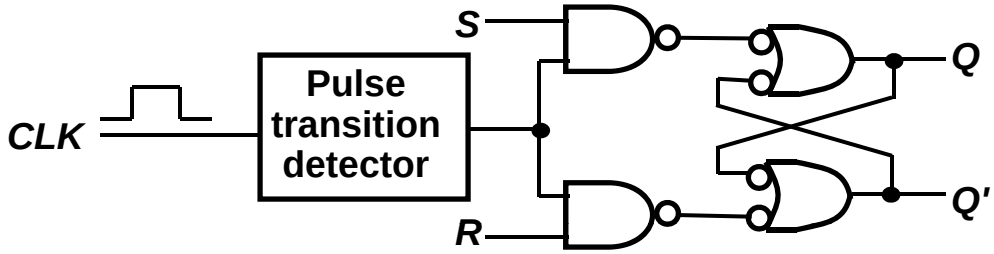
↑ = clock transition LOW to HIGH

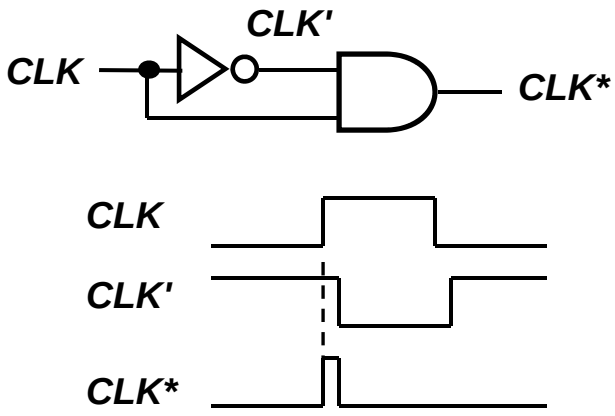
S-R Flip-flop

- It comprises 3 parts:
  - a basic *NAND latch*
  - a *pulse-steering* circuit
  - a *pulse transition detector* (or *edge detector*) circuit
- The pulse transition detector detects a rising (or falling) edge and produces a very *short-duration spike*.

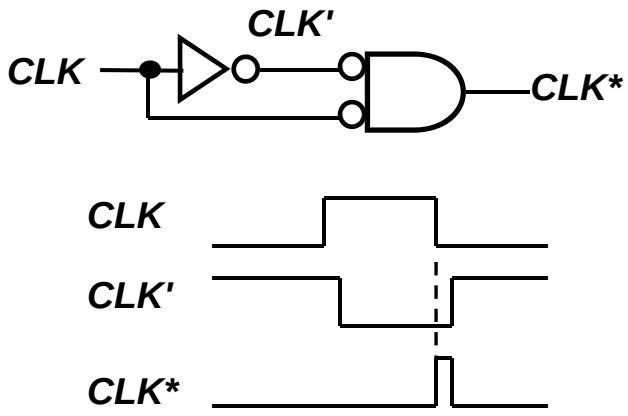
S-R Flip-flop

The pulse transition detector.





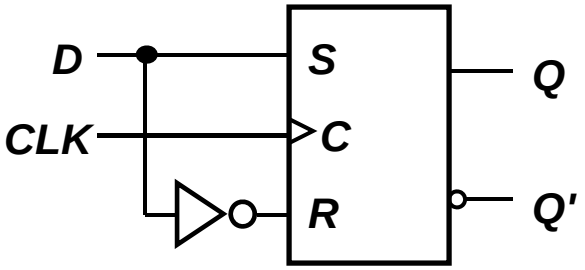
Positive-going transition (rising edge)



Negative-going transition (falling edge)

D Flip-flop

- D flip-flop: single input *D* (data)
  - *D*=HIGH ⇨ SET state
  - *D*=LOW ⇨ RESET state
- *Q* follows *D* at the clock edge.
- Convert S-R flip-flop into a D flip-flop: add an inverter.



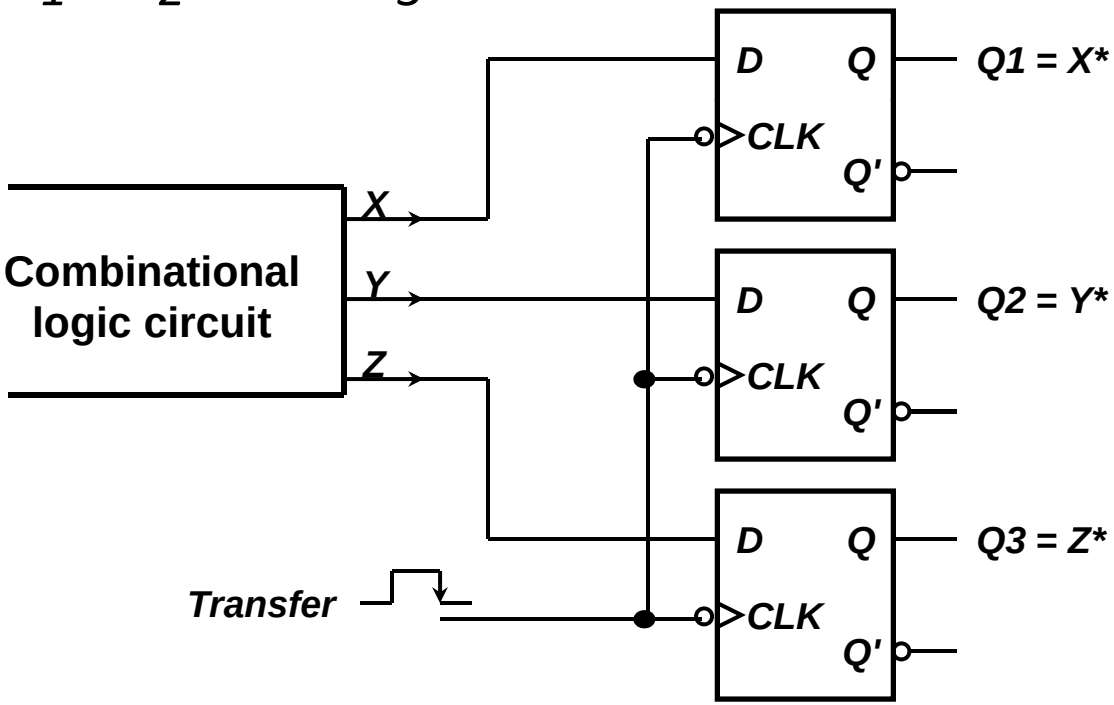
A positive edge-triggered D flip-flop formed with an S-R flip-flop.

D	*CLK*	*Q(t+1)*	Comments
1	↑	1	Set
0	↑	0	Reset

↑ = clock transition LOW to HIGH

D Flip-flop

- Application: *Parallel data transfer*.
  To transfer logic-circuit outputs *X*, *Y*, *Z* to flip-flops Q_1, Q_2 and Q_3 for storage.



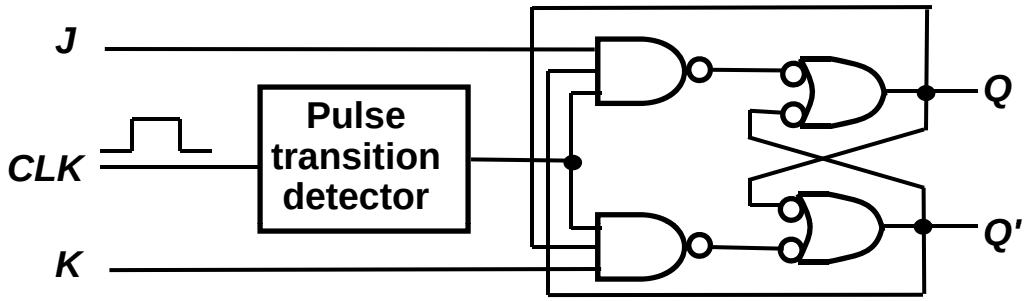
*** After occurrence of negative-going transition**

J-K Flip-flop

- J-K flip-flop: Q and Q' are fed back to the pulse-steering NAND gates.
- No invalid state.
- Include a *toggle* state.
  - *J*=HIGH (and *K*=LOW) ⇨ SET state
  - *K*=HIGH (and *J*=LOW) ⇨ RESET state
  - both inputs LOW ⇨ no change
  - both inputs HIGH ⇨ toggle

J-K Flip-flop

- J-K flip-flop.



- Characteristic table.

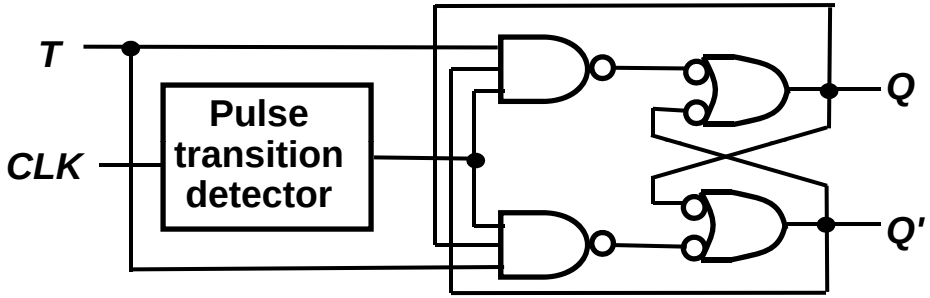
J	K	CLK	Q(t+1)	Comments
0	0	↑	Q(t)	No change
0	1	↑	0	Reset
1	0	↑	1	Set
1	1	↑	Q(t)'	Toggle

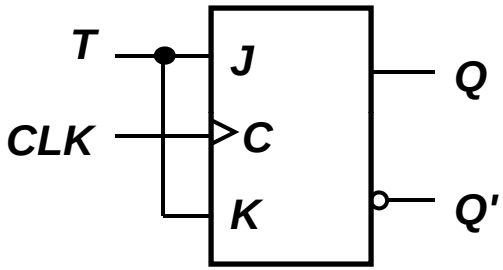
$$Q(t+1) = J.Q' + K'.Q$$

Q	J	K	Q(t+1)
0	0	0	0
0	0	1	0
0	1	0	1
0	1	1	1
1	0	0	1
1	0	1	0
1	1	0	1
1	1	1	0

T Flip-flop

- T flip-flop: single-input version of the J-K flip flop, formed by tying both inputs together.





- Characteristic table.

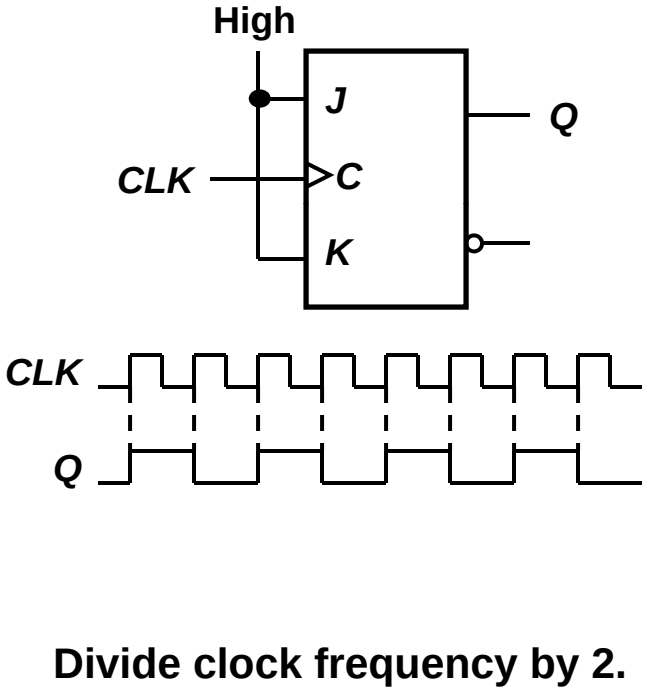
T	*CLK*	*Q(t+1)*	Comments
0	↑	*Q(t)*	No change
1	↑	*Q(t)'*	Toggle

Q	*T*	*Q(t+1)*
0	0	0
0	1	1
1	0	1
1	1	0

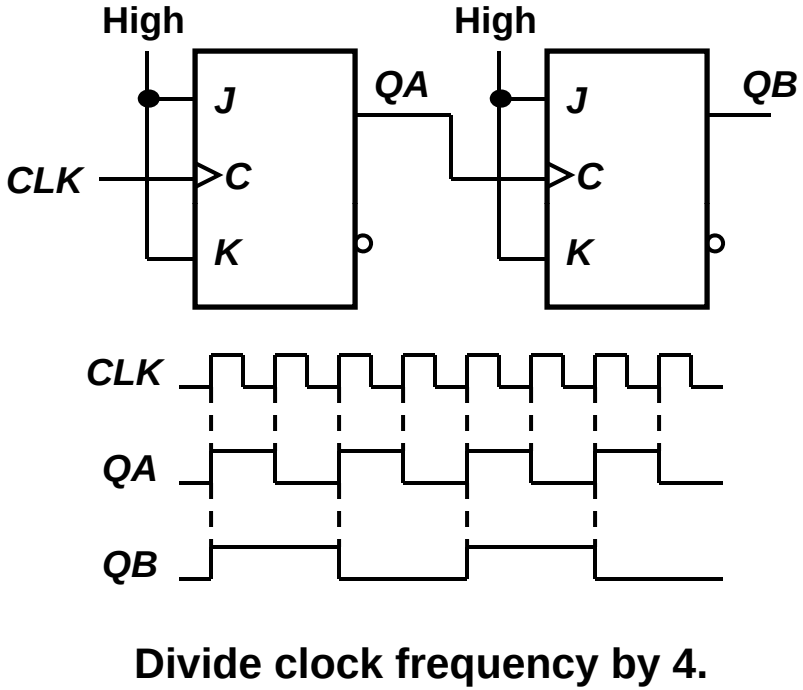
$$Q(t+1) = T.Q' + T'.Q$$

T Flip-flop

- Application: *Frequency division*.



Divide clock frequency by 2.



Divide clock frequency by 4.

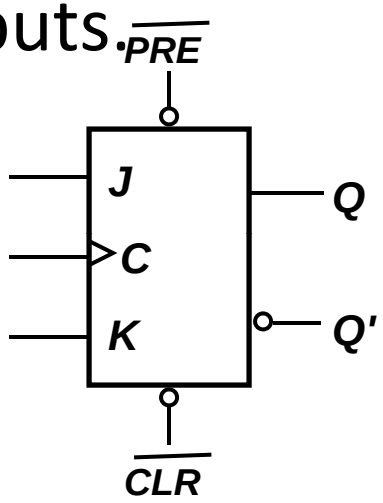
- Application: *Counter* (to be covered in Lecture 13.)

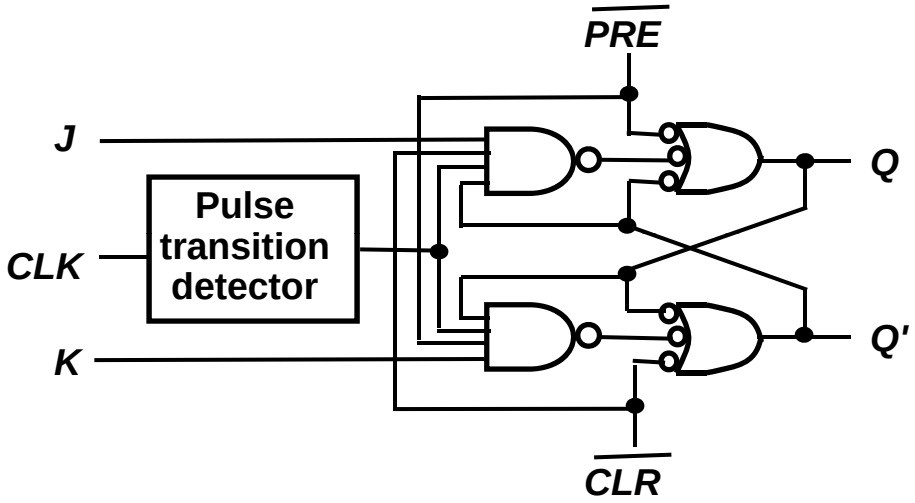
Asynchronous Inputs

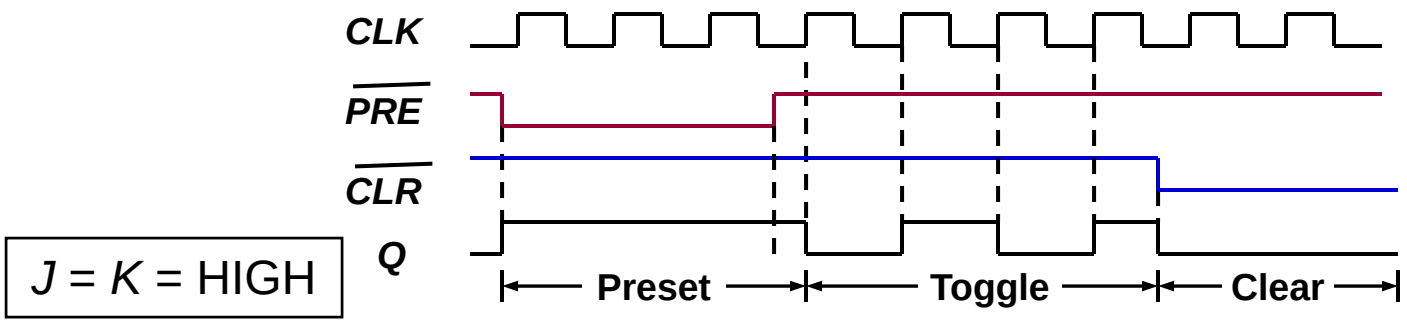
- S-R, D and J-K inputs are synchronous inputs, as data on these inputs are transferred to the flip-flop's output only on the triggered edge of the clock pulse.
- Asynchronous inputs affect the state of the flip-flop independent of the clock; example: *preset* (*PRE*) and *clear* (*CLR*) [or *direct set* (*SD*) and *direct reset* (*RD*)]
- When *PRE*=HIGH, *Q* is immediately set to HIGH.
- When *CLR*=HIGH, *Q* is immediately cleared to LOW.
- Flip-flop in normal operation mode when both *PRE* and *CLR* are LOW.

Asynchronous Inputs

- A J-K flip-flop with active-LOW preset and clear inputs.

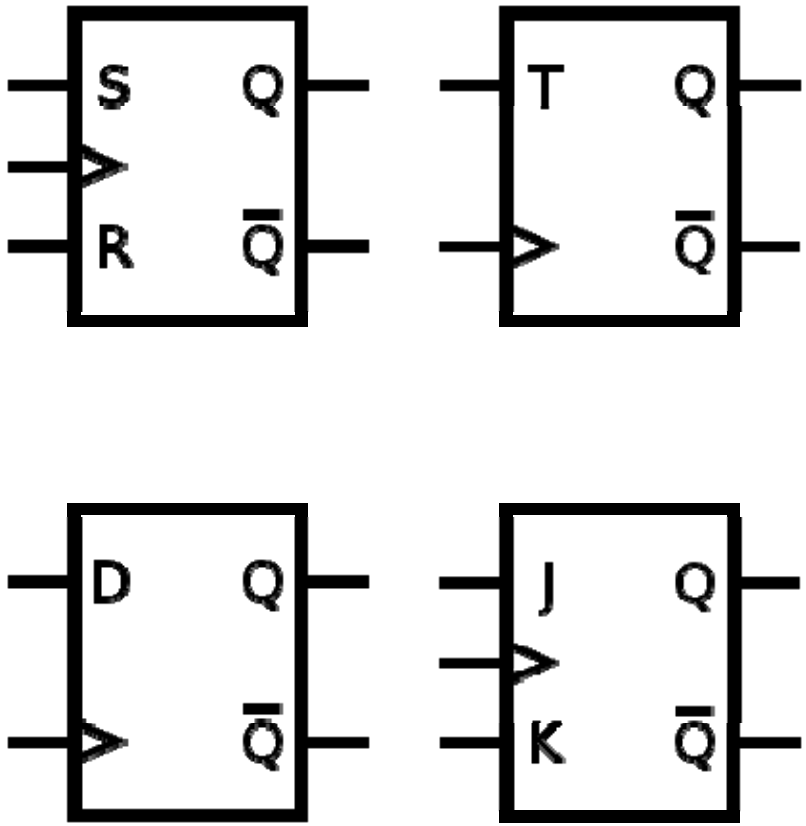






Types of Flip-flops

- SR flip-flop (Set, Reset)
- T flip-flop (Toggle)
- D flip-flop (Delay)
- JK flip-flop



Excitation Tables

Previous State -> Present State	S	R
0 -> 0	0	X
0 -> 1	1	0
1 -> 0	0	1
1 -> 1	X	0

Previous State -> Present State	T
0 -> 0	0
0 -> 1	1
1 -> 0	1
1 -> 1	0

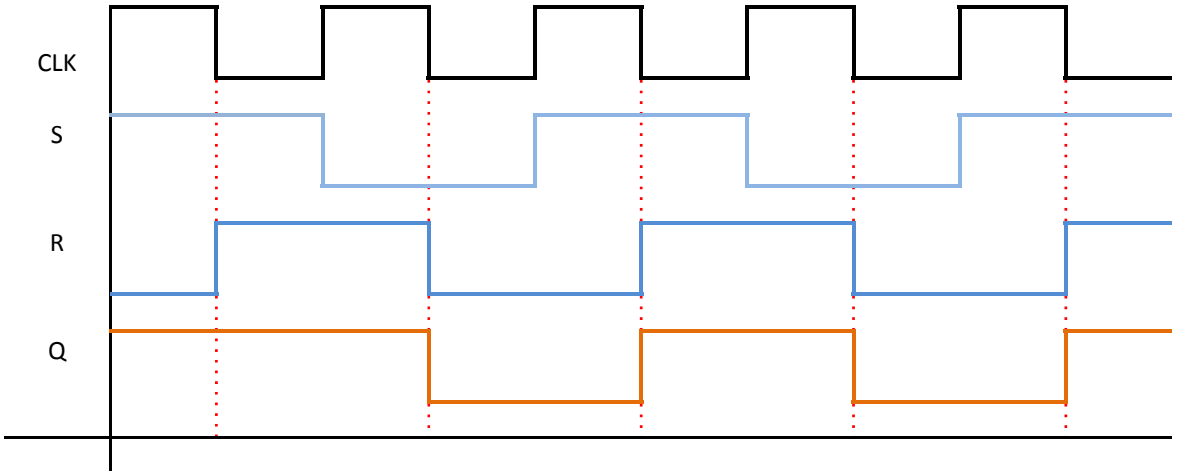
Excitation Tables

Previous State -> Present State	D
0 -> 0	0
0 -> 1	1
1 -> 0	0
1 -> 1	1

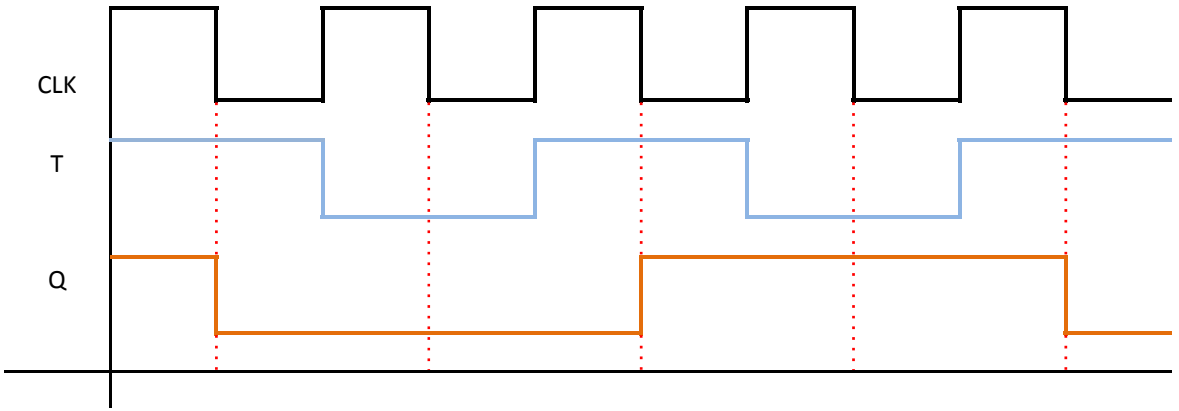
Previous State -> Present State	J	K
0 -> 0	0	X
0 -> 1	1	X
1 -> 0	X	1
1 -> 1	X	0

Timing Diagrams

	S	R
0->0	0	X
0->1	1	0
1->0	0	1
1->1	X	0

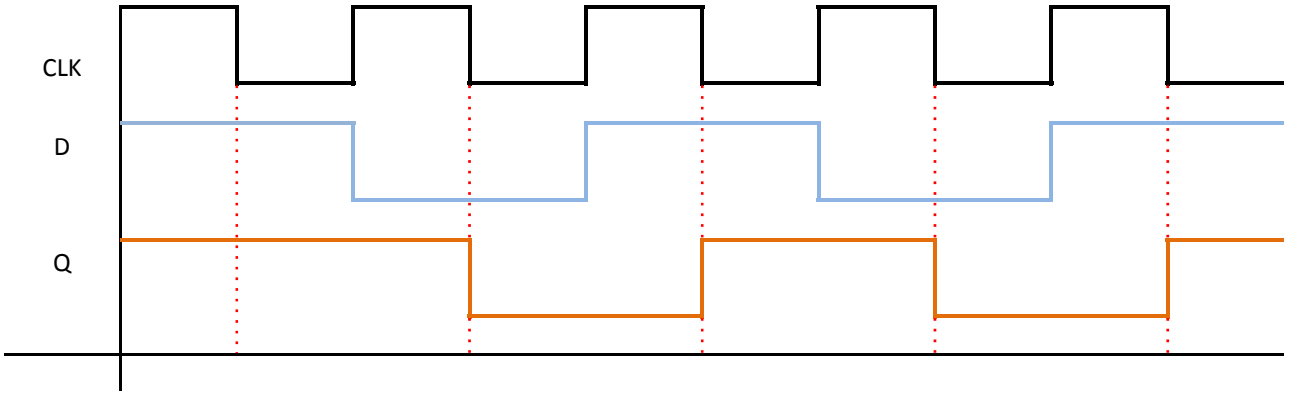


	T
0->0	0
0->1	1
1->0	1
1->1	0

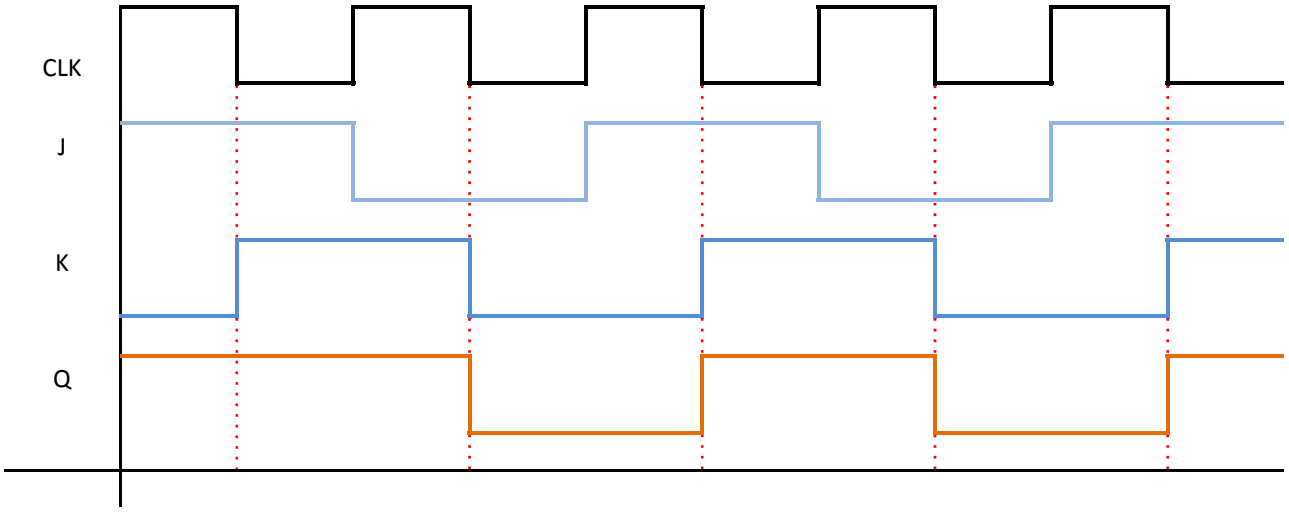


Timing Diagrams

	D
0->0	0
0->1	1
1->0	0
1->1	1



	J	K
0->0	0	X
0->1	1	X
1->0	X	1
1->1	X	0



Converting Flip-flops

- Use T flip-flop to implement D flip-flop

D	Q+
0	0
1	1

D\Q+	0	1
0	0	0
1	1	1

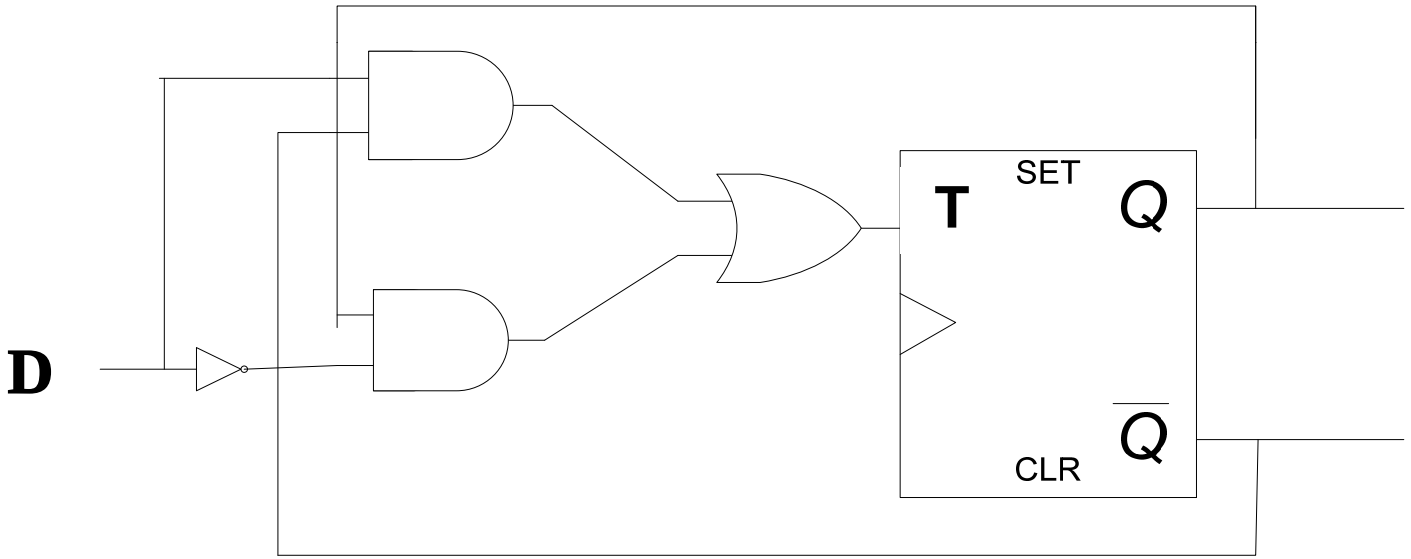
D\Q+	0	1
0	0	1
1	1	0

T	Q+
0	Q
1	Q’

$$\mathbf{T = DQ' + D'Q}$$

Converting flip-flops

- Use T flip-flop to implement D flip-flop



Converting Flip-flops

- Use T flip-flop to implement JK flip-flop

T	Q+
0	Q
1	Q’

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

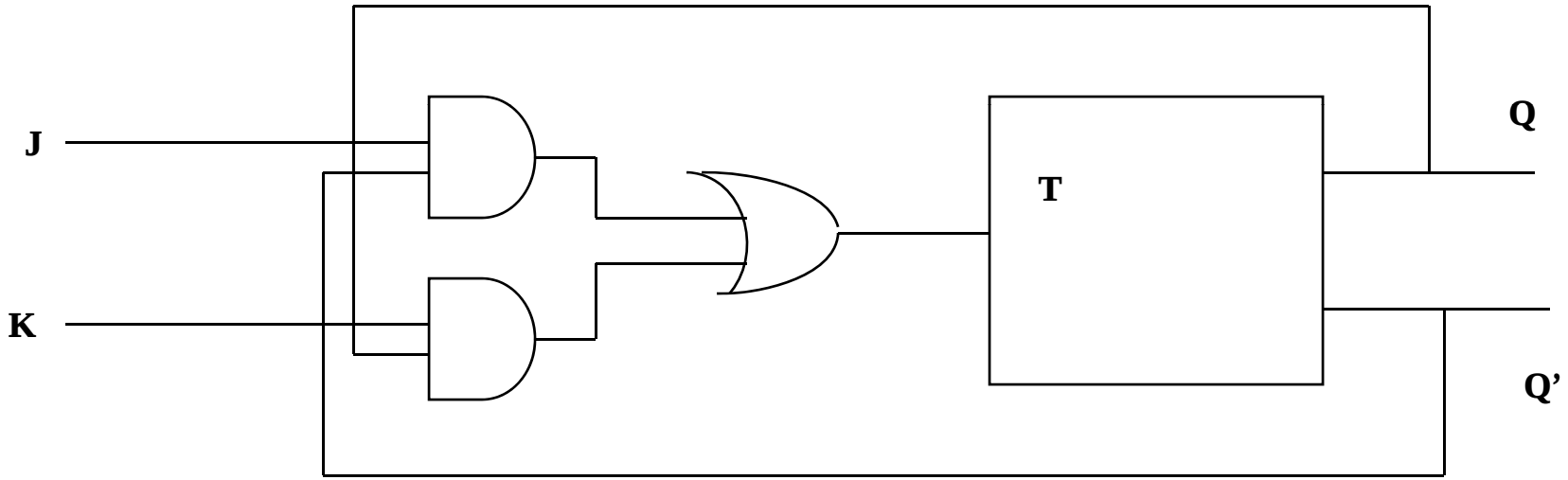
JK\Q+	0	1
00	0	0
01	0	1
11	1	1
10	1	0

J	K	Q+
0	0	Q
0	1	0
1	0	1
1	1	Q’

$$\mathbf{T = JQ' + KQ}$$

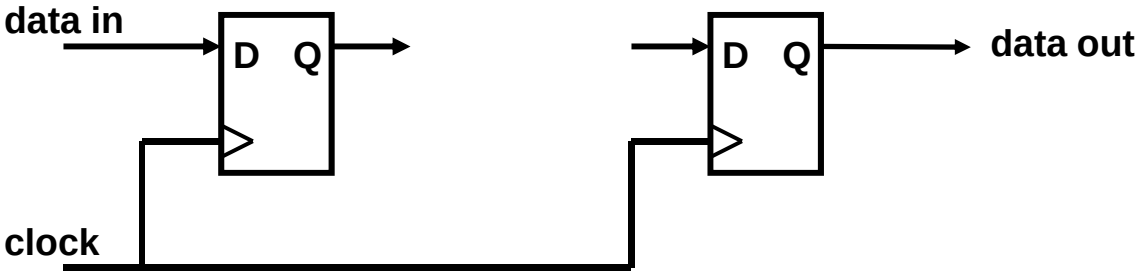
Converting flip-flops

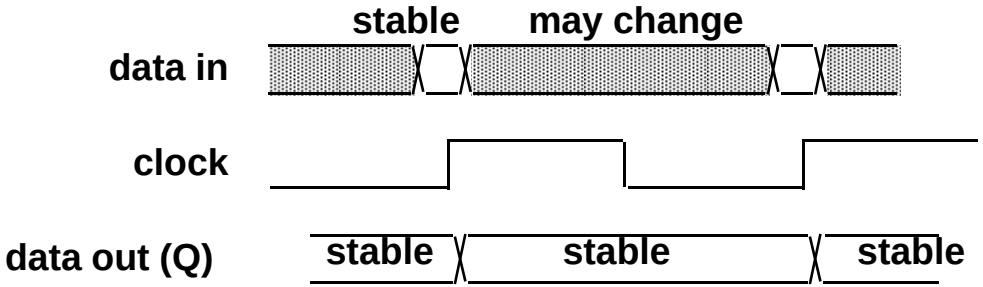
- Use T flip-flop to implement JK flip-flop



Registers

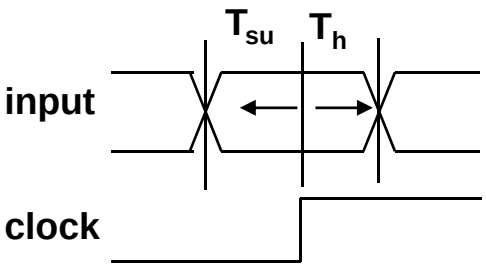
- Sample data using clock
- Hold data between clock cycles
- Computation (and delay) occurs between registers

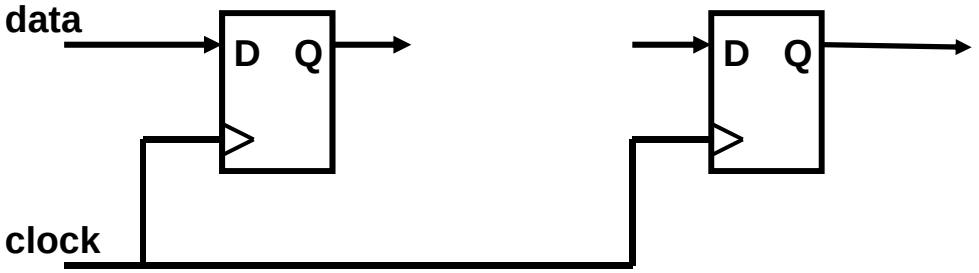




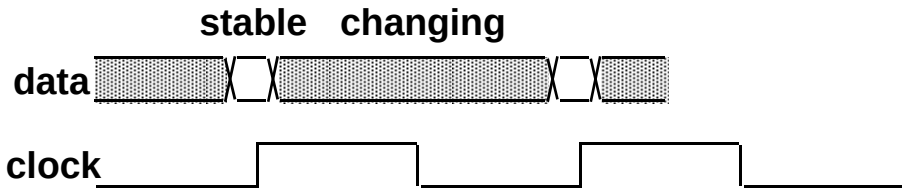
Timing Methodologies (cont'd)

- Definition of terms
  - setup time: minimum time before the clocking event by which the input must be stable (T_{su})
  - hold time: minimum time after the clocking event until which the input must remain stable (T_h)



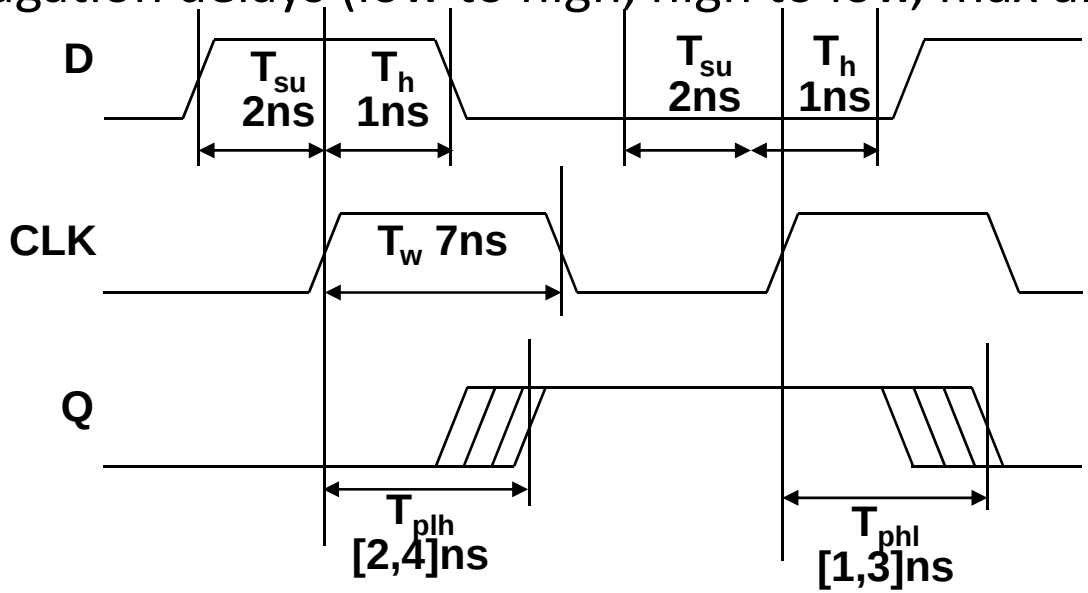


there is a timing "window" around the clocking event during which the input must remain stable and unchanged in order to be recognized



Typical timing specifications

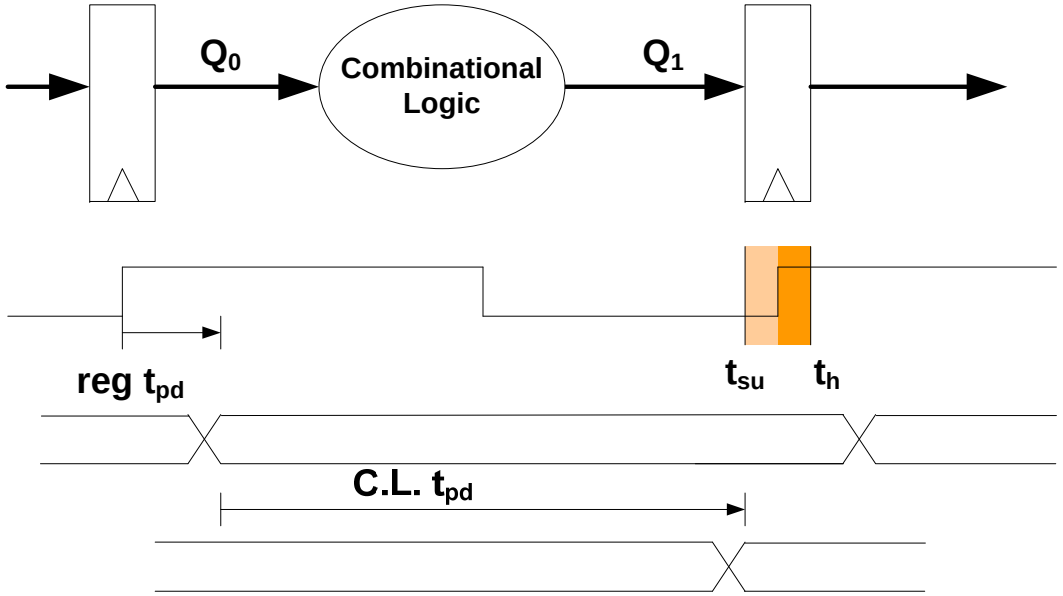
- Positive edge-triggered D flip-flop
  - setup and hold times
  - minimum clock width
  - propagation delays (low to high, high to low, max and typical)



all measurements are made from the clocking event that is, the rising edge of the clock

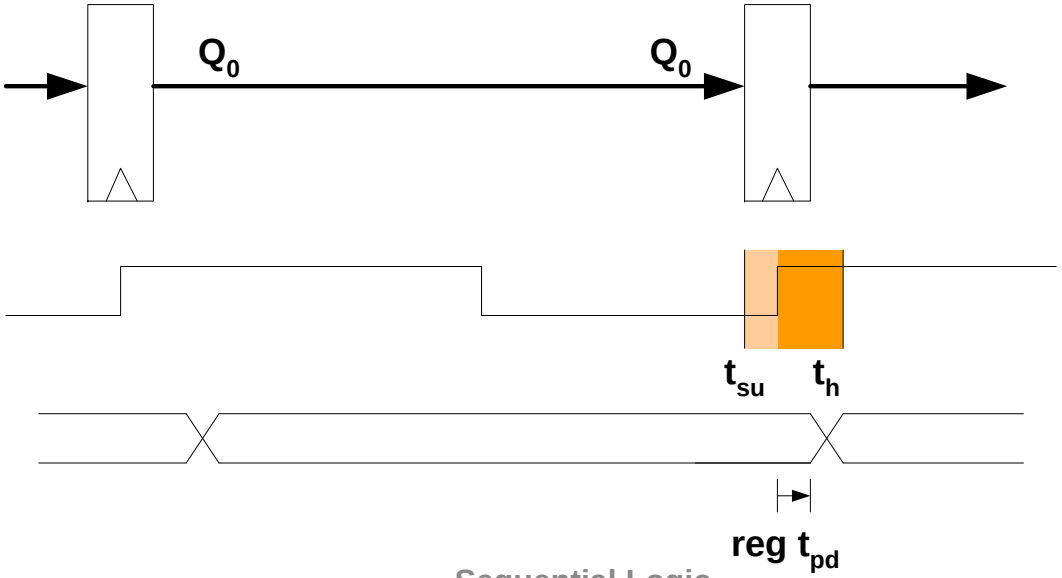
System Clock Frequency

- Register transfer must fit into one clock cycle
  - reg t_{pd} + C.L. t_{pd} + reg t_{su} < T_{clk}
  - Use maximum delays
  - Find the “critical path”
    - Longest register-register delay



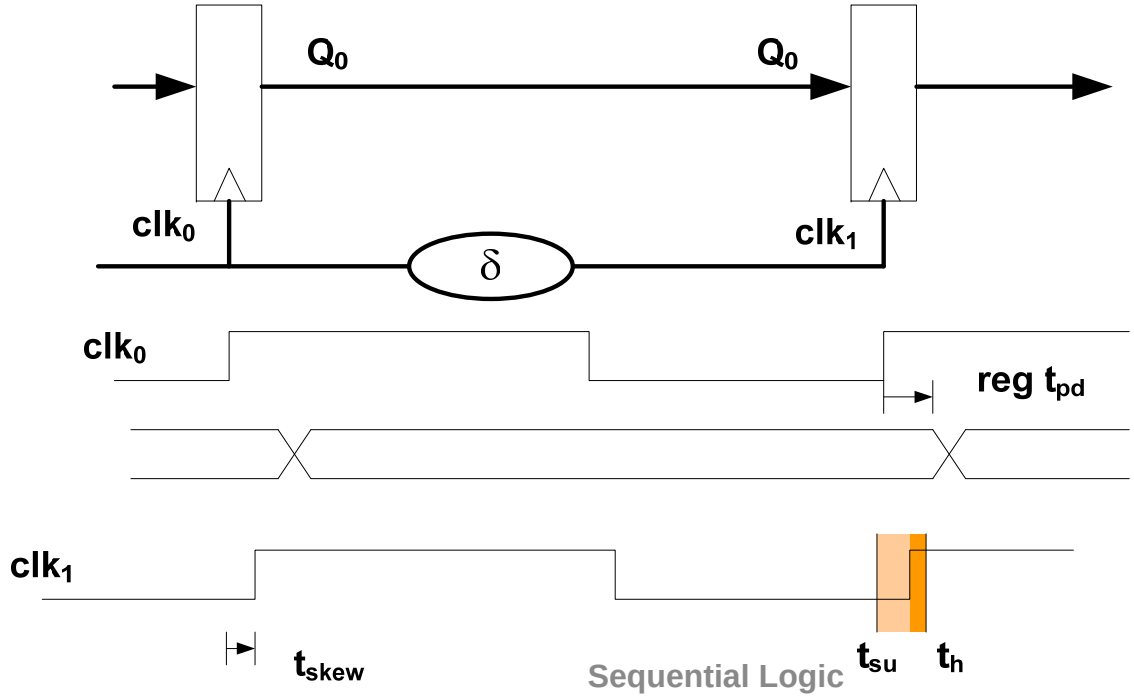
Short Paths

- Can a path have too little delay?
  - Yes: Hold time can be violated
  - Unless $t_{pd} > t_h$
  - Use min delay (contamination delay)
- Fortunately, most registers have hold time = 0
  - But there can still be a problem! Clock skew…



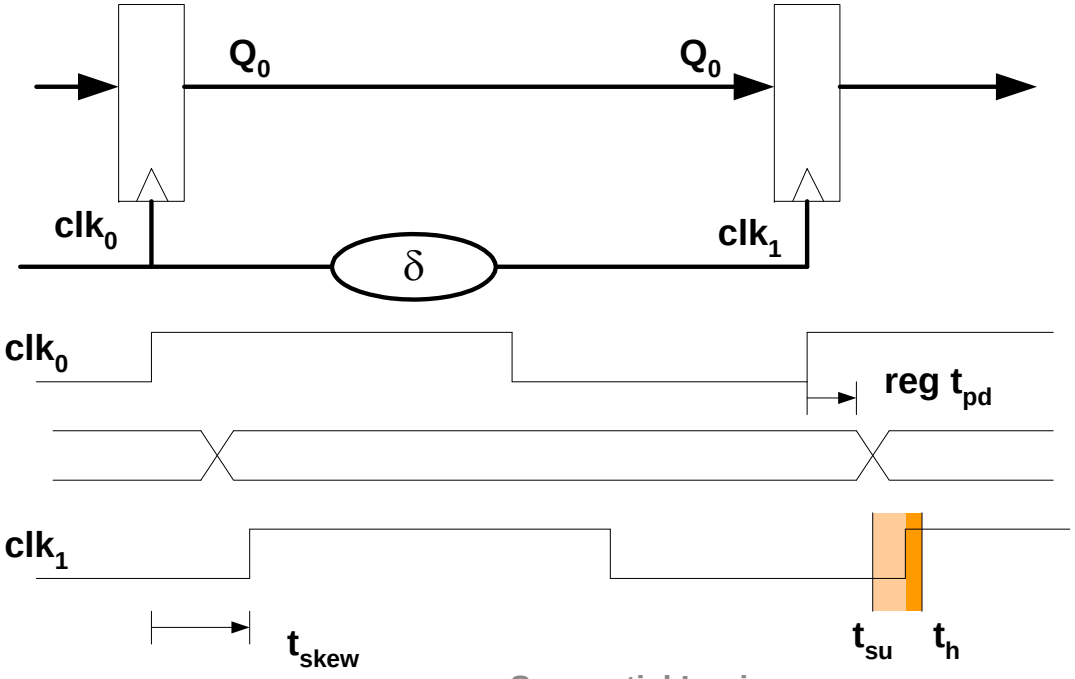
Clock Skew

- Cannot make clock arrive at registers at the same time
- If skew > 0:
  - $t_{pd} > t_h + t_{skew}$
  - Clock skew can cause system failure
    - Can you fix this after you've fabbed the chip?



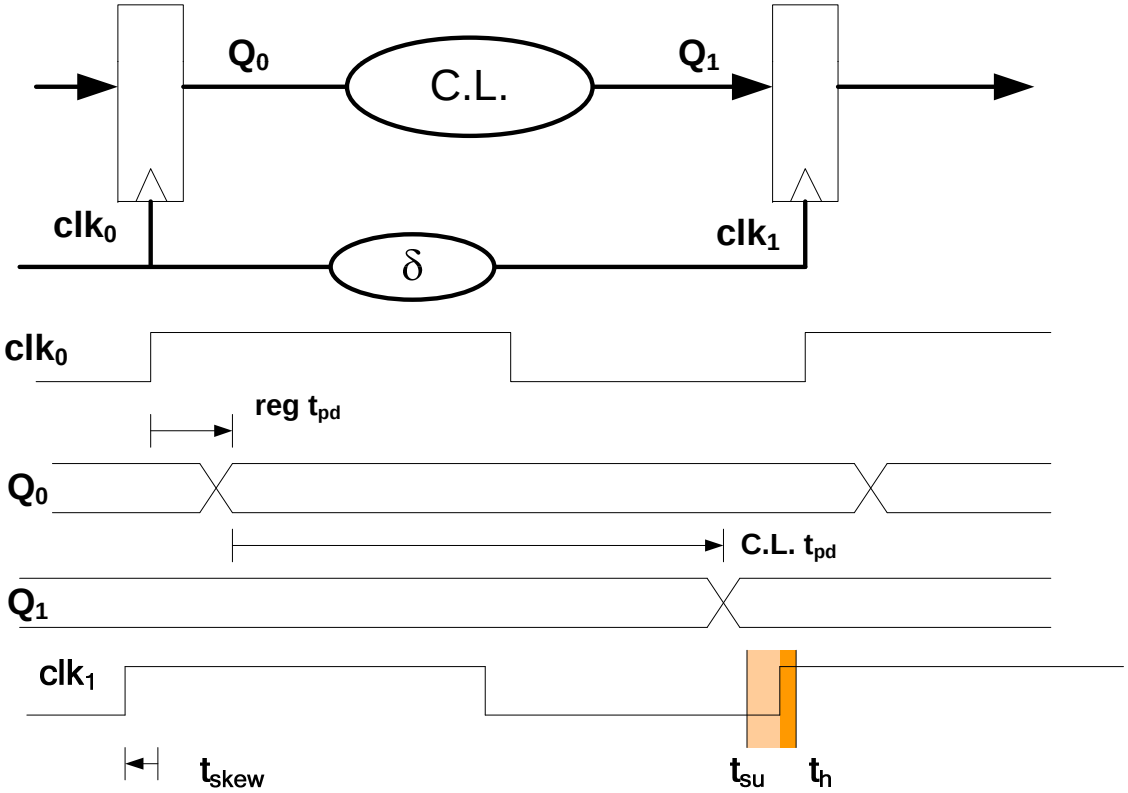
Clock Skew

- Cannot make clock arrive at registers at the same time
- If skew > 0:
  - $t_{pd} > t_h + t_{skew}$
  - Clock skew can cause system failure
    - Can you fix this after you've fabbed the chip?



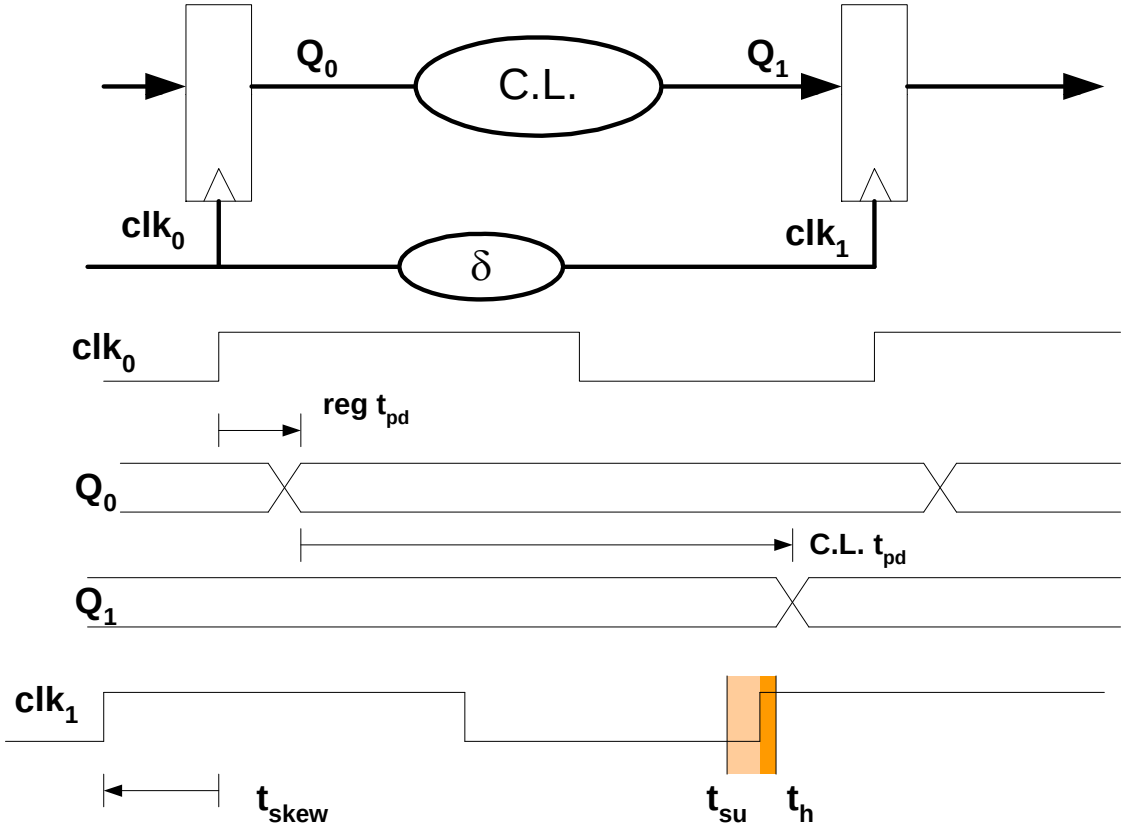
Clock Skew

- If skew < 0:
  - $t_{clk} > reg\ t_{pd} + CL\ t_{pd} + reg\ t_{SU} + |t_{skew}|$
  - Can you fix this after fab?



Clock Skew

- If skew < 0:
  - $t_{clk} > reg\ t_{pd} + CL\ t_{pd} + reg\ t_{SU} + |t_{skew}|$
  - Can you fix this after fab?

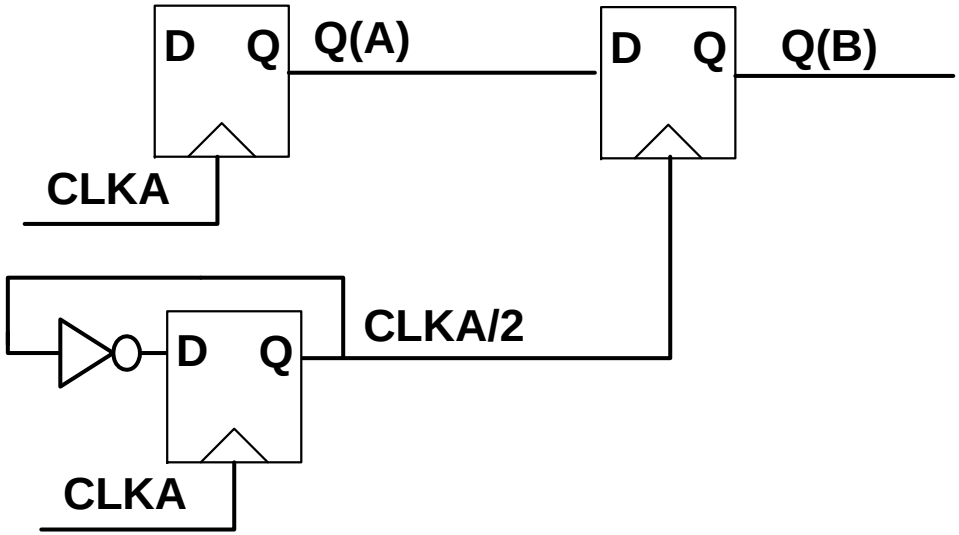


Clock Skew

- Correct behavior assumes that all storage elements sample at exactly the same time
- Not possible in real systems:
  - clock driven from some central location
  - different wire delay to different points in the circuit
- Problems arise if skew is of the same order as FF contamination delay
- Gets worse as systems get faster (wires don't improve as fast)
  - 1) distribute clock signals against the data flow
  - 2) wire carrying the clock between two communicating components should be as short as possible
  - 3) try to make all wires from the clock generator be the same length => clock tree

Nasty Example

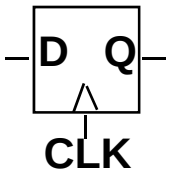
- What can go wrong?
- How can you fix it?



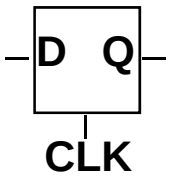
Other Types of Latches and Flip-Flops

- D-FF is ubiquitous
  - simplest design technique, minimizes number of wires
    preferred in PLDs and FPGAs
    good choice for data storage register
    edge-triggered has most straightforward timing constraints
- Historically J-K FF was popular
    versatile building block, often requires less total logic
    two inputs require more wiring and logic
    can always be implemented using D-FF
- Level-sensitive latches in special circumstances
    popular in VLSI because they can be made very small (4 T)
    fundamental building block of all other flip-flop types
      two latches make a D-FF
- Preset and clear inputs are highly desirable
  - System reset

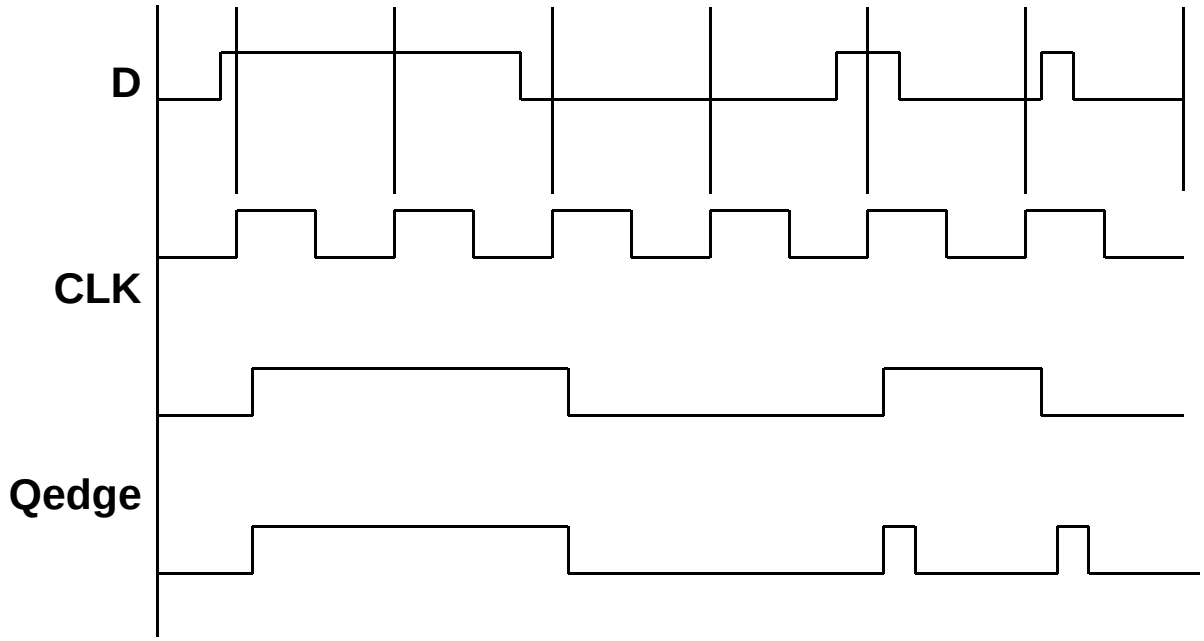
Comparison of latches and flip-flops



positive edge-triggered flip-flop



transparent, flow-through (level-sensitive) latch

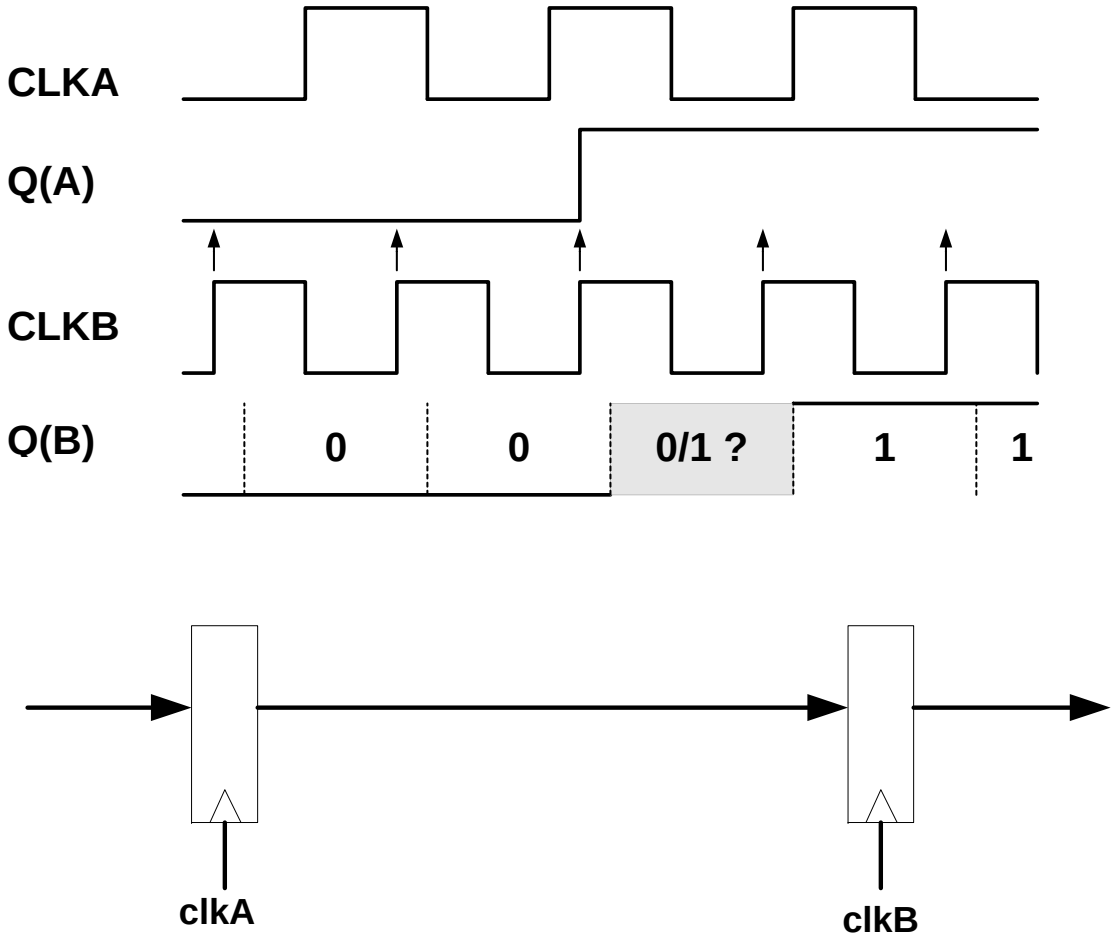


behavior is the same unless input changes while the clock is high

What About External Inputs?

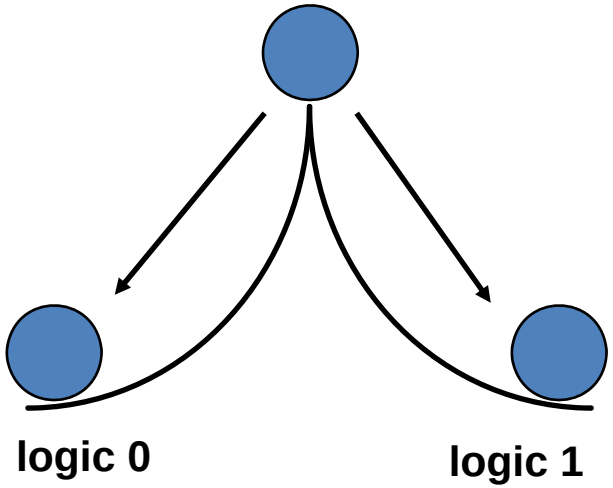
- Internal signals are OK
  - Can only change when clock changes
- External signals can change at any time
  - Asynchronous inputs
  - Truly asynchronous
  - Produced by a different clock
- This means register may sample a signal that is changing
  - Violates setup/hold time
  - What happens?

Sampling external inputs

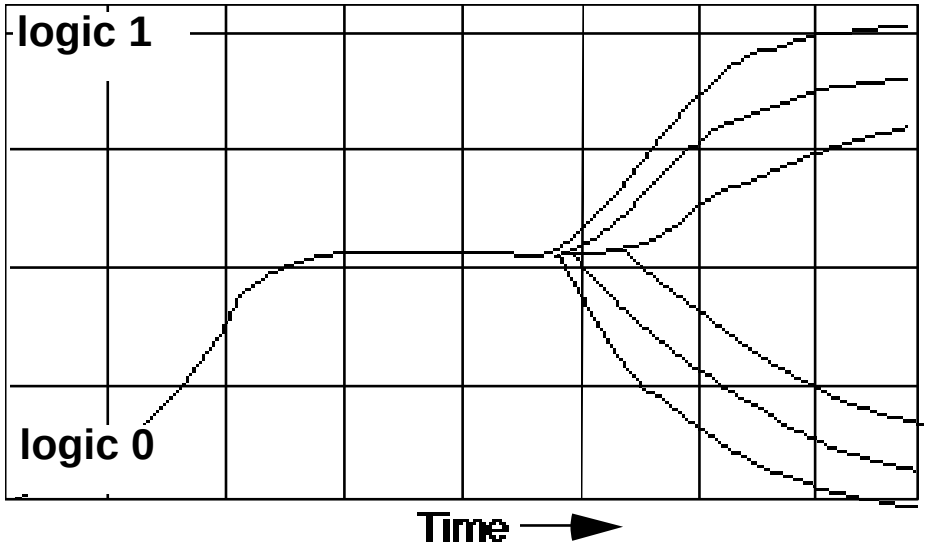


Synchronization failure

- Occurs when FF input changes close to clock edge
  - the FF may enter a metastable state – neither a logic 0 nor 1 –
  - it may stay in this state an indefinite amount of time
  - this is not likely in practice but has some probability



small, but non-zero probability that the FF output will get stuck in an in-between state



oscilloscope traces demonstrating synchronizer failure and eventual decay to steady state

Calculating probability of failure

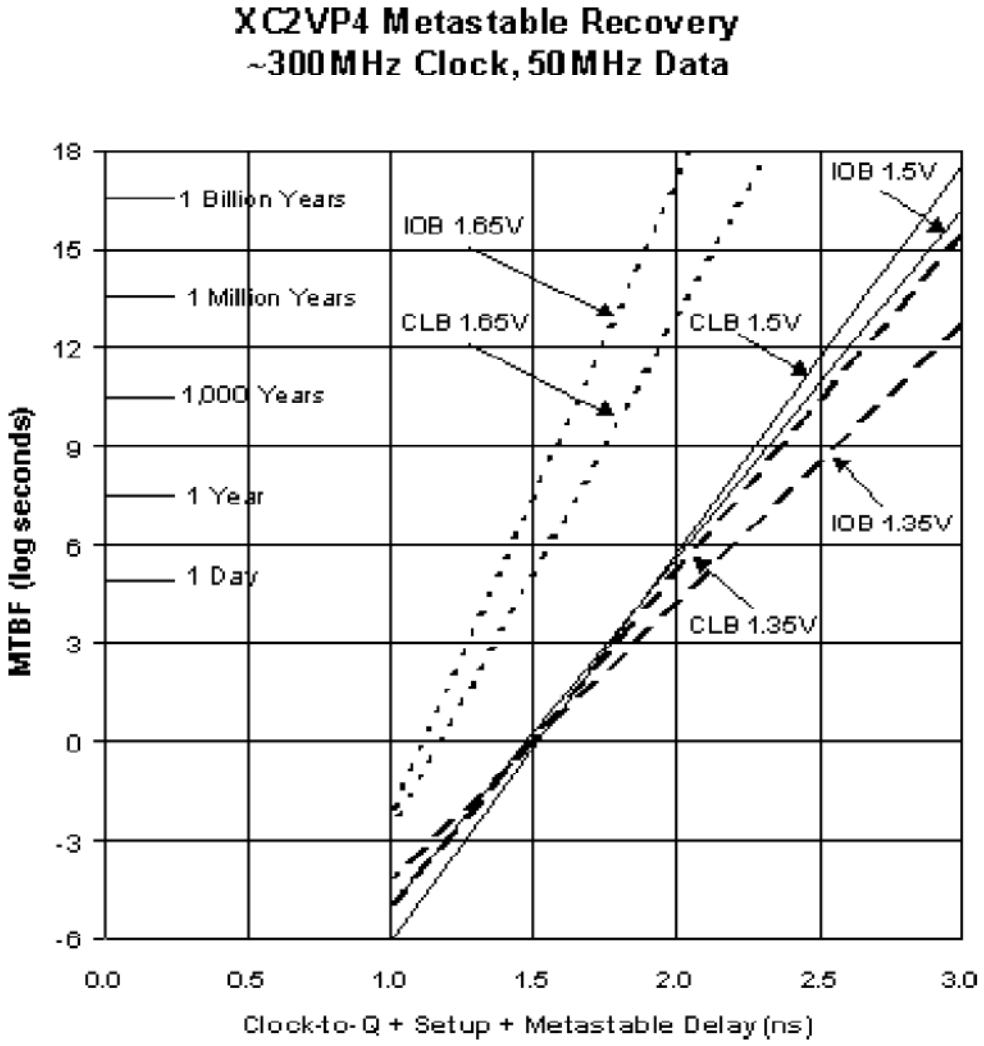
- For a single synchronizer

$$\text{Mean-Time Between Failure (MTBF)} = \exp(t_r / \tau) / (T0 \times f_c \times f_a)$$

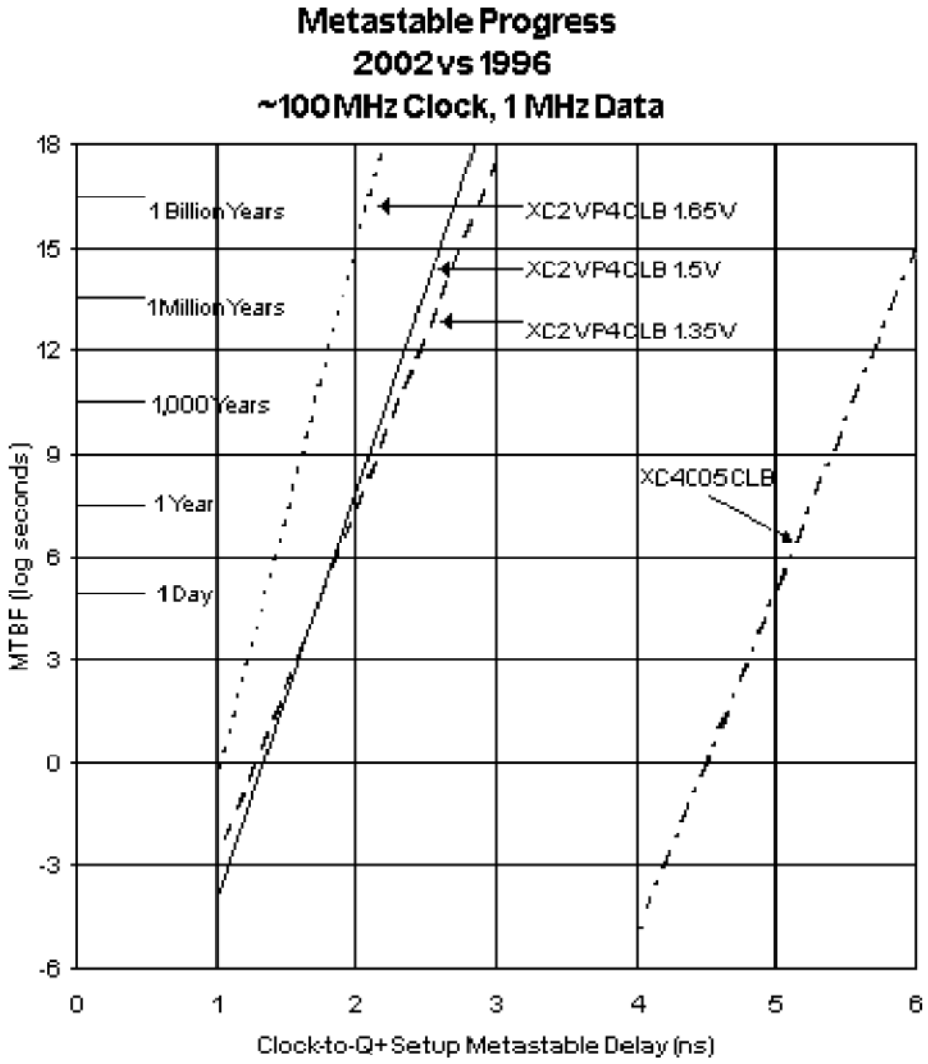
  where a failure occurs if metastability persists beyond time tr
- t_r is the resolution time - extra time in clock period for settling
  - Tclk - (t_{pd} + T_{CL} + t_{setup})
- f_c is the frequency of the FF clock
- f_a is the number of asynchronous input changes per second applied to the FF
- T0 and τ are constaints that depend on the FF's electrical characteristics (e.g., gain or steepness of curve)
  - example values are T0 = 1ms and τ = 30ps (sensitive to temperature, voltage, cosmic rays, etc.).
- Must add probabilities from all synchronizers in system

  $1/\text{MTBFsystem} = \Sigma\ 1/\text{MTBFsynch}$

Xilinx Measurements

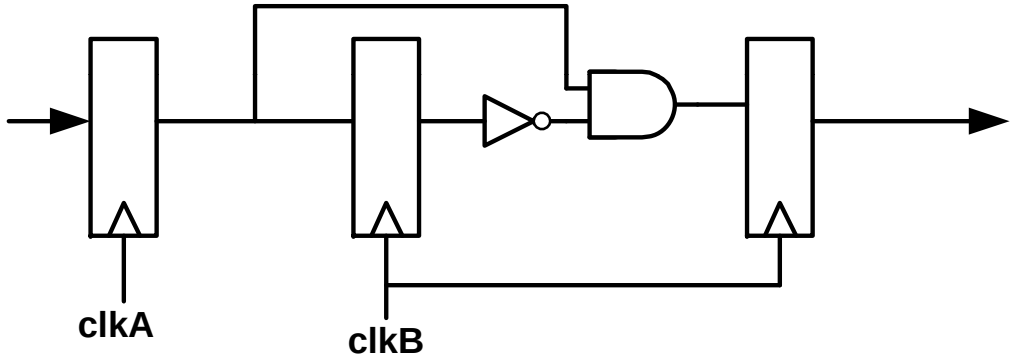


Xilinx Measurements



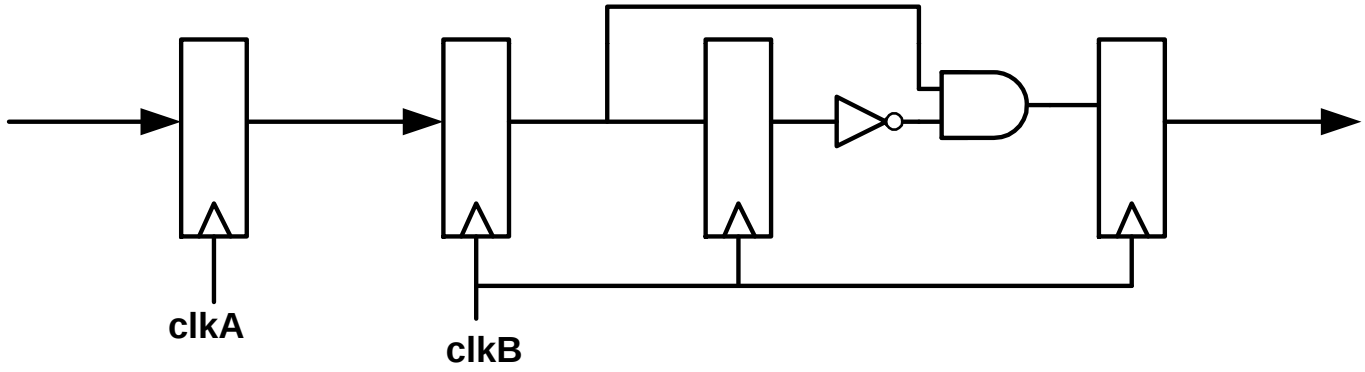
What does this circuit do?

- What's wrong with this?



What does this circuit do?

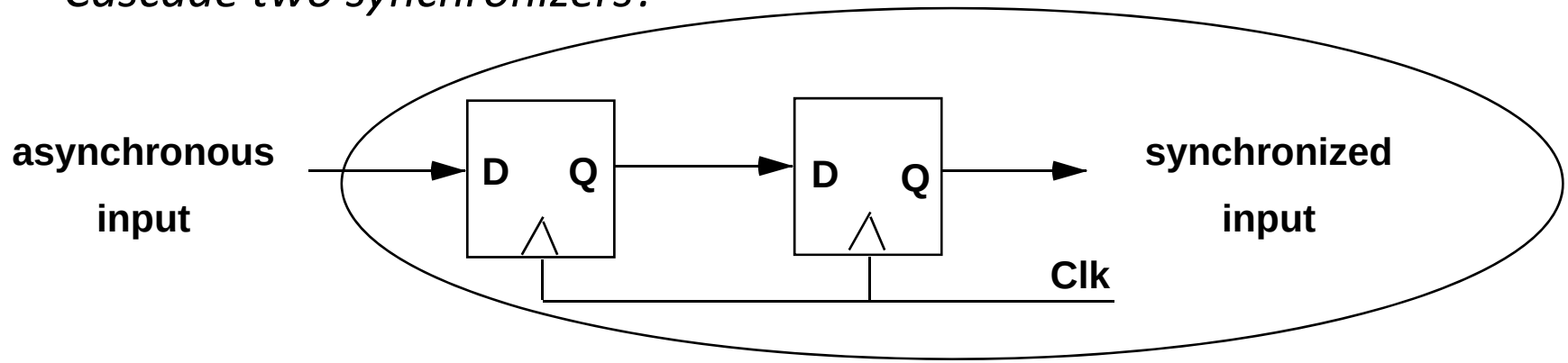
- How much better is this?



- Can you do better?

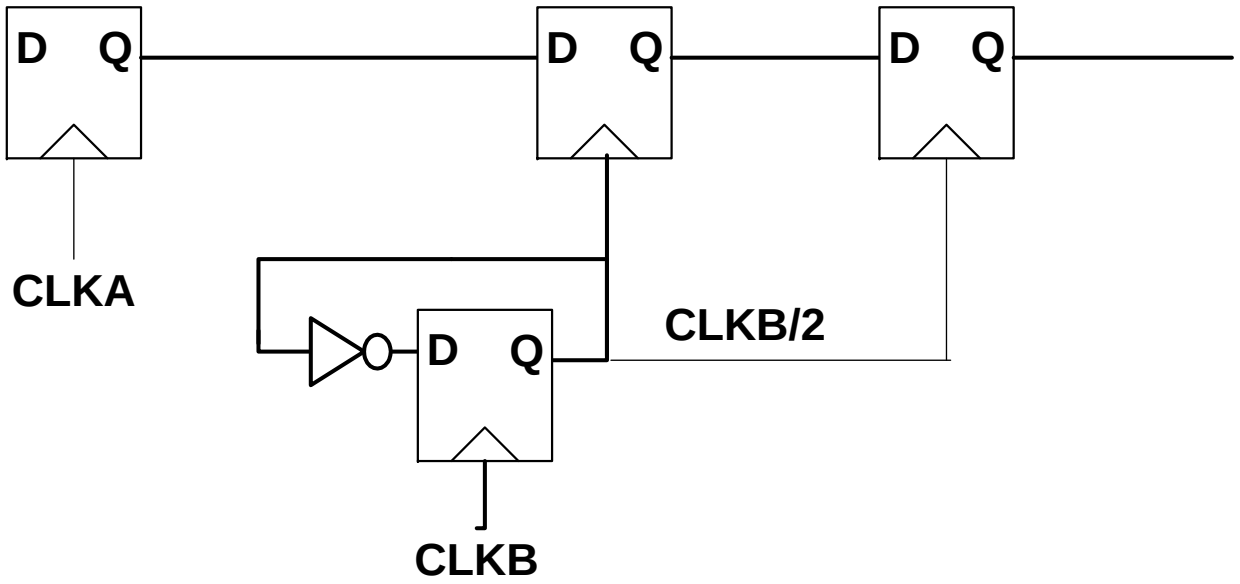
Guarding against synchronization failure

- Give the register time to decide
  - Probability of failure cannot be reduced to 0, but it can be reduced
- *Slow down the system clock?*
- *Use very fast technology for synchronizer -> quicker decision?*
- *Cascade two synchronizers?*



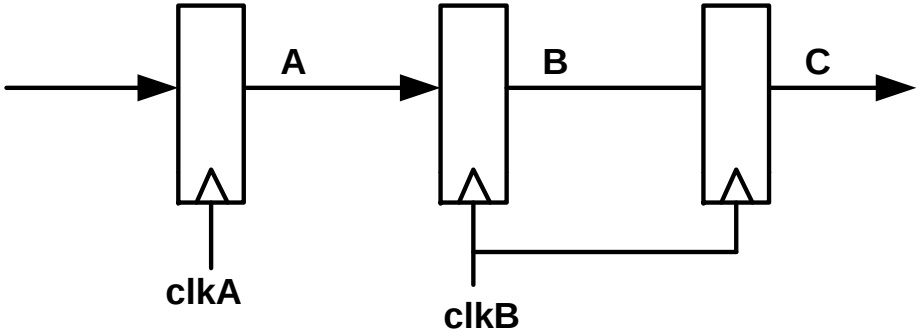
Stretching the Resolution Time

- Also slows the sample rate and transfer rate



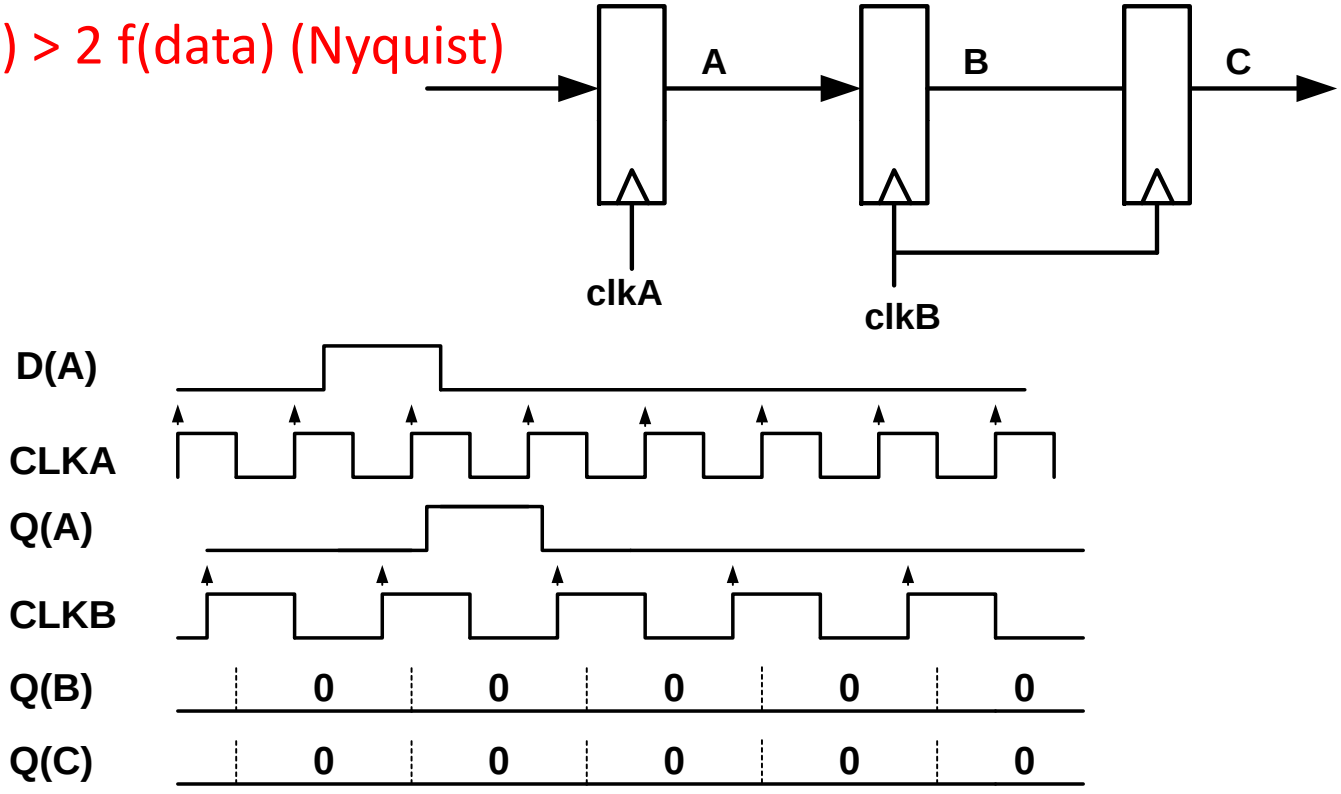
Sampling Rate

- How fast does your sample clock need to be?



Sampling Rate

- How fast does your sample clock need to be?
  - f(clkB) > f(clkA)
  - f(clkB) > 2 f(data) (Nyquist)

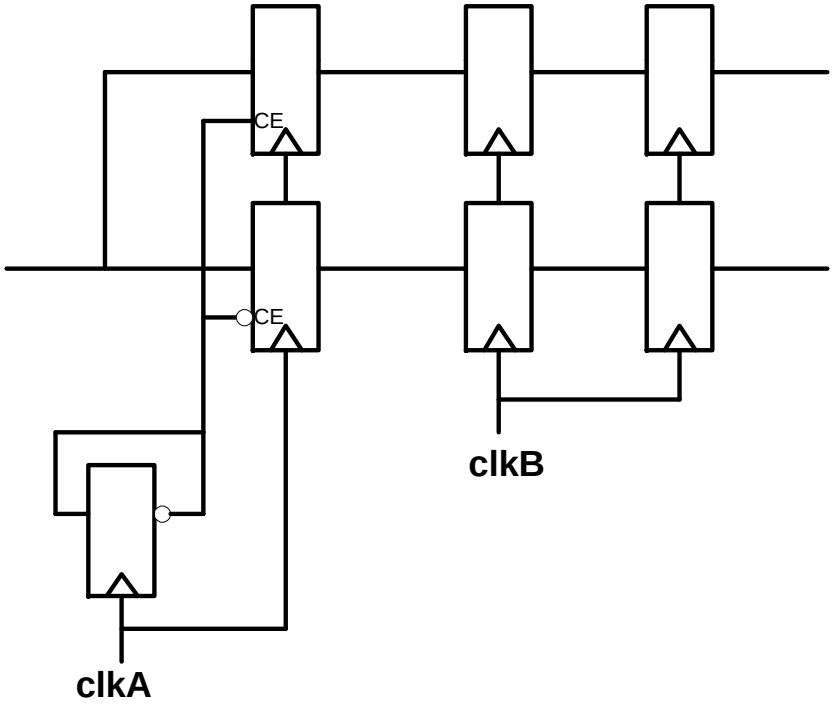


Sampling Rate

- What if sample clock can’t go faster?
- If input clock is not available, no solution(?)
- If input clock is available (e.g. video codec)

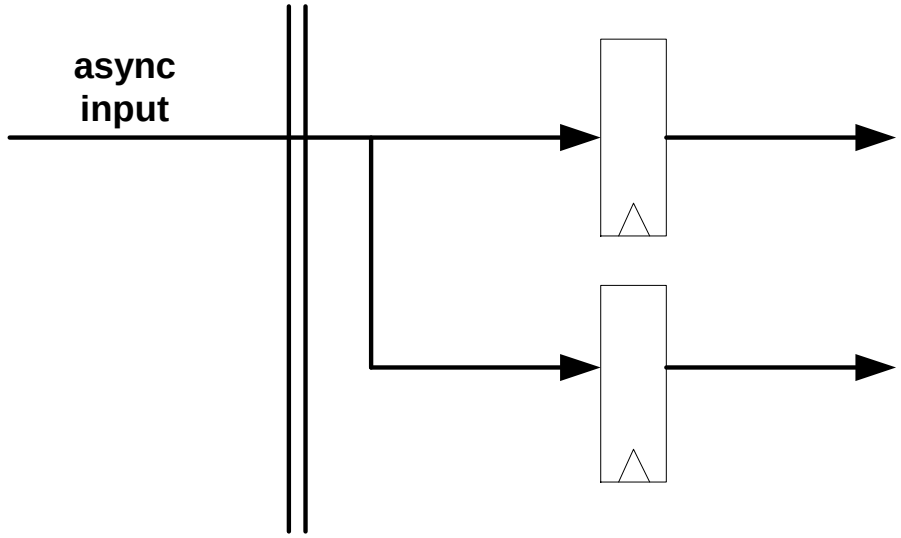
Increasing sample rate

- The problem is the **relative** sample rate
  - Slow down the input clock!



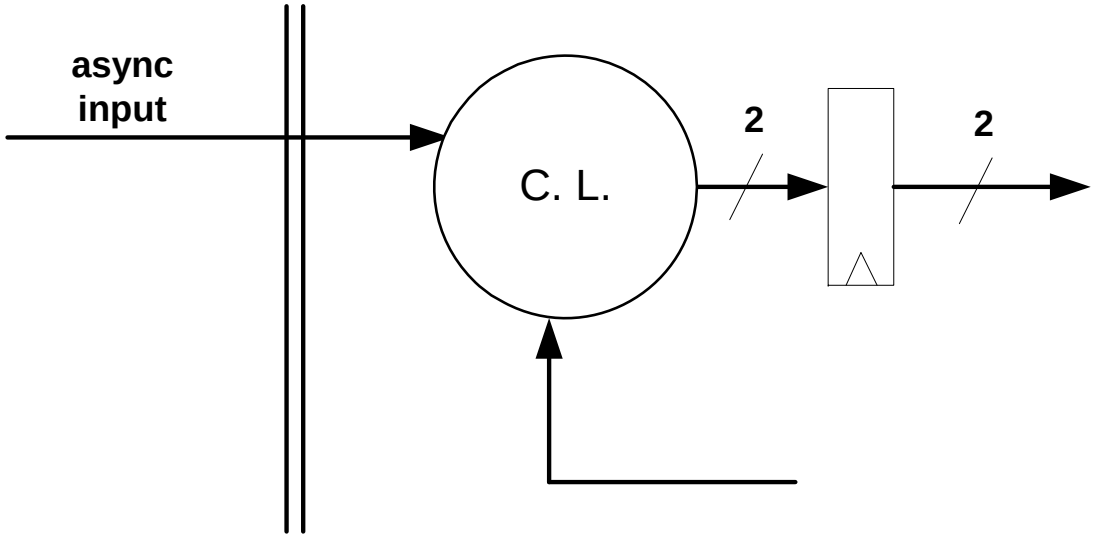
Another Problem with Asynchronous inputs

- What goes wrong here? (Hint: it's not a metastability thing)
- What is the fix?



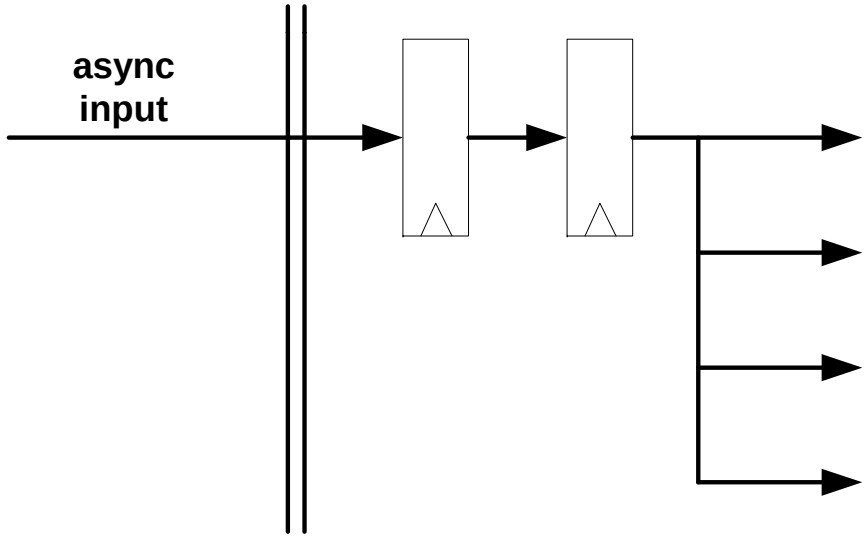
More Asynchronous inputs

- What is the problem?
- What is the fix?



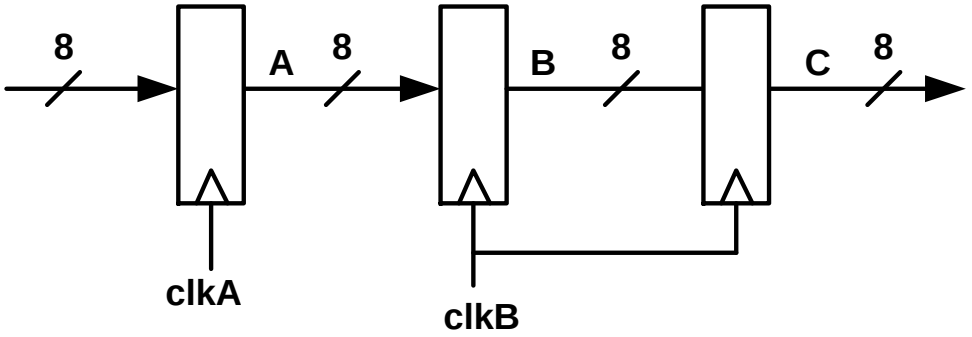
Important Rule!

- Exactly one register makes the synchronizing decision



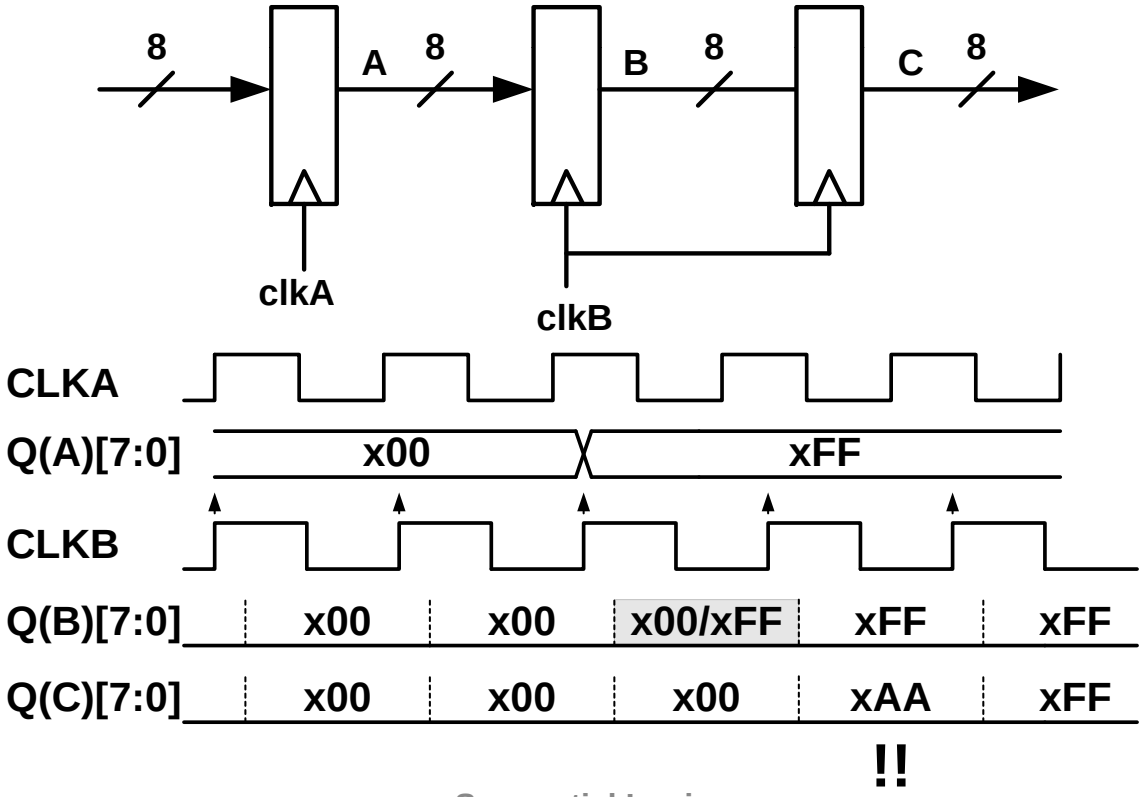
More Asynchronous inputs

- Can we input asynchronous data values with several bits?



More Asynchronous inputs

- How can we input asynchronous data values with several bits?

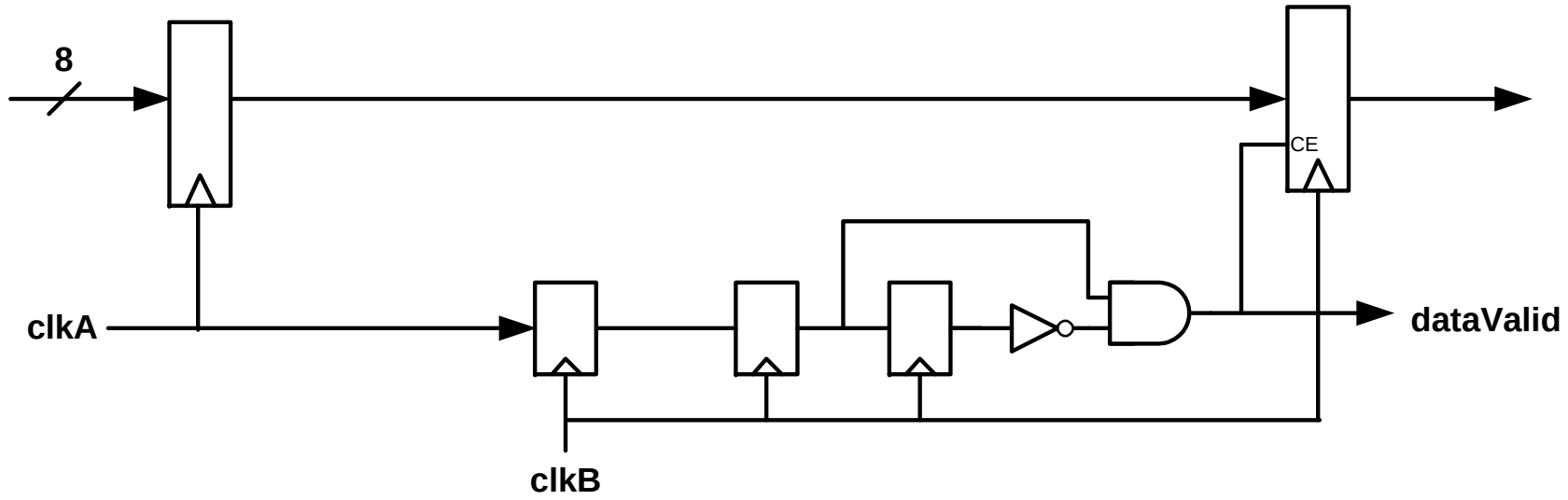


What Went Wrong?

- Each bit has a different delay
  - Wire lengths differ
  - Gate thresholds differ
  - Driver speeds are different
  - Register delays are different
    - Rise vs. Fall times
  - Clock skews to register bits
- Bottom line – “data skew” is inevitable
  - aka Bus Skew
  - Longer wires => More skew
- What is the solution??

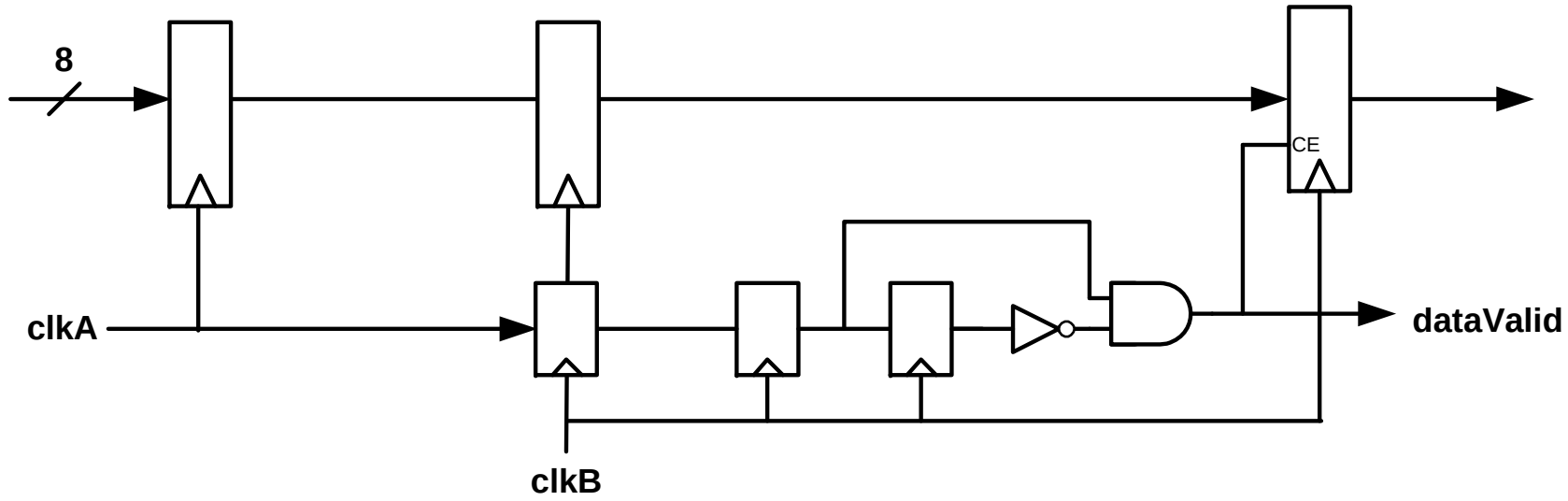
Sending Multiple Data Bits

- Must send a “clock” with the data
  - Waits until data is stable
  - De-skewing delay
- **f(clkB) > 2 f(clkA)**



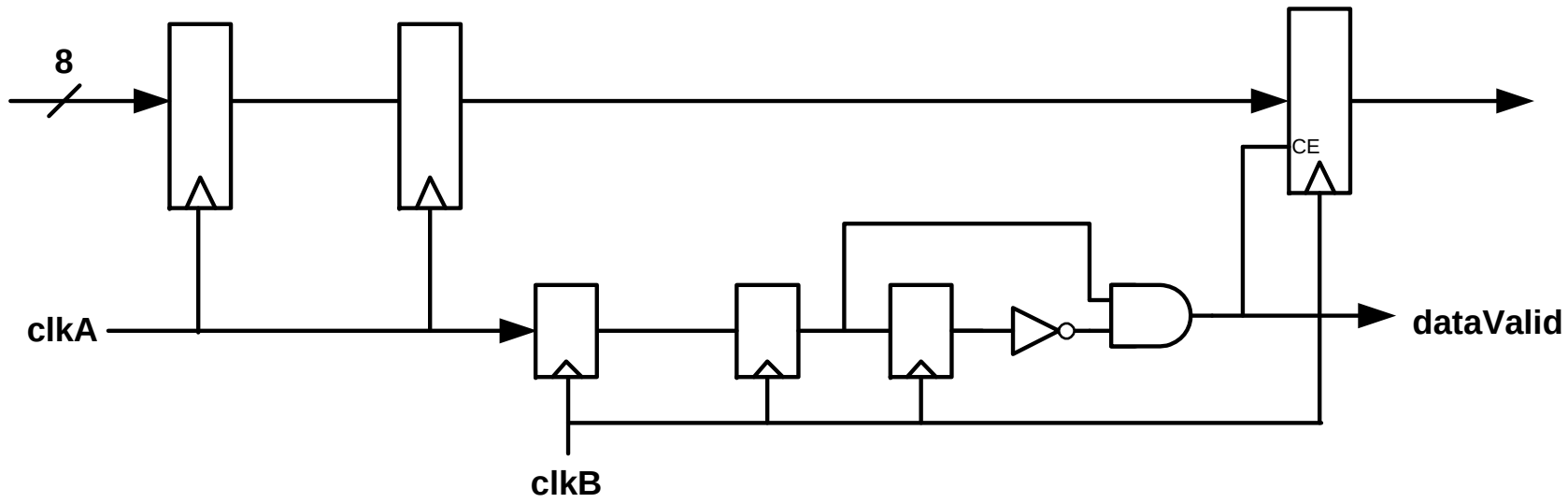
Sending Multiple Data Bits

- Balancing path delays . . .
- What's wrong with this solution?
- What's the right way to do it?



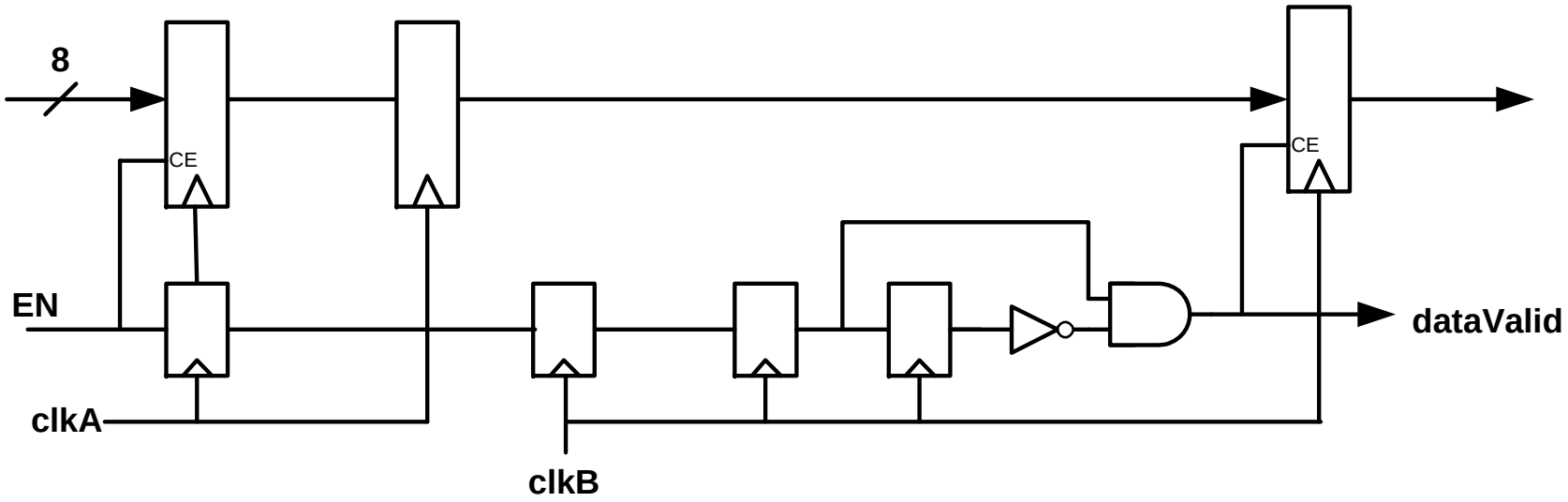
Sending Multiple Data Bits

- The right way to do it . . .



Sending Multiple Data Bits

- Slightly different alternative . . .



Sequential Logic
Counters and Registers

Counters

- Introduction: Counters
- Asynchronous (Ripple) Counters
- Asynchronous Counters with MOD number < 2^n
- Asynchronous Down Counters
- Cascading Asynchronous Counters

Lecture 13: Sequential Logic Counters and Registers

- Synchronous (Parallel) Counters
- Up/Down Synchronous Counters
- Designing Synchronous Counters
- Decoding A Counter
- Counters with Parallel Load

Lecture 13: Sequential Logic Counters and Registers

Registers

- Introduction: Registers
  - Simple Registers
  - Registers with Parallel Load
- Using Registers to implement Sequential Circuits
- Shift Registers
  - Serial In/Serial Out Shift Registers
  - Serial In/Parallel Out Shift Registers
  - Parallel In/Serial Out Shift Registers
  - Parallel In/Parallel Out Shift Registers

Lecture 13: Sequential Logic Counters and Registers

- Bidirectional Shift Registers
- An Application – Serial Addition
- Shift Register Counters
  - Ring Counters
  - Johnson Counters
- Random-Access Memory (RAM)

Introduction: Counters

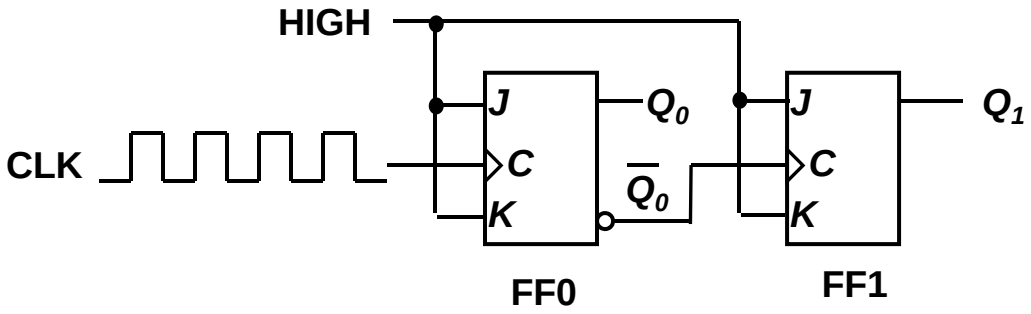
- Counters are circuits that cycle through a specified number of states.
- Two types of counters:
  - ❖ synchronous (parallel) counters
  - ❖ asynchronous (ripple) counters
- Ripple counters allow some flip-flop outputs to be used as a source of clock for other flip-flops.
- Synchronous counters apply the same clock to all flip-flops.

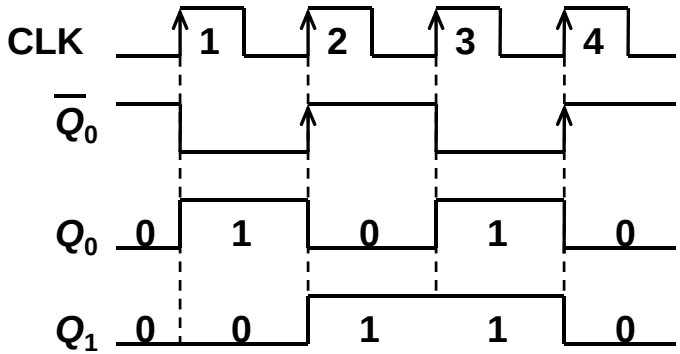
Asynchronous (Ripple) Counters

- Asynchronous counters: the flip-flops do not change states at exactly the same time as they do not have a common clock pulse.
- Also known as ripple counters, as the input clock pulse "ripples" through the counter – cumulative delay is a drawback.
- *n* flip-flops $\rightarrow$ a MOD (modulus) 2^n counter. (Note: A MOD-*x* counter cycles through *x* states.)
- Output of the last flip-flop (MSB) divides the input clock frequency by the MOD number of the counter, hence a counter is also a *frequency divider*.

Asynchronous (Ripple) Counters

- Example: 2-bit ripple binary counter.
- Output of one flip-flop is connected to the clock input of the next more-significant flip-flop.



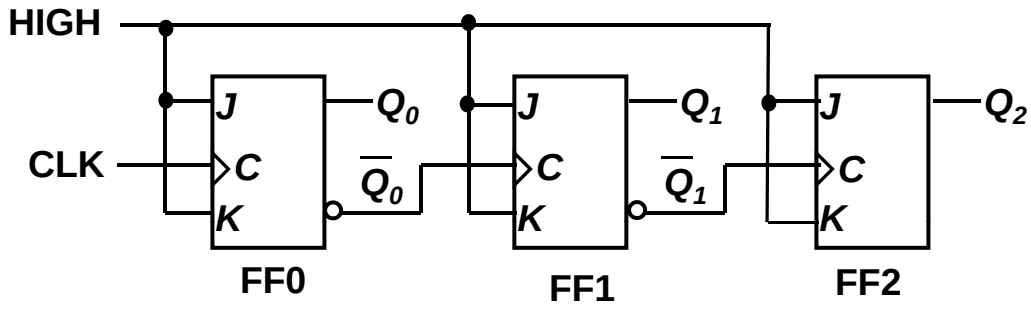


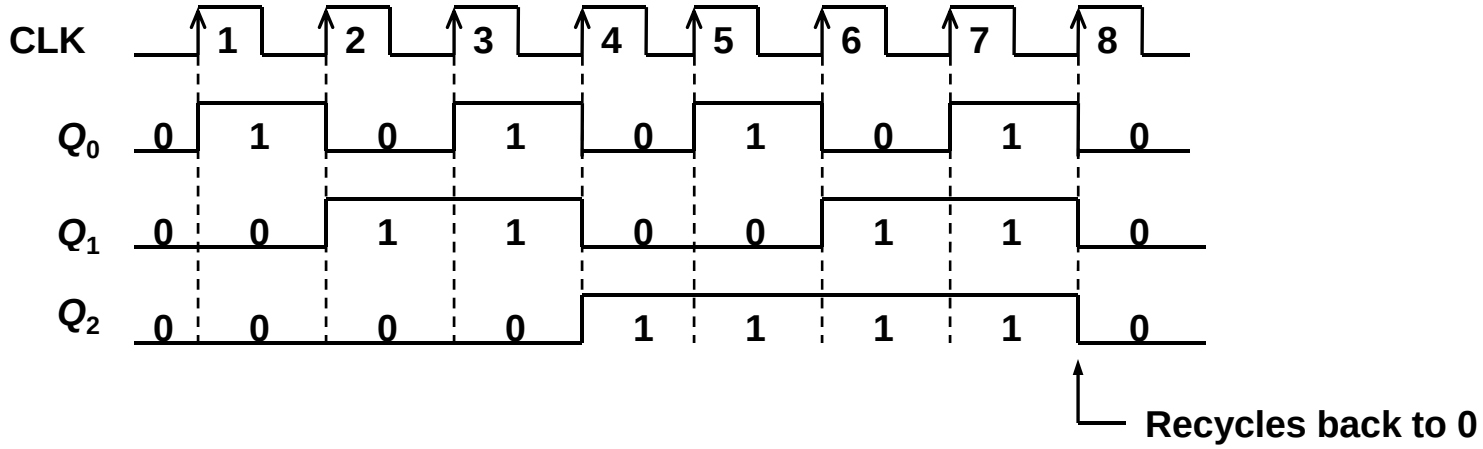
Timing diagram

$00 \rightarrow 01 \rightarrow 10 \rightarrow 11 \rightarrow 00$ ...

Asynchronous (Ripple) Counters

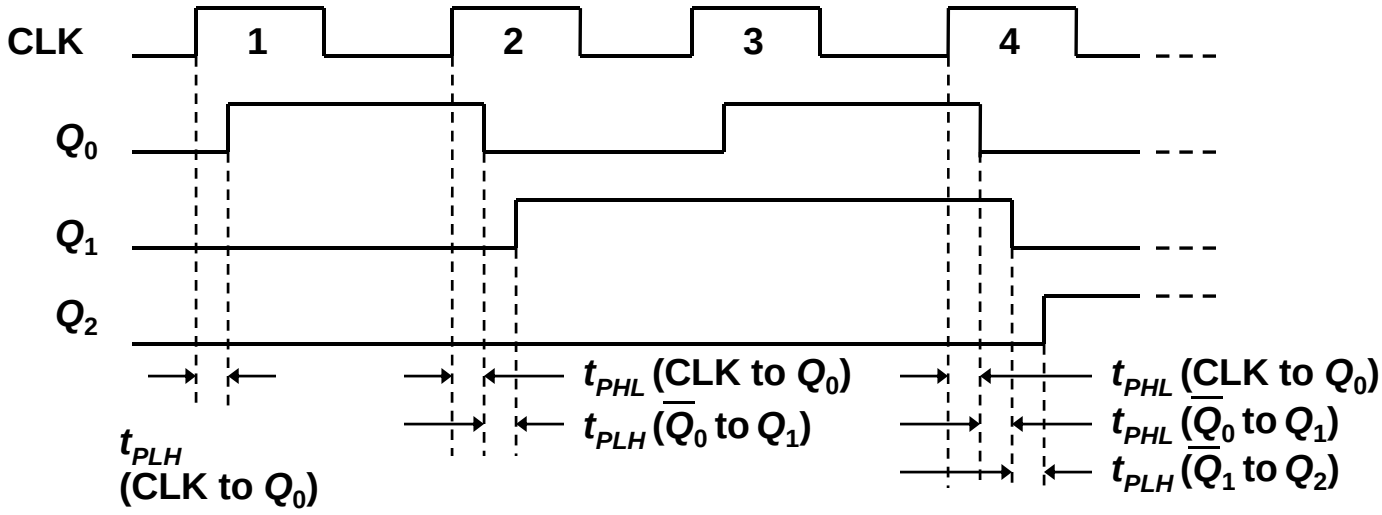
- Example: 3-bit ripple binary counter.





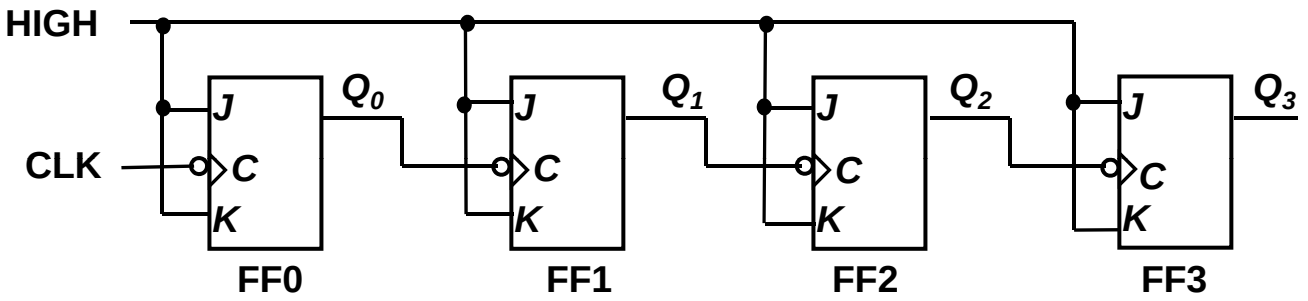
Asynchronous (Ripple) Counters

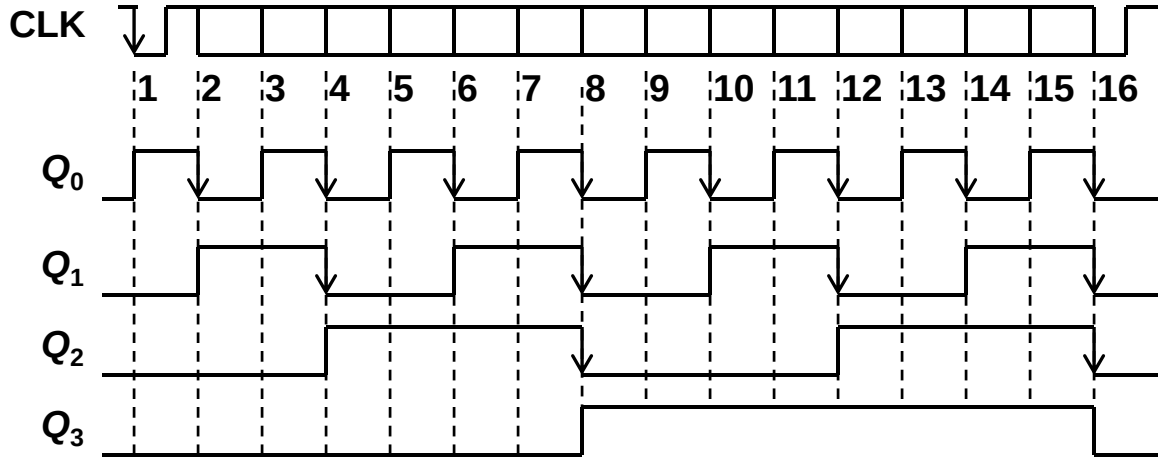
- Propagation delays in an asynchronous (ripple-clocked) binary counter.
- If the accumulated delay is greater than the clock pulse, some counter states may be misrepresented!



Asynchronous (Ripple) Counters

- Example: 4-bit ripple binary counter (negative-edge triggered).

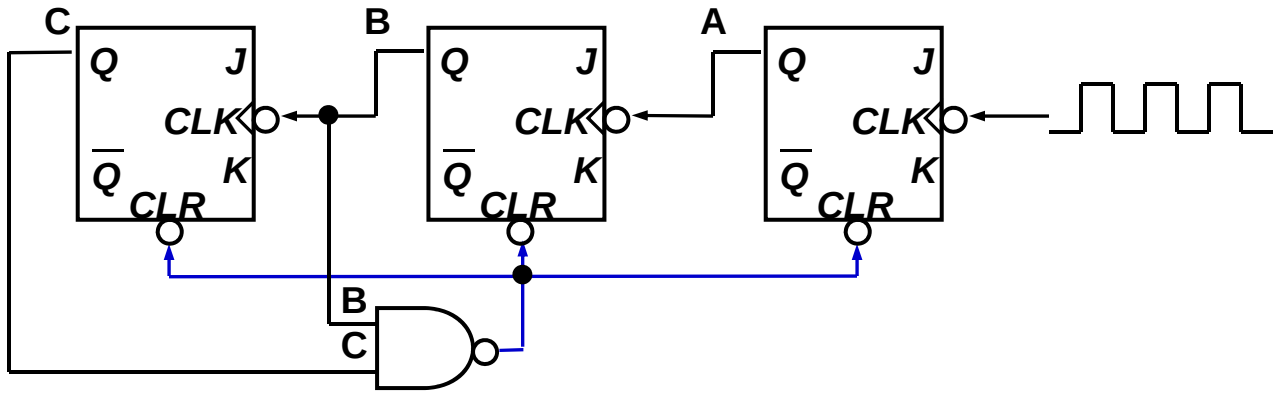




Asyn. Counters with MOD no. < 2^n

- States may be skipped resulting in a truncated sequence.
- Technique: force counter to *recycle before going through all of the states* in the binary sequence.
- Example: Given the following circuit, determine the counting sequence (and hence the modulus no.)

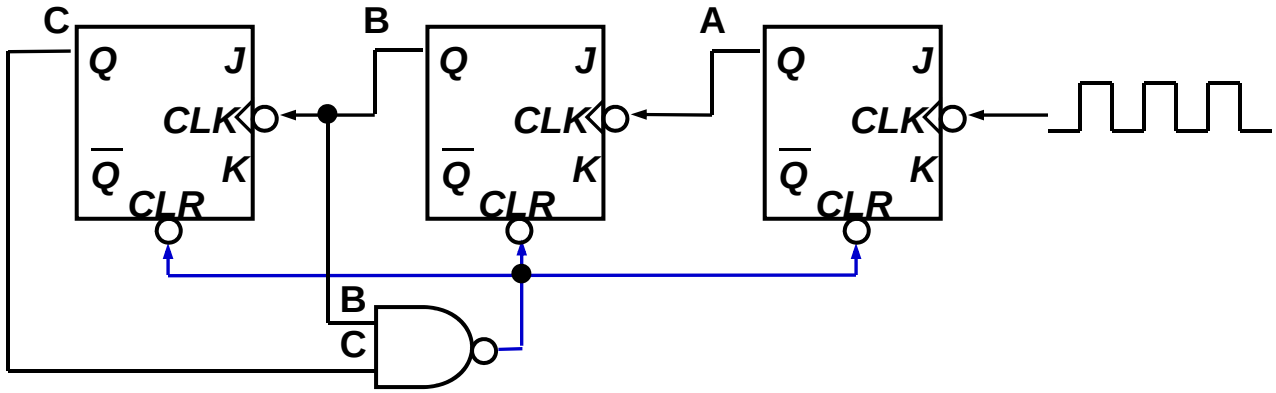
All *J*, *K* inputs are 1 (HIGH).

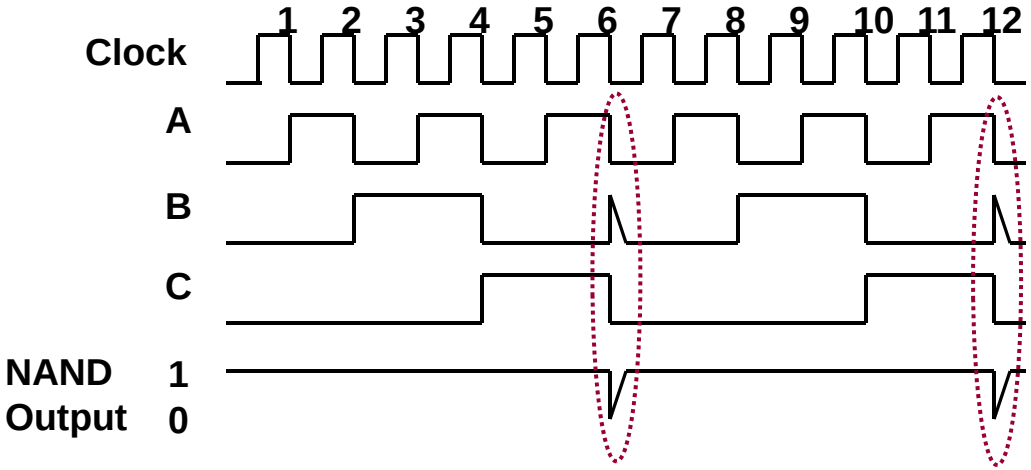


Asyn. Counters with MOD no. < 2^n

- Example (cont'd):

All *J*, *K* inputs are 1 (HIGH).

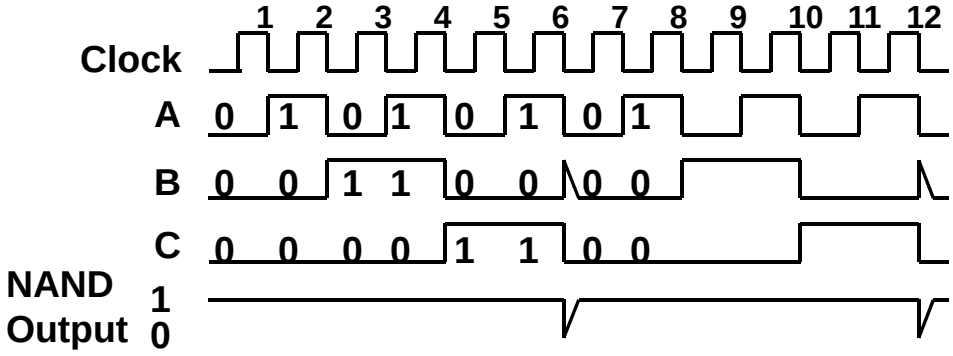


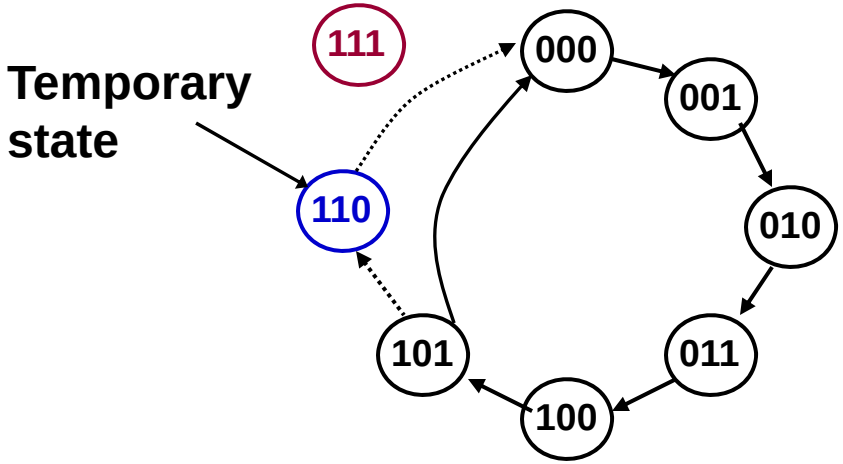


MOD-6 counter produced by clearing (a MOD-8 binary counter) when count of six (110) occurs.

Asyn. Counters with MOD no. < 2^n

- Example (cont'd): Counting sequence of circuit (in CBA order).

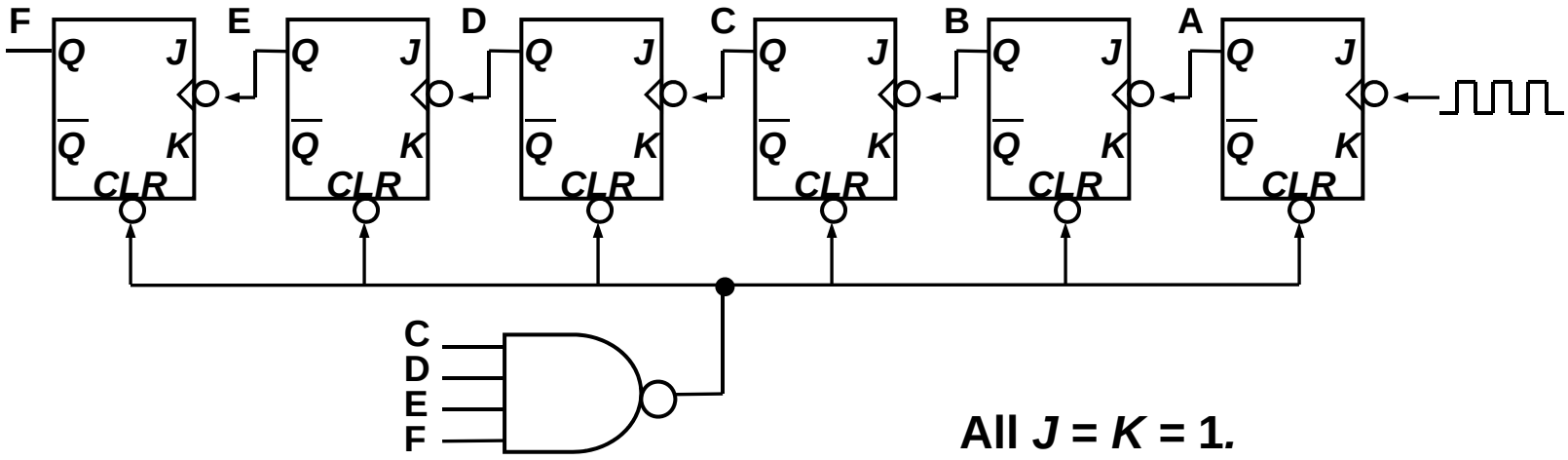




Counter is a MOD-6 counter.

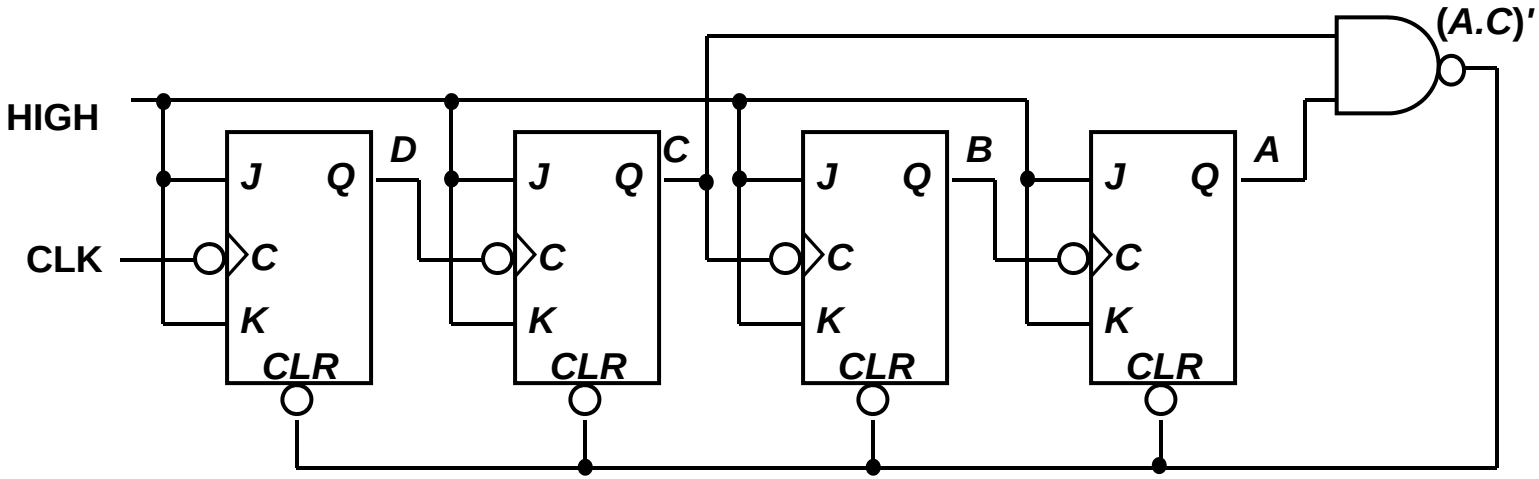
Asyn. Counters with MOD no. < 2^n

- *Exercise:* How to construct an asynchronous MOD-5 counter? MOD-7 counter? MOD-12 counter?
- *Question:* The following is a MOD-? counter?



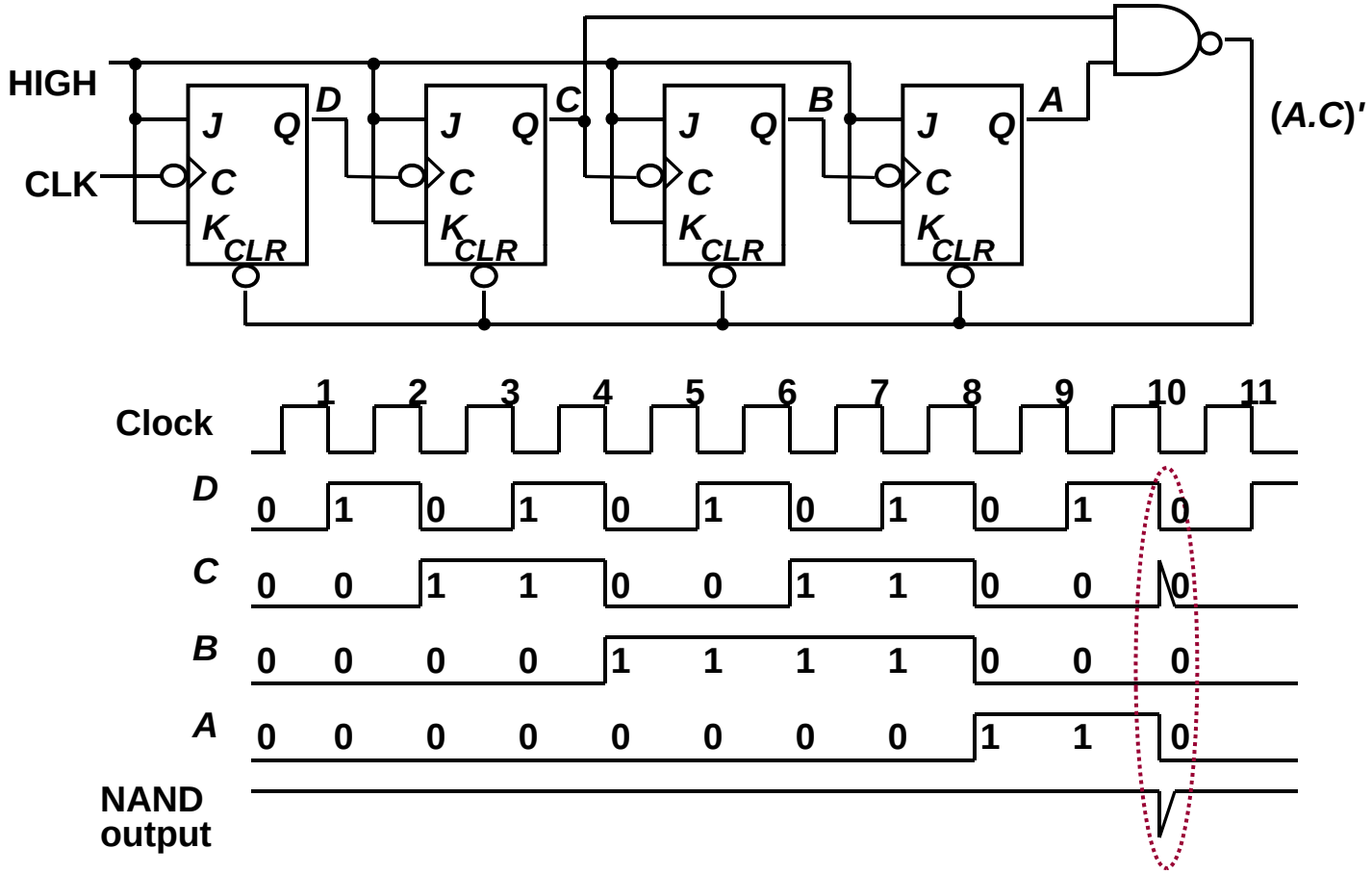
Asyn. Counters with MOD no. < 2^n

- Decade counters (or BCD counters) are counters with 10 states (modulus-10) in their sequence. They are commonly used in daily life (e.g.: utility meters, odometers, etc.).
- Design an asynchronous decade counter.



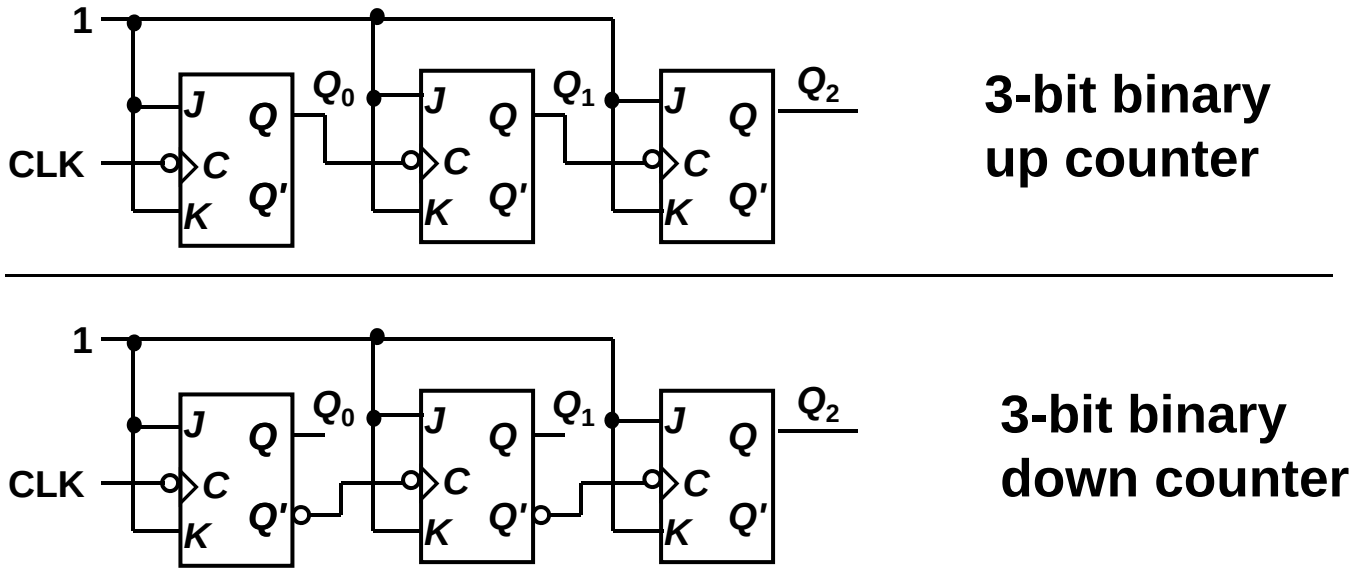
Asyn. Counters with MOD no. < 2^n

- Asynchronous decade/BCD counter (cont'd).



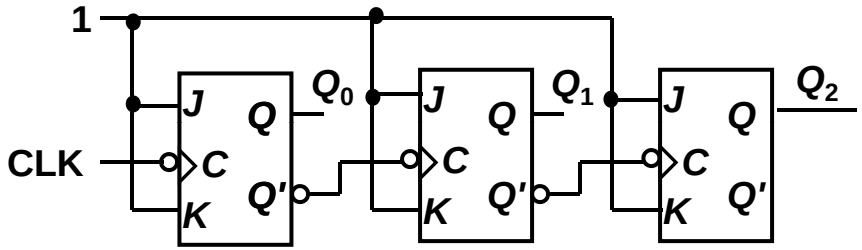
Asynchronous Down Counters

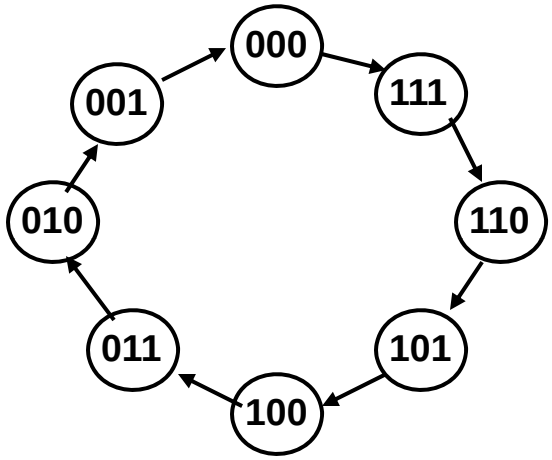
- So far we are dealing with *up counters*. *Down counters*, on the other hand, count downward from a maximum value to zero, and repeat.
- Example: A 3-bit binary (MOD-2^3) down counter.

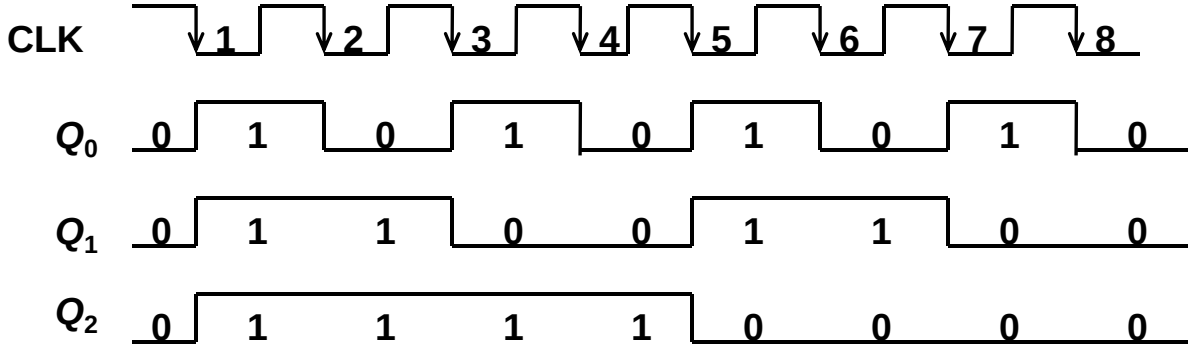


Asynchronous Down Counters

- Example: A 3-bit binary (MOD-8) down counter.

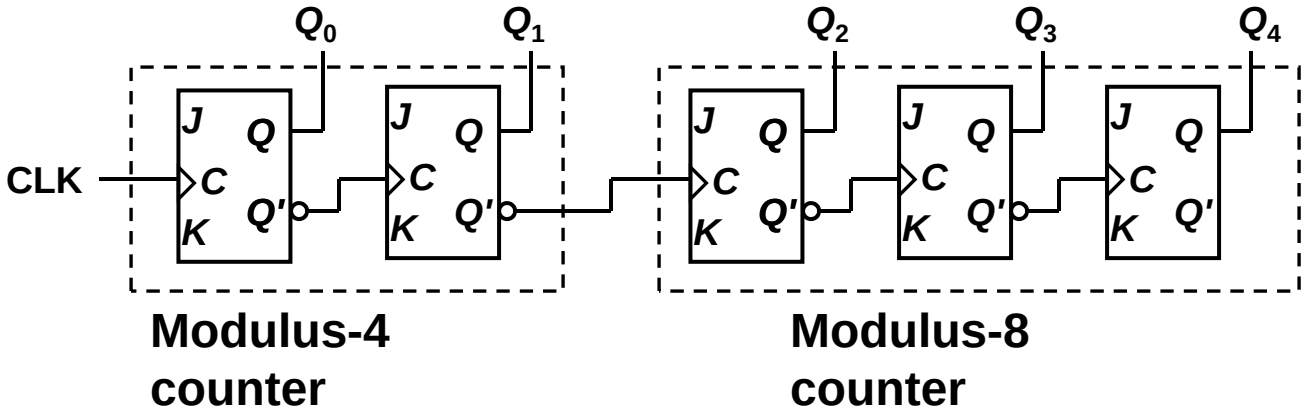






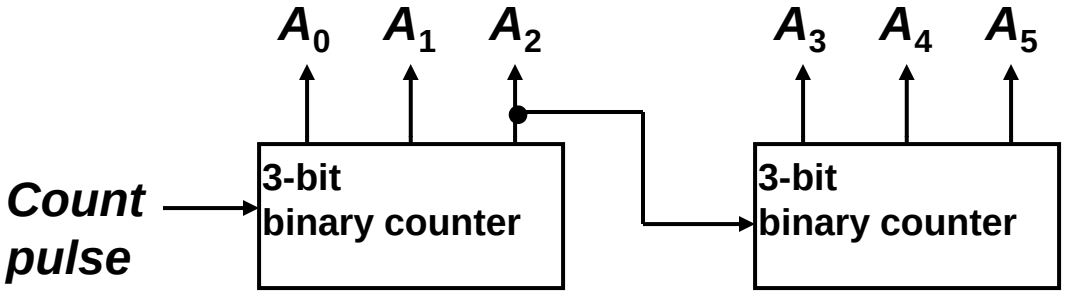
Cascading Asynchronous Counters

- Larger asynchronous (ripple) counter can be constructed by cascading smaller ripple counters.
- Connect last-stage output of one counter to the clock input of next counter so as to achieve higher-modulus operation.
- Example: A modulus-32 ripple counter constructed from a modulus-4 counter and a modulus-8 counter.



Cascading Asynchronous Counters

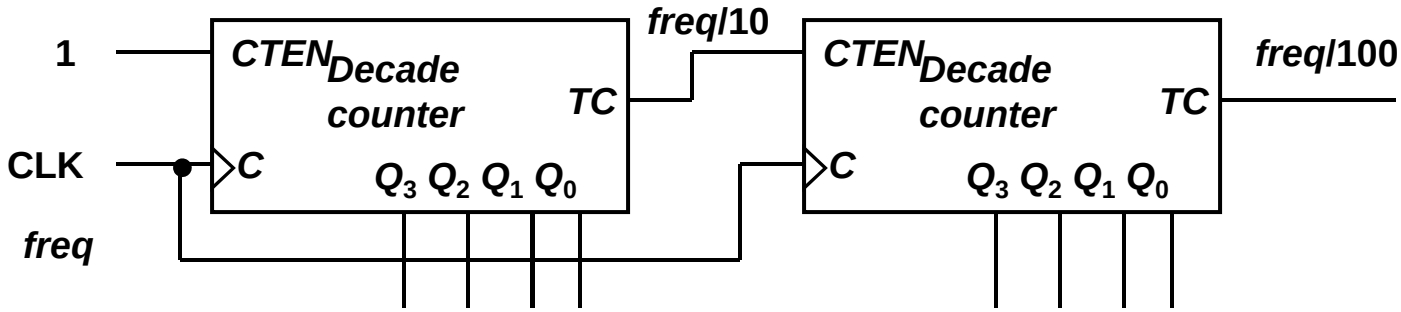
- Example: A 6-bit binary counter (counts from 0 to 63) constructed from two 3-bit counters.



A_5	A_4	A_3	A_2	A_1	A_0
0	0	0	0	0	0
0	0	0	0	0	1
0	0	0	:	:	:
0	0	0	1	1	1
0	0	**1**	0	0	0
0	0	1	0	0	1
:	:	:	:	:	:

Cascading Asynchronous Counters

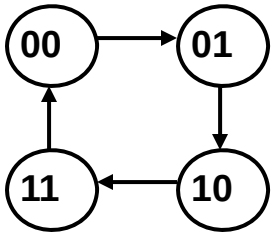
- If counter is a not a binary counter, requires additional output.
- Example: A modulus-100 counter using two decade counters.



***TC* = 1 when counter recycles to 0000**

Synchronous (Parallel) Counters

- Synchronous (parallel) counters: the flip-flops are clocked at the same time by a common clock pulse.
- We can design these counters using the sequential logic design process (covered in Lecture #12).
- Example: 2-bit synchronous binary counter (using T flip-flops, or JK flip-flops with identical J,K inputs).



Present state		Next state		Flip-flop inputs	
A_1	A_0	A_1^+	A_0^+	TA_1	TA_0
0	0	0	1	0	1
0	1	1	0	1	1
1	0	1	1	0	1
1	1	0	0	1	1

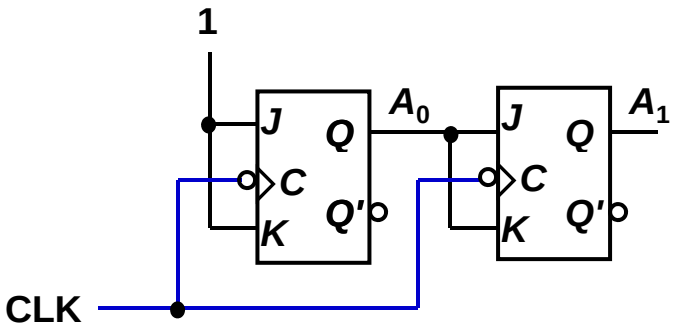
Synchronous (Parallel) Counters

- Example: 2-bit synchronous binary counter (using T flip-flops, or JK flip-flops with identical J,K inputs).

Present state		Next state		Flip-flop inputs	
A_1	A_0	A_1^+	A_0^+	TA_1	TA_0
0	0	0	1	0	1
0	1	1	0	1	1
1	0	1	1	0	1
1	1	0	0	1	1

$TA_1 = A_0$

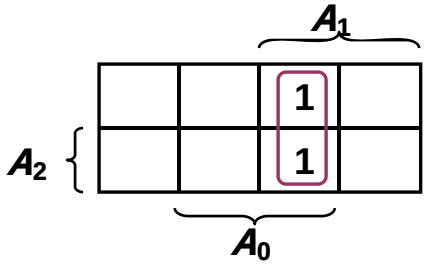
$TA_0 = 1$



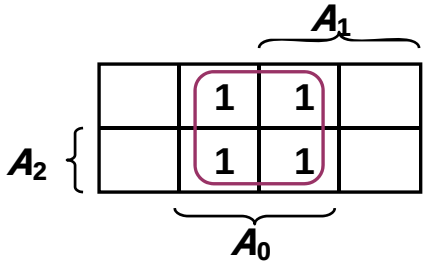
Synchronous (Parallel) Counters

- Example: 3-bit synchronous binary counter (using T flip-flops, or JK flip-flops with identical J, K inputs).

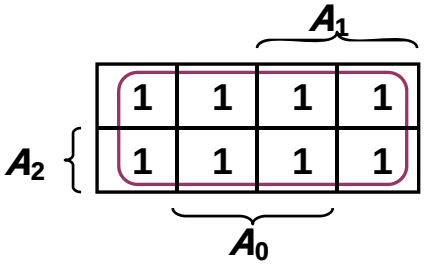
Present state			Next state			Flip-flop inputs		
A_2	A_1	A_0	A_2^+	A_1^+	A_0^+	TA_2	TA_1	TA_0
0	0	0	0	0	1	0	0	1
0	0	1	0	1	0	0	1	1
0	1	0	0	1	1	0	0	1
0	1	1	1	0	0	1	1	1
1	0	0	1	0	1	0	0	1
1	0	1	1	1	0	0	1	1
1	1	0	1	1	1	0	0	1
1	1	1	0	0	0	1	1	1



$TA_2 = A_1.A_0$



$TA_1 = A_0$

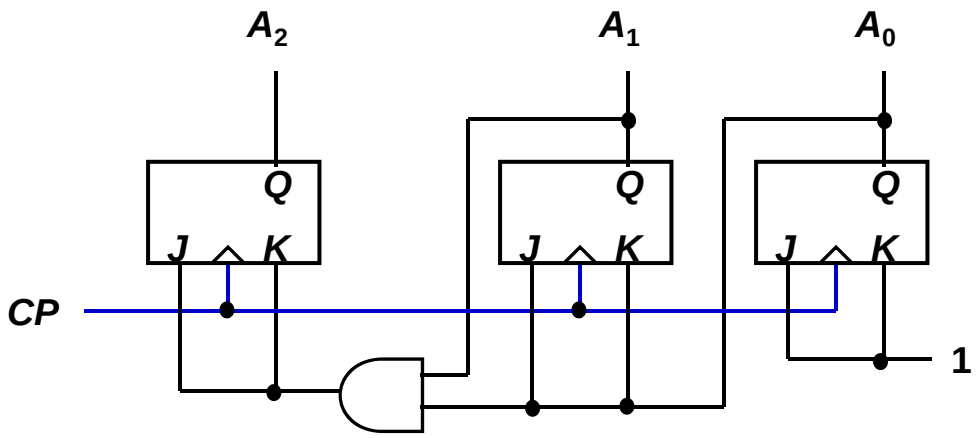


$TA_0 = 1$

Synchronous (Parallel) Counters

- Example: 3-bit synchronous binary counter (cont'd).

$TA_2 = A_1.A_0$ $TA_1 = A_0$ $TA_0 = 1$



Synchronous (Parallel) Counters

- Note that in a binary counter, the n^{th} bit (shown underlined) is always complemented whenever

  $\underline{0}11\ldots11 \rightarrow \underline{1}00\ldots00$

  or $\underline{1}11\ldots11 \rightarrow \underline{0}00\ldots00$

- Hence, X_n is complemented whenever

  $X_{n-1}X_{n-2} \ldots X_1X_0 = 11\ldots11.$

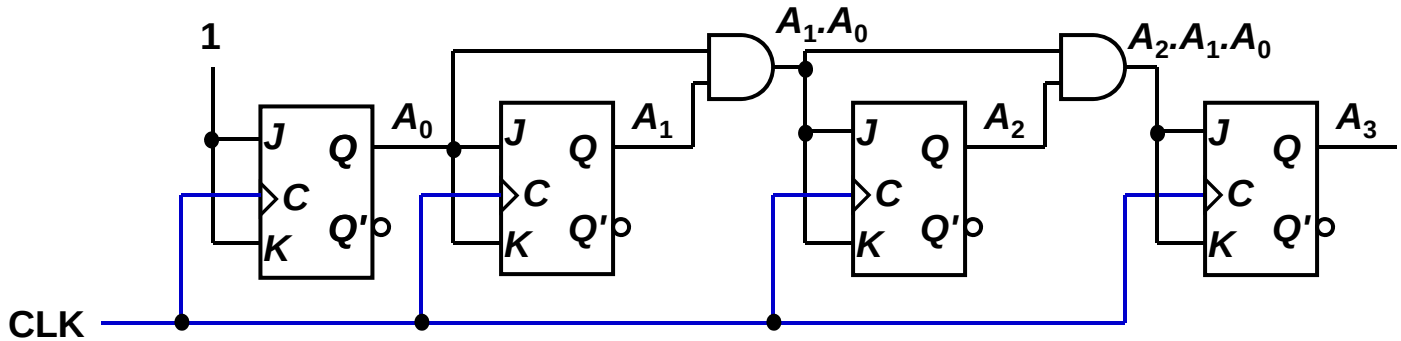
- As a result, if T flip-flops are used, then

  $TX_n = X_{n-1} \cdot X_{n-2} \cdot \ldots \cdot X_1 \cdot X_0$

Synchronous (Parallel) Counters

- Example: 4-bit synchronous binary counter.

$$TA_3 = A_2 \cdot A_1 \cdot A_0$$
$$TA_2 = A_1 \cdot A_0$$
$$TA_1 = A_0$$
$$TA_0 = 1$$



Synchronous (Parallel) Counters

- Example: Synchronous decade/BCD counter.

Clock pulse	Q_3	Q_2	Q_1	Q_0
Initially	0	0	0	0
1	0	0	0	1
2	0	0	1	0
3	0	0	1	1
4	0	1	0	0
5	0	1	0	1
6	0	1	1	0
7	0	1	1	1
8	1	0	0	0
9	1	0	0	1
10 (recycle)	0	0	0	0

$$T_0 = 1$$

$$T_1 = Q_3'.Q_0$$

$$T_2 = Q_1.Q_0$$

$$T_3 = Q_2.Q_1.Q_0 + Q_3.Q_0$$

Synchronous (Parallel) Counters

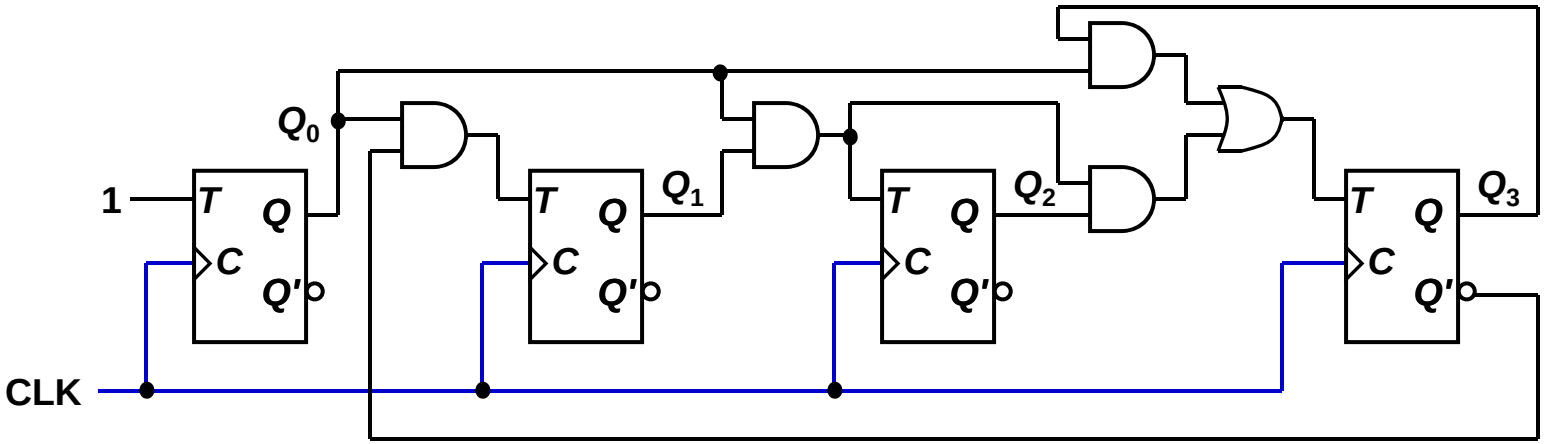
- Example: Synchronous decade/BCD counter (cont'd).

$T_0 = 1$

$T_1 = Q_3'.Q_0$

$T_2 = Q_1.Q_0$

$T_3 = Q_2.Q_1.Q_0 + Q_3.Q_0$



Up/Down Synchronous Counters

- Up/down synchronous counter: a *bidirectional* counter that is capable of counting either up or down.
- An input (control) line $\overline{Up/Down}$ (or simply *Up*) specifies the direction of counting.
  - $Up/\overline{Down} = 1 \rightarrow$ Count upward
  - $Up/\overline{Down} = 0 \rightarrow$ Count downward

Up/Down Synchronous Counters

- Example: A 3-bit up/down synchronous binary counter.

Clock pulse	*Up*	Q_2	Q_1	Q_0	*Down*
0		0	0	0	
1		0	0	1	
2		0	1	0	
3		0	1	1	
4		1	0	0	
5		1	0	1	
6		1	1	0	
7		1	1	1	

$TQ_0 = 1$

$TQ_1 = (Q_0.Up) + (Q_0'.Up')$

$TQ_2 = (Q_0.Q_1.Up) + (Q_0'.Q_1'.Up')$

Up counter	Down counter
$TQ_0 = 1$	$TQ_0 = 1$
$TQ_1 = Q_0$	$TQ_1 = Q_0'$
$TQ_2 = Q_0.Q_1$	$TQ_2 = Q_0'.Q_1'$

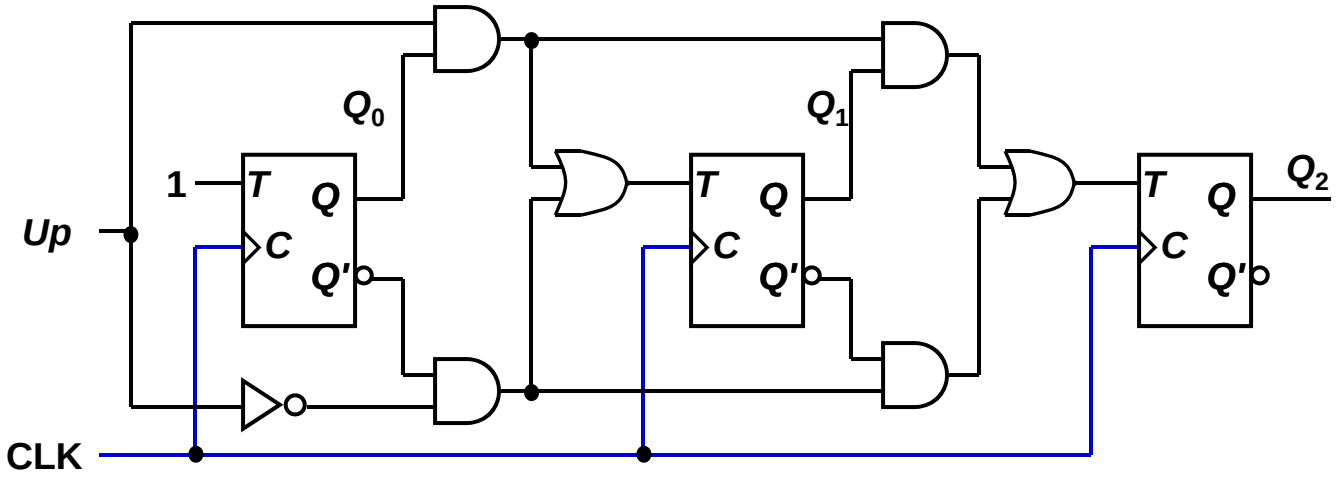
Up/Down Synchronous Counters

- Example: A 3-bit up/down synchronous binary counter (cont'd).

$$TQ_0 = 1$$

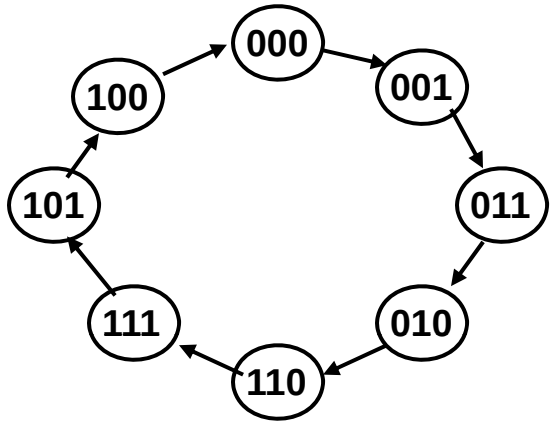
$$TQ_1 = (Q_0.Up) + (Q_0'.Up')$$

$$TQ_2 = (Q_0.Q_1.Up) + (Q_0'.Q_1'.Up')$$



Designing Synchronous Counters

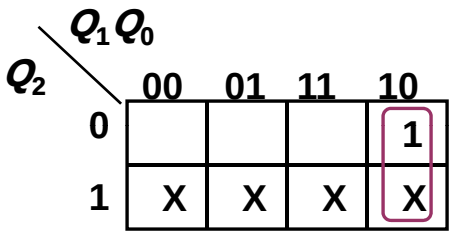
- Covered in Lecture #12.
- Example: A 3-bit Gray code counter (using JK flip-flops).



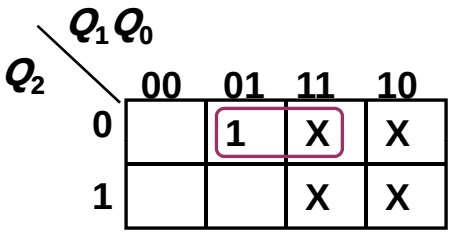
Present state			Next state			Flip-flop inputs					
Q_2	Q_1	Q_0	Q_2^+	Q_1^+	Q_0^+	JQ_2	KQ_2	JQ_1	KQ_1	JQ_0	KQ_0
0	0	0	0	0	1	0	X	0	X	1	X
0	0	1	0	1	1	0	X	1	X	X	0
0	1	0	1	1	0	1	X	X	0	0	X
0	1	1	0	1	0	0	X	X	0	X	1
1	0	0	0	0	0	X	1	0	X	0	X
1	0	1	1	0	0	X	0	0	X	X	1
1	1	0	1	1	1	X	0	X	0	1	X
1	1	1	1	0	1	X	0	X	1	X	0

Designing Synchronous Counters

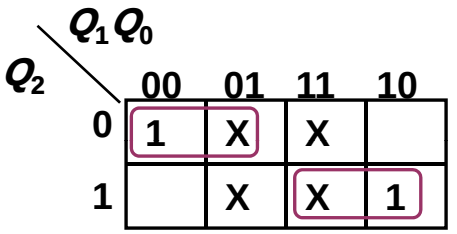
- 3-bit Gray code counter: flip-flop inputs.



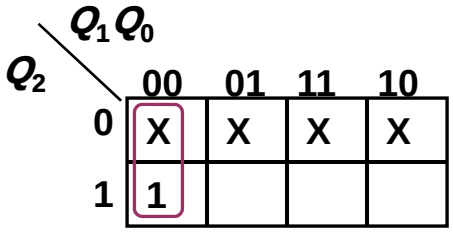
$JQ_2 = Q_1.Q_0'$



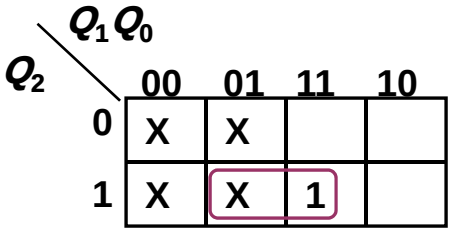
$JQ_1 = Q_2'.Q_0$



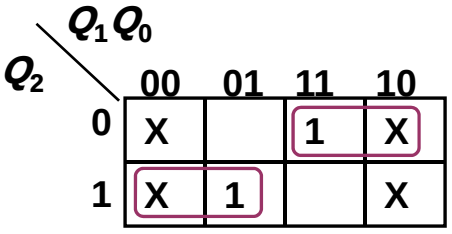
$JQ_0 = Q_2.Q_1 + Q_2'.Q_1'$
$= (Q_2 \oplus Q_1)'$



$KQ_2 = Q_1'.Q_0'$



$KQ_1 = Q_2.Q_0$



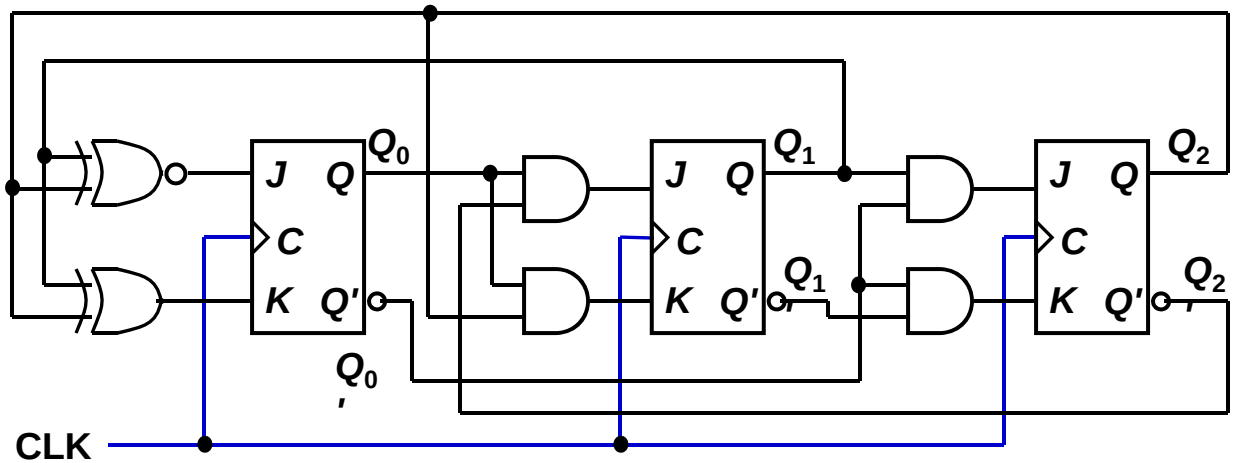
$KQ_0 = Q_2.Q_1' + Q_2'.Q_1$
$= Q_2 \oplus Q_1$

Designing Synchronous Counters

- 3-bit Gray code counter: logic diagram.

$JQ_2 = Q_1.Q_0'$ $JQ_1 = Q_2'.Q_0$ $JQ_0 = (Q_2 \oplus Q_1)'$

$KQ_2 = Q_1'.Q_0'$ $KQ_1 = Q_2.Q_0$ $KQ_0 = Q_2 \oplus Q_1$

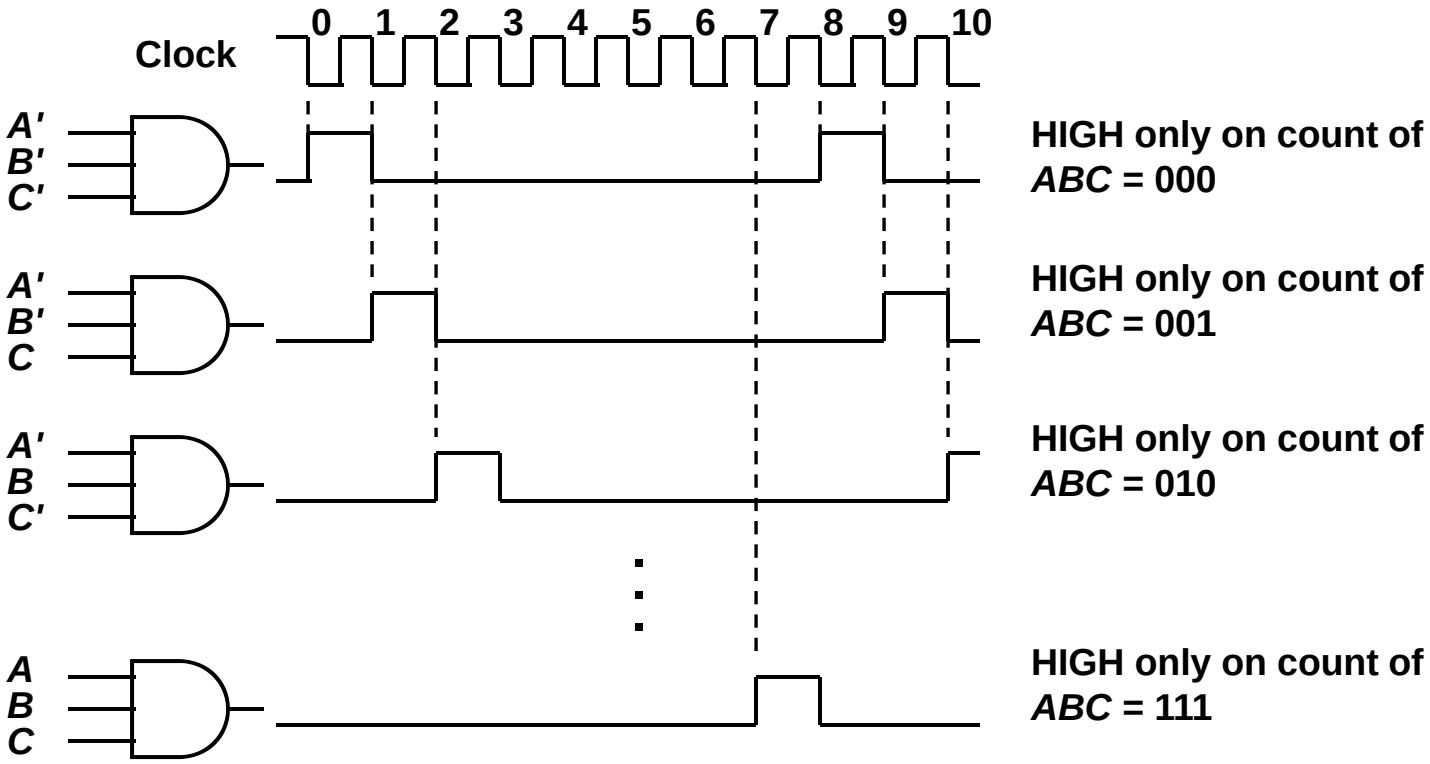


Decoding A Counter

- Decoding a counter involves determining which state in the sequence the counter is in.
- Differentiate between *active-HIGH* and *active-LOW* decoding.
- Active-HIGH decoding: output HIGH if the counter is in the state concerned.
- Active-LOW decoding: output LOW if the counter is in the state concerned.

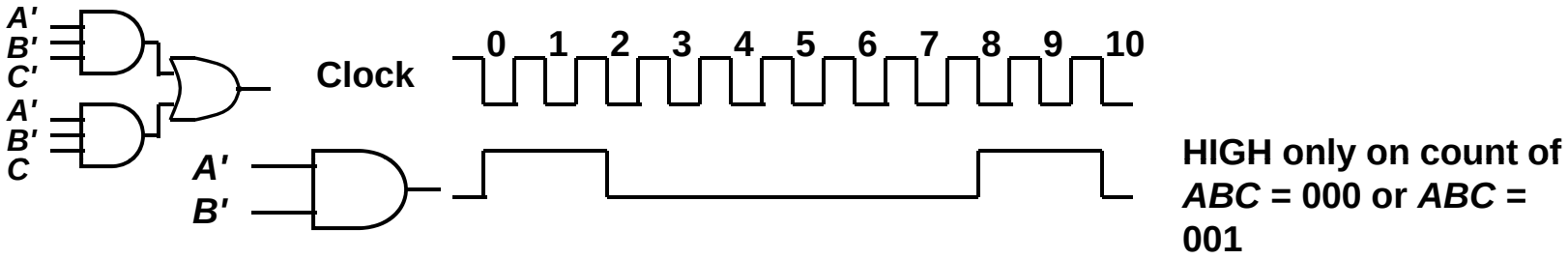
Decoding A Counter

- Example: MOD-8 ripple counter (active-HIGH decoding).

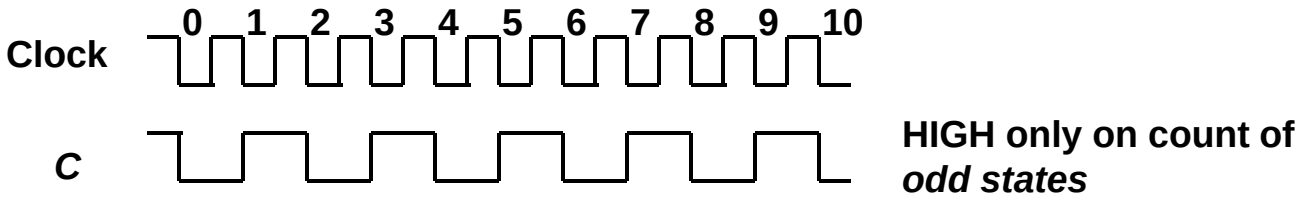


Decoding A Counter

- Example: To detect that a MOD-8 counter is in state 0 (000) or state 1 (001).



- **Example: To detect that a MOD-8 counter is in the odd states (states 1, 3, 5 or 7), simply use *C*.**

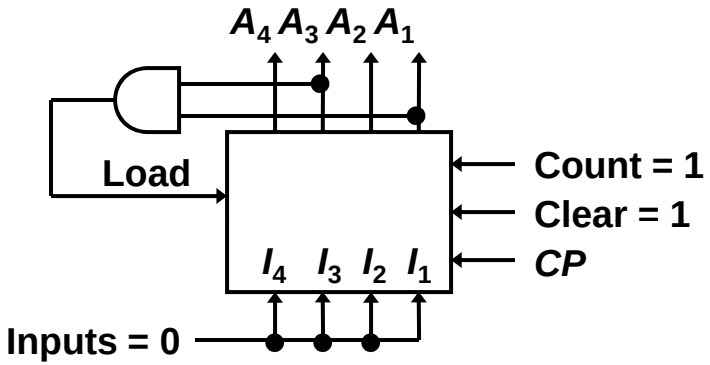


Counters with Parallel Load

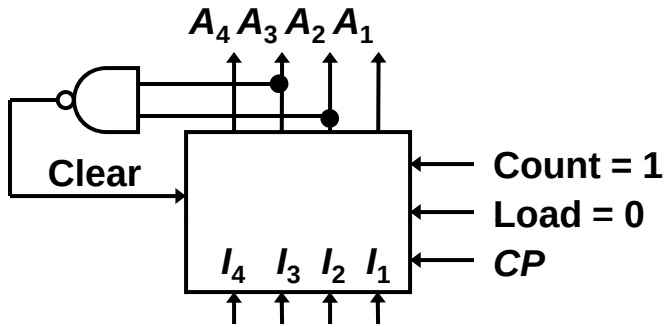
- Counters could be augmented with parallel load capability for the following purposes:
  - To start at a different state
  - To count a different sequence
  - As more sophisticated register with increment/decrement functionality.

Counters with Parallel Load

- Different ways of getting a MOD-6 counter:

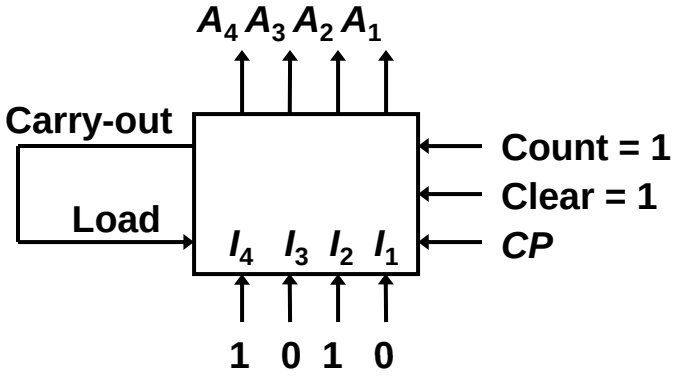


(a) Binary states 0,1,2,3,4,5.

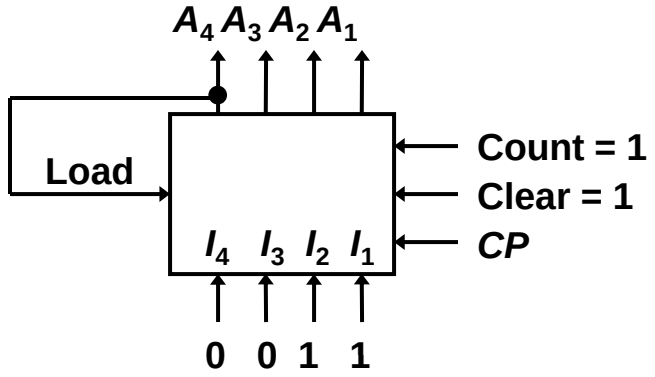


Inputs have no effect

(b) Binary states 0,1,2,3,4,5.



(c) Binary states 10,11,12,13,14,15.

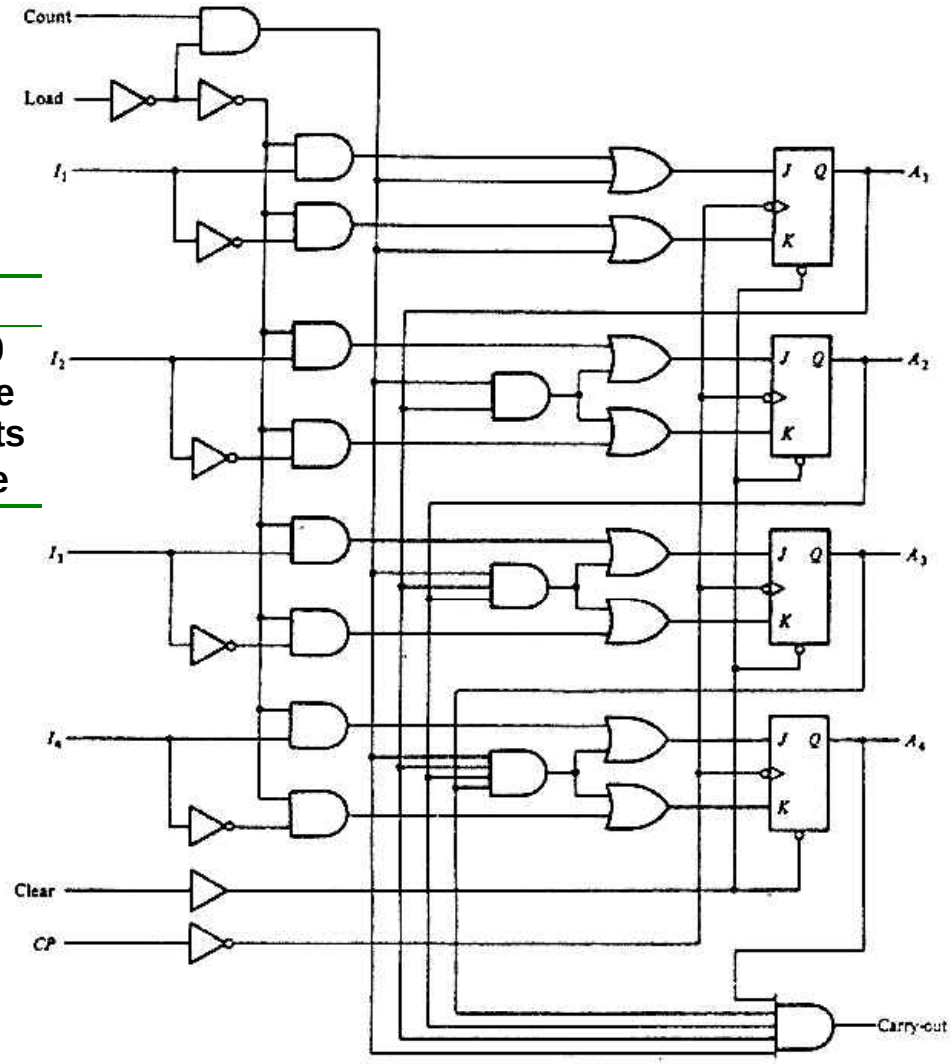


(d) Binary states 3,4,5,6,7,8.

Counters with Parallel Load

- 4-bit counter with parallel load.

Clear	CP	Load	Count	Function
0	X	X	X	Clear to 0
1	X	0	0	No change
1	↑	1	X	Load inputs
1	↑	0	1	Next state

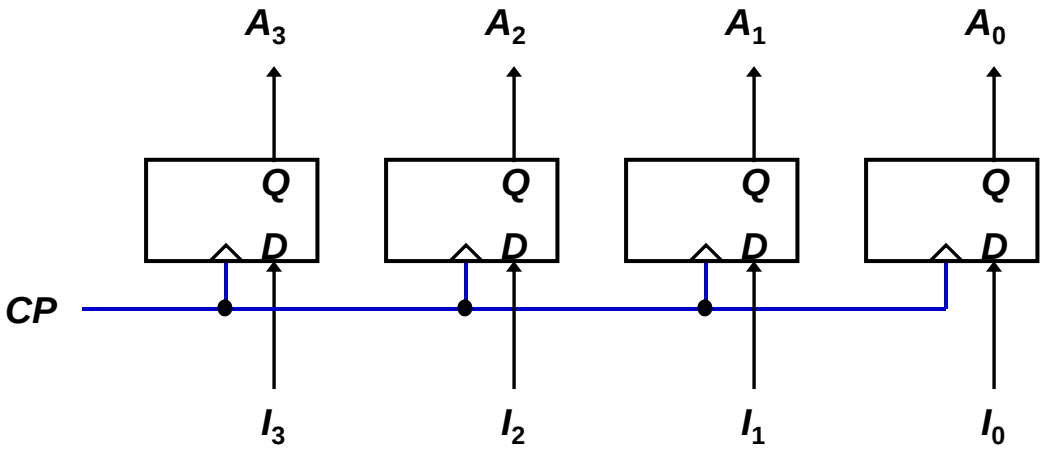


Introduction: Registers

- An *n*-bit register has a group of *n* flip-flops and some logic gates and is capable of storing *n* bits of information.
- The flip-flops store the information while the gates control when and how new information is transferred into the register.
- Some functions of register:
  - ❖ retrieve data from register
  - ❖ store/load new data into register (serial or parallel)
  - ❖ shift the data within register (left or right)

Simple Registers

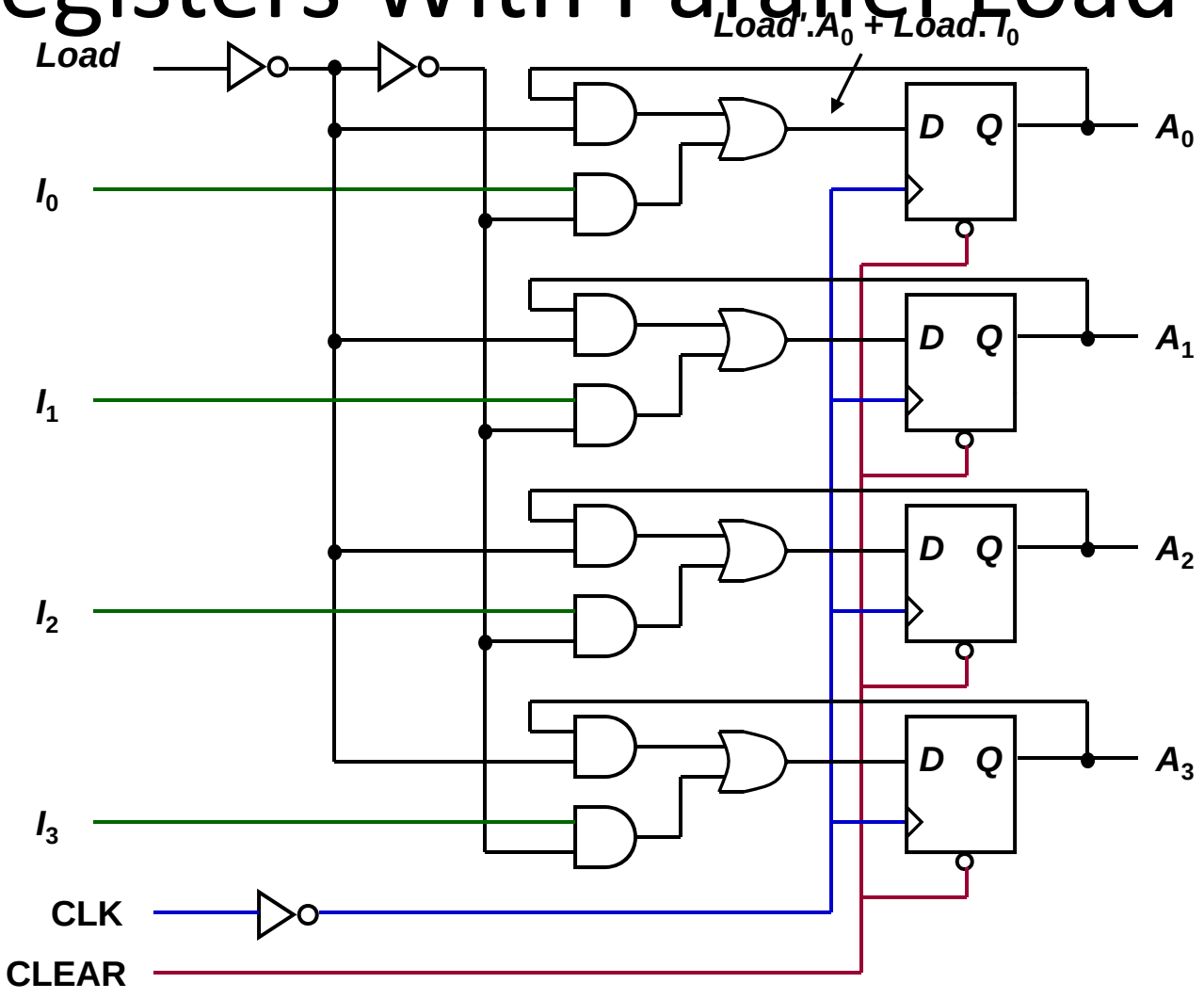
- No external gates.
- Example: A 4-bit register. A new 4-bit data is loaded every clock cycle.



Registers With Parallel Load

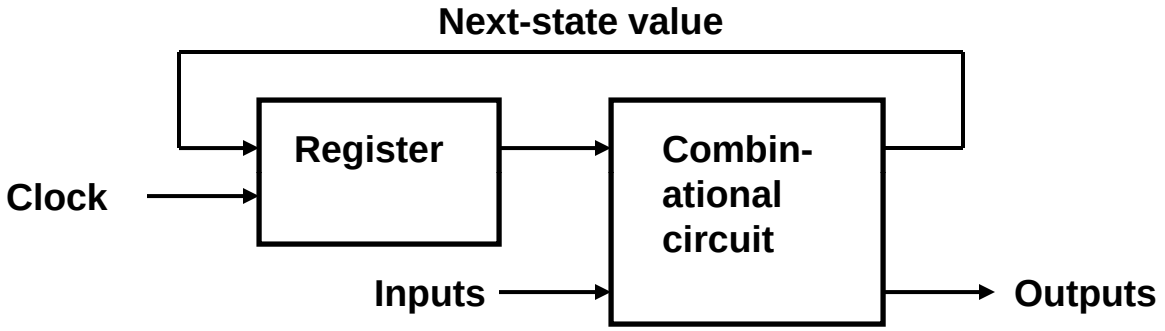
- Instead of loading the register at every clock pulse, we may want to control when to load.
- *Loading* a register: transfer new information into the register. Requires a *load* control input.
- *Parallel loading*: all bits are loaded simultaneously.

Registers With Parallel Load



Using Registers to implement Sequential Circuits

- A sequential circuit may consist of a *register* (memory) and a *combinational circuit*.



- **The external inputs and present states of the register determine the next states of the register and the external outputs, through the combinational circuit.**
- **The combinational circuit may be implemented by any of the methods covered in *MSI components* and *Programmable Logic Devices*.**

Using Registers to implement Sequential Circuits

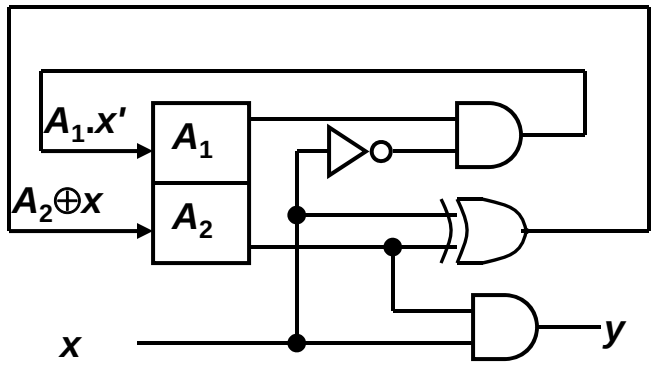
- Example 1:

$A_1^+ = \Sigma\, m(4,6) = A_1.x'$

$A_2^+ = \Sigma\, m(1,2,5,6) = A_2.x' + A_2'.x = A_2 \oplus x$

$y = \Sigma\, m(3,7) = A_2.x$

Present state		Input	Next State		Output
A_1	A_2	x	A_1^+	A_2^+	y
0	0	0	0	0	0
0	0	1	0	1	0
0	1	0	0	1	0
0	1	1	0	0	1
1	0	0	1	0	0
1	0	1	0	1	0
1	1	0	1	1	0
1	1	1	0	0	1

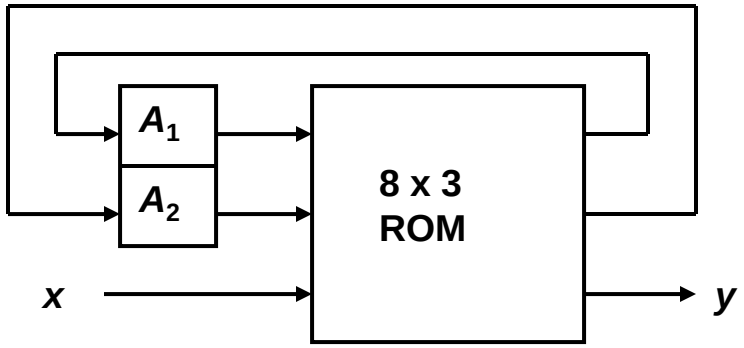


Using Registers to implement Sequential Circuits

- Example 2: Repeat example 1, but use a ROM.

Address			Outputs		
1	2	3	1	2	3
0	0	0	0	0	0
0	0	1	0	1	0
0	1	0	0	1	0
0	1	1	0	0	1
1	0	0	1	0	0
1	0	1	0	1	0
1	1	0	1	1	0
1	1	1	0	0	1

ROM truth table

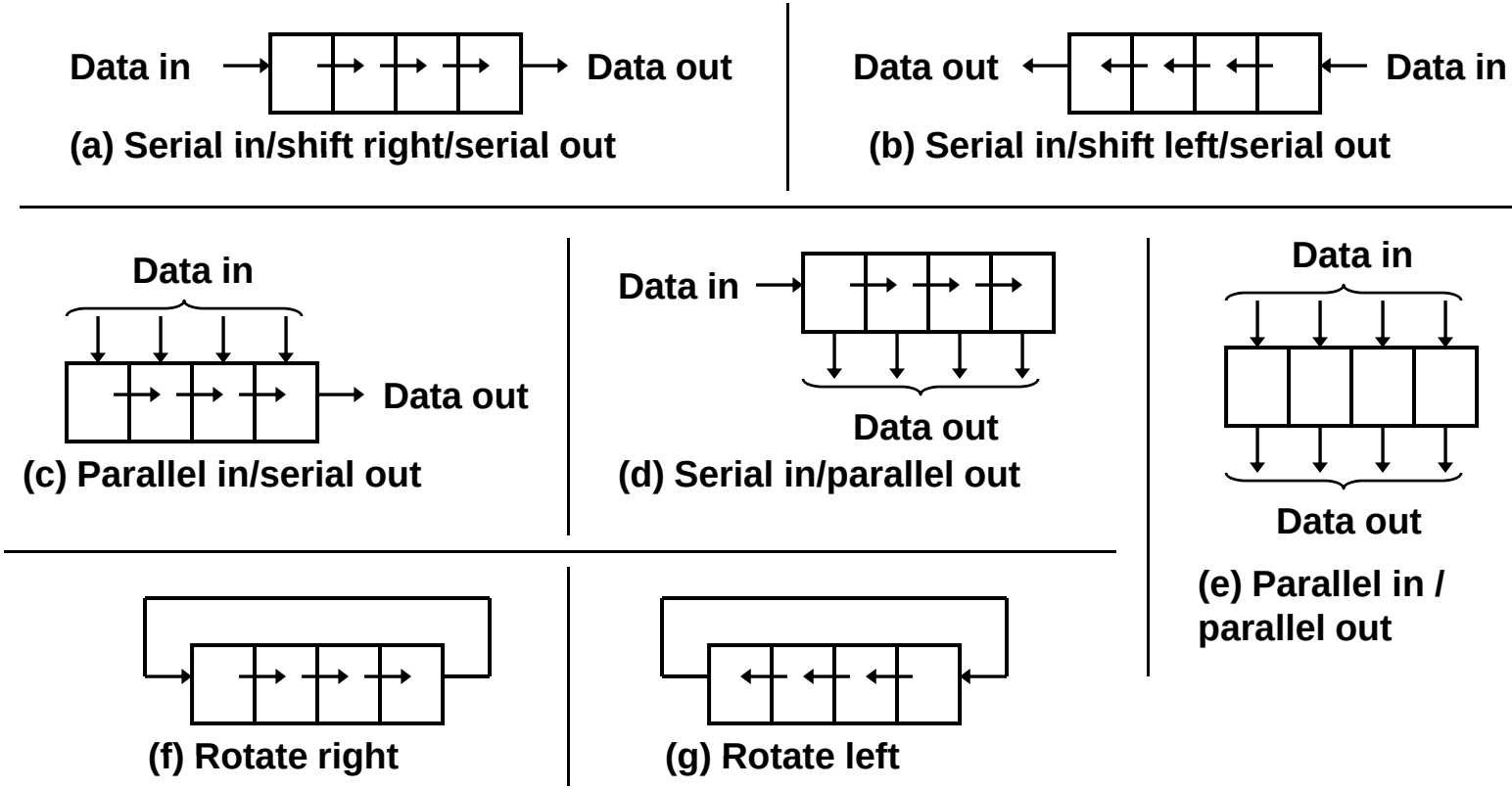


Shift Registers

- Another function of a register, besides storage, is to provide for *data movements*.
- Each *stage* (flip-flop) in a shift register represents one bit of storage, and the shifting capability of a register permits the movement of data from stage to stage within the register, or into or out of the register upon application of clock pulses.

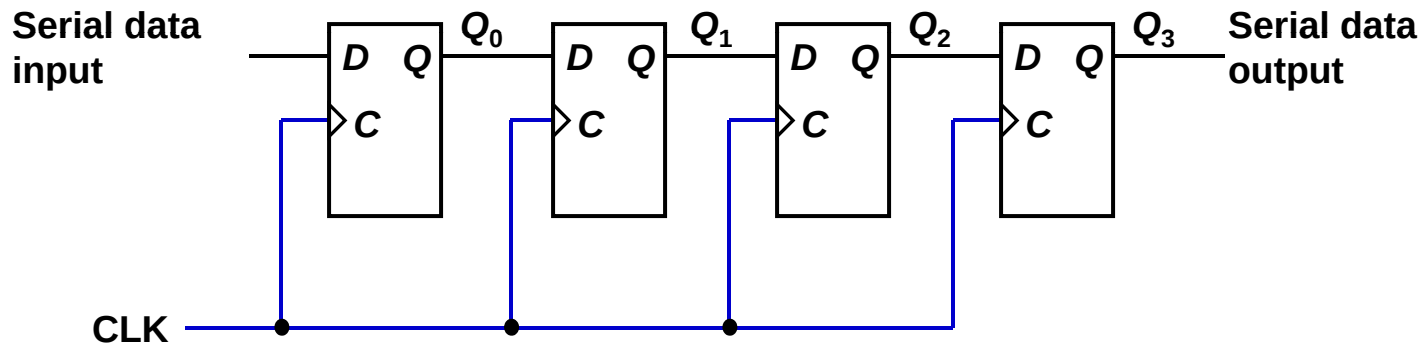
Shift Registers

- Basic data movement in shift registers (four bits are used for illustration).



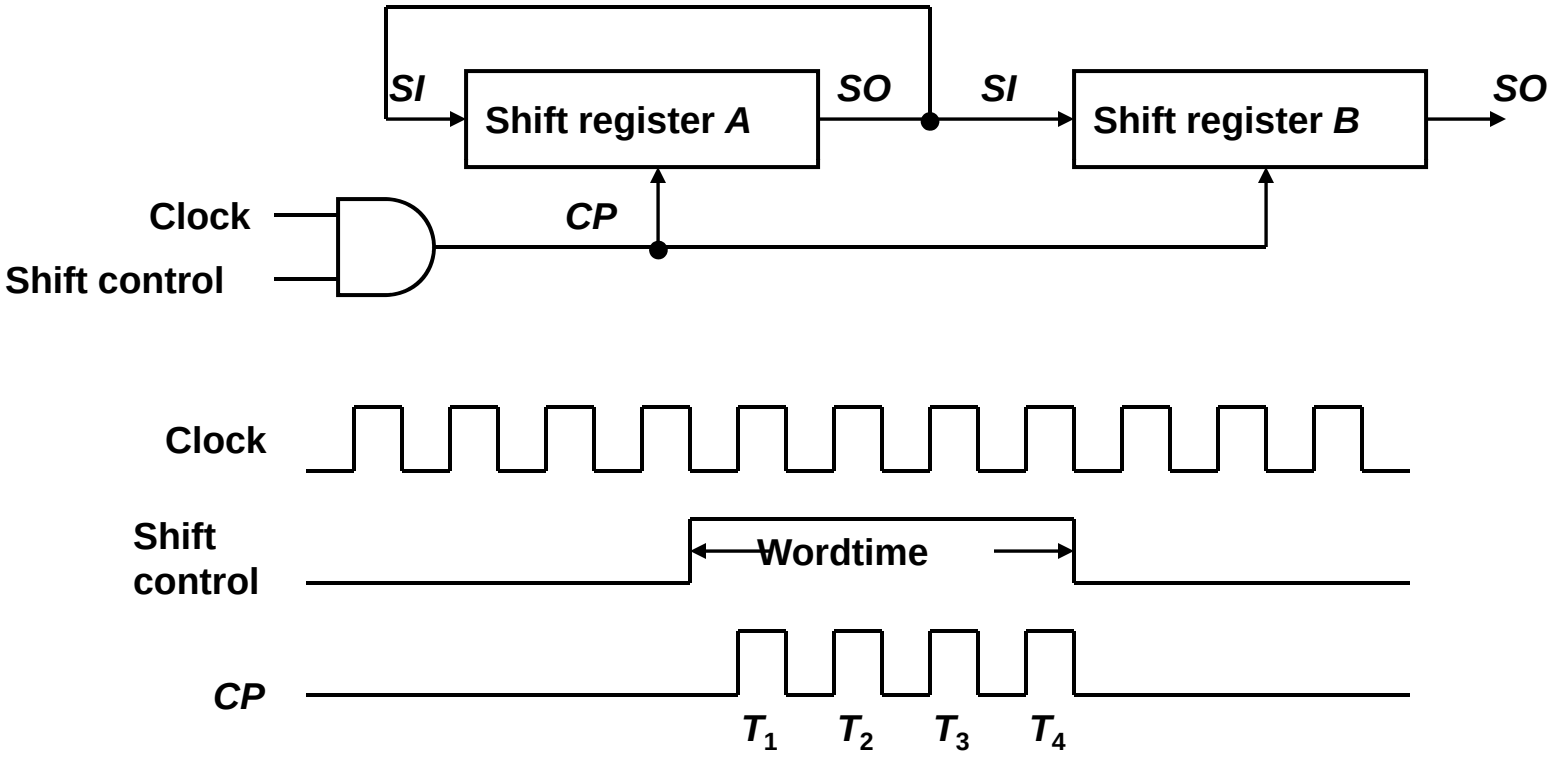
Serial In/Serial Out Shift Registers

- Accepts data serially – one bit at a time – and also produces output serially.



Serial In/Serial Out Shift Registers

- Application: Serial transfer of data from one register to another.



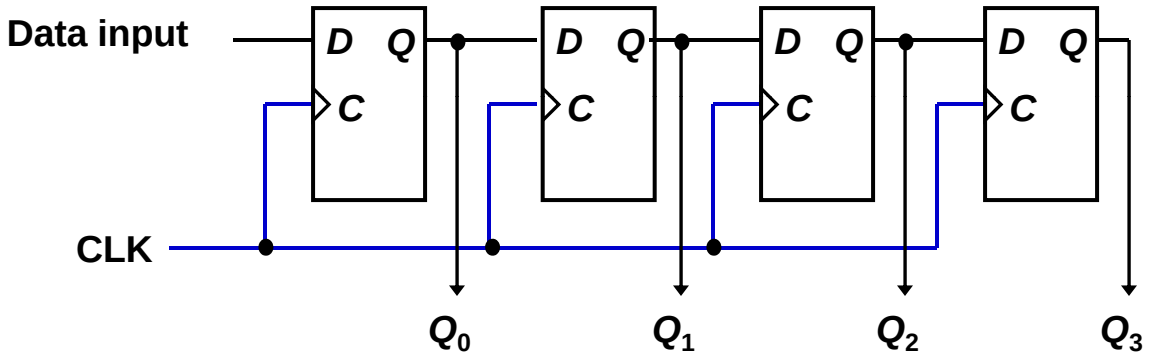
Serial In/Serial Out Shift Registers

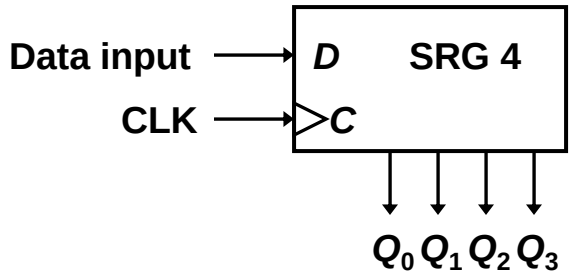
- Serial-transfer example.

Timing Pulse	Shift register *A*				Shift register *B*				Serial output of *B*
Initial value	1	0	1	1	0	0	1	0	0
After T_1	1	1	0	1	1	0	0	1	1
After T_2	1	1	1	0	1	1	0	0	0
After T_3	0	1	1	1	0	1	1	0	0
After T_4	1	0	1	1	1	0	1	1	1

Serial In/Parallel Out Shift Registers

- Accepts data serially.
- Outputs of all stages are available simultaneously.

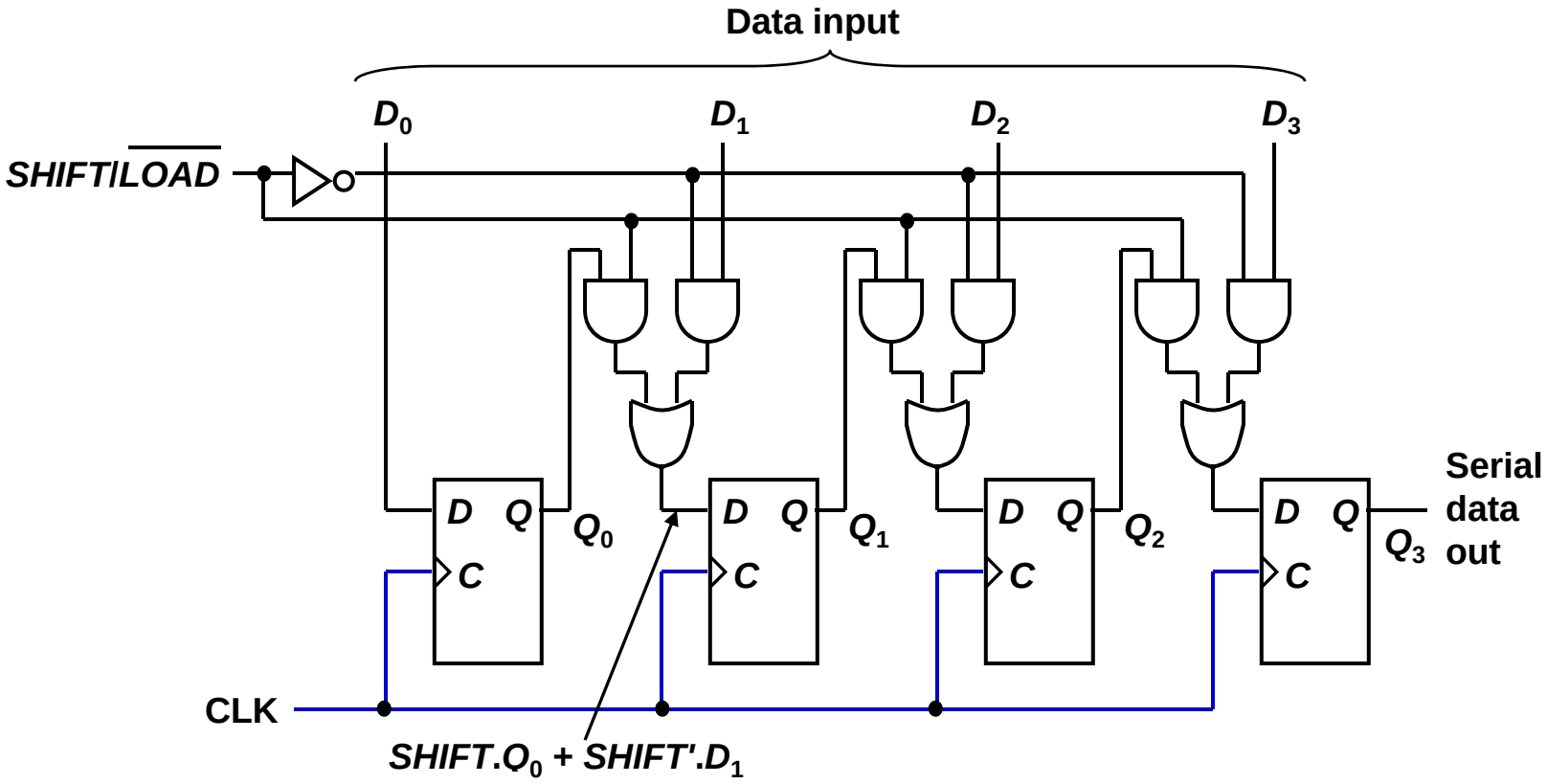




Logic symbol

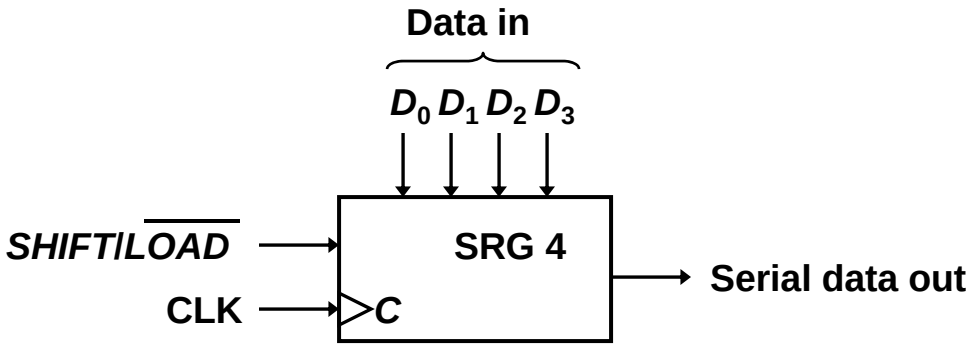
Parallel In/Serial Out Shift Registers

- Bits are entered simultaneously, but output is serial.



Parallel In/Serial Out Shift Registers

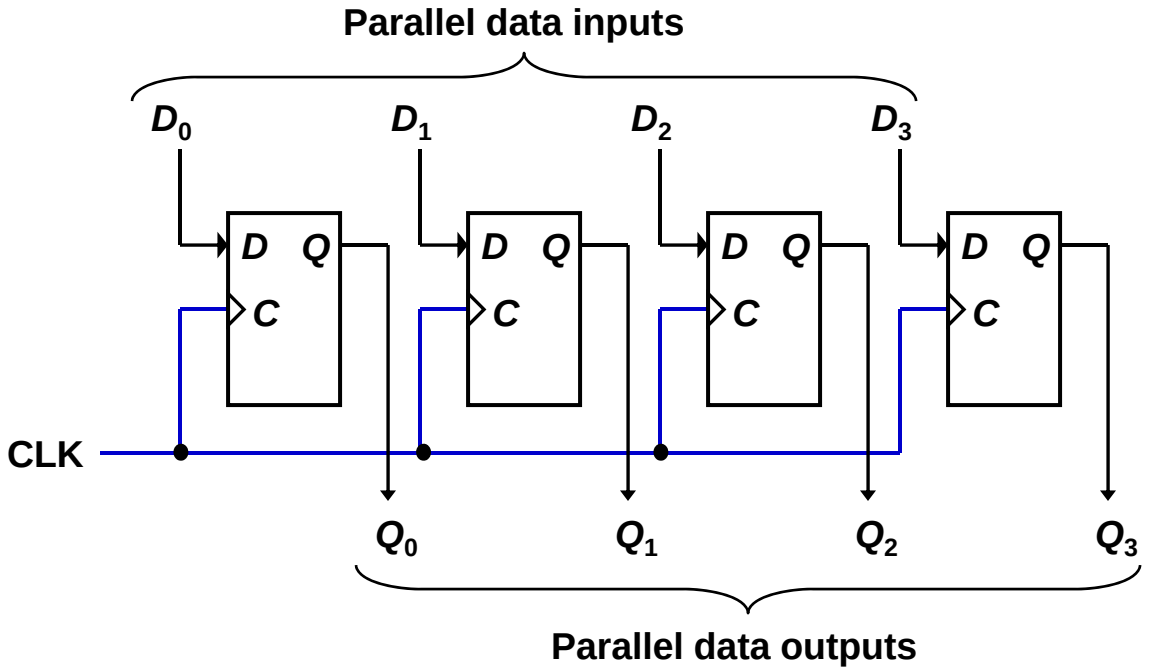
- Bits are entered simultaneously, but output is serial.



Logic symbol

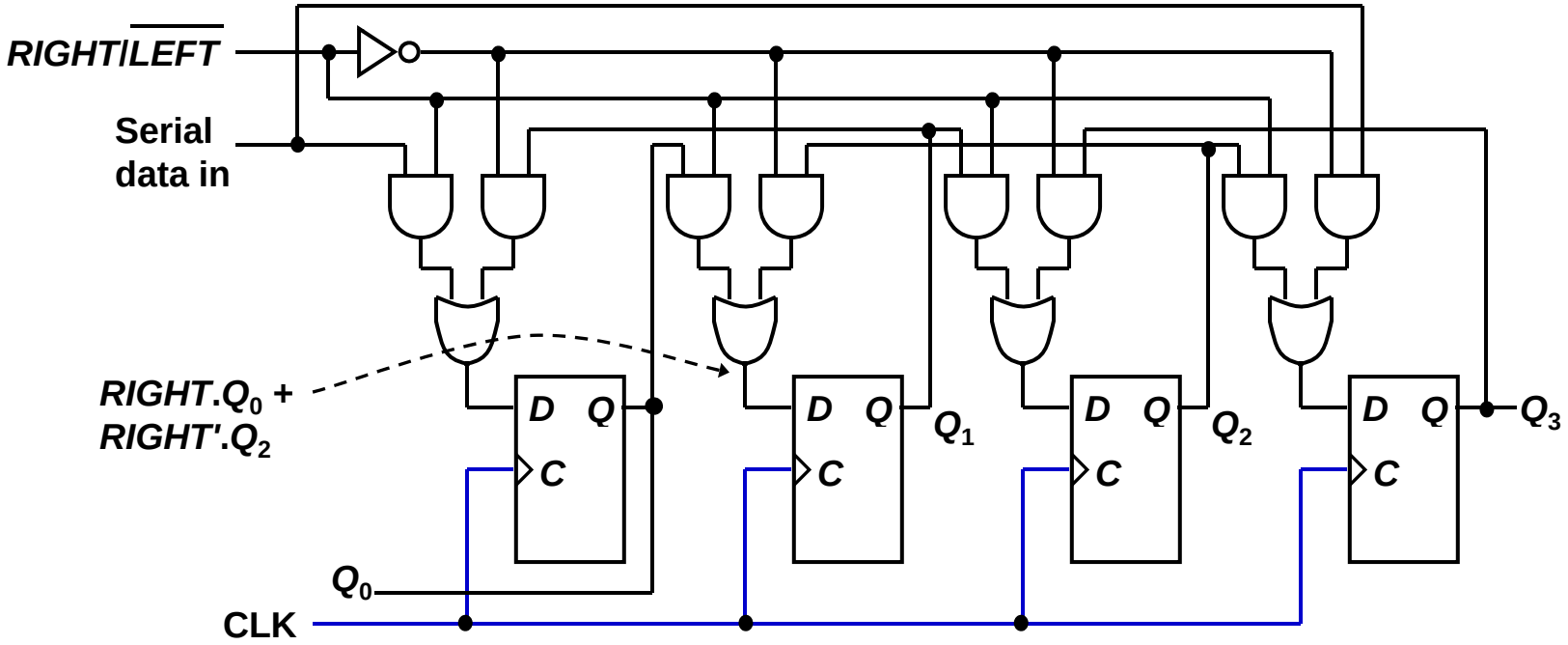
Parallel In/Parallel Out Shift Registers

- Simultaneous input and output of all data bits.



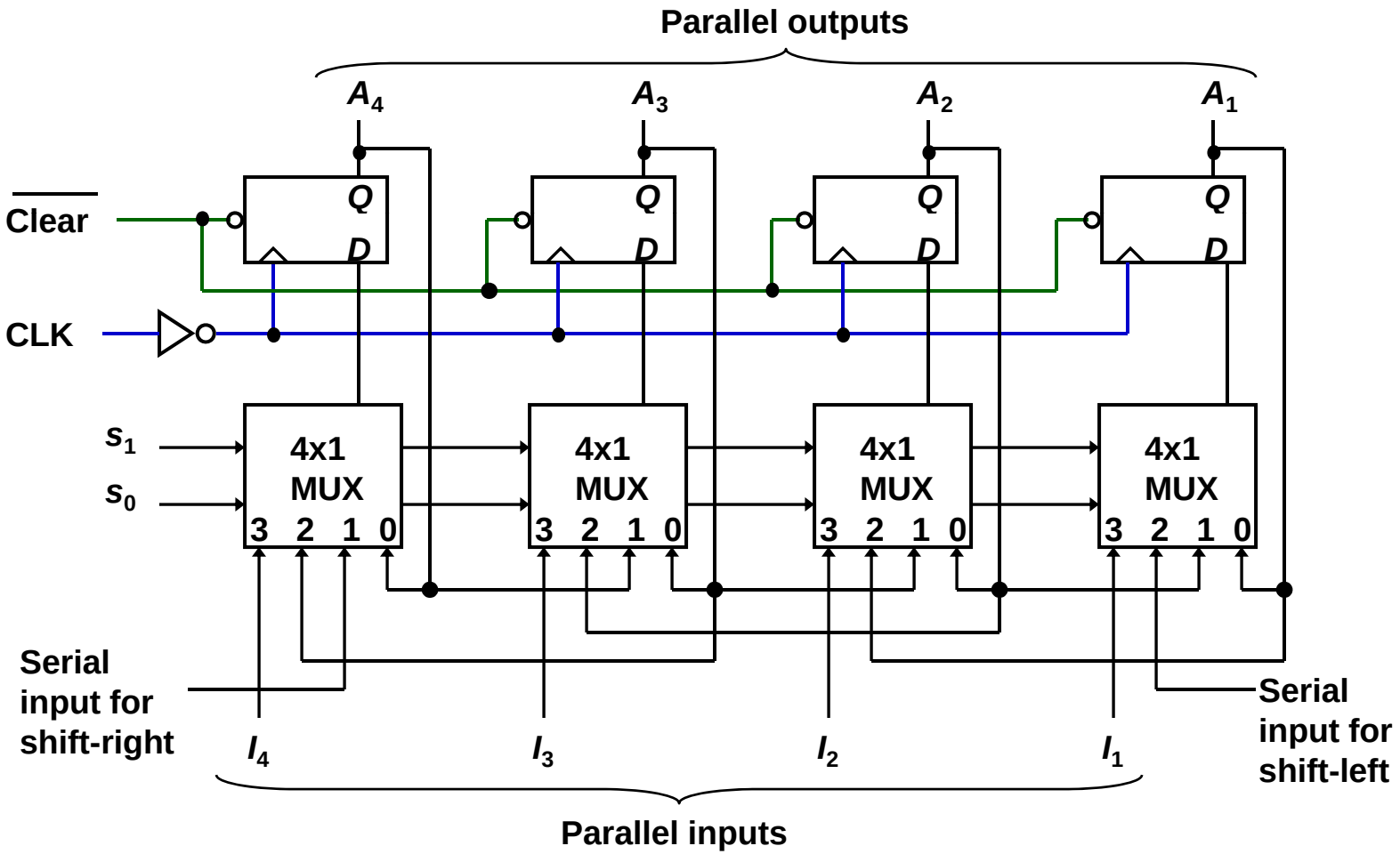
Bidirectional Shift Registers

- Data can be shifted either left or right, using a control line *RIGHT*/$\overline{\textit{LEFT}}$ (or simply *RIGHT*) to indicate the direction.



Bidirectional Shift Registers

- 4-bit bidirectional shift register with parallel load.



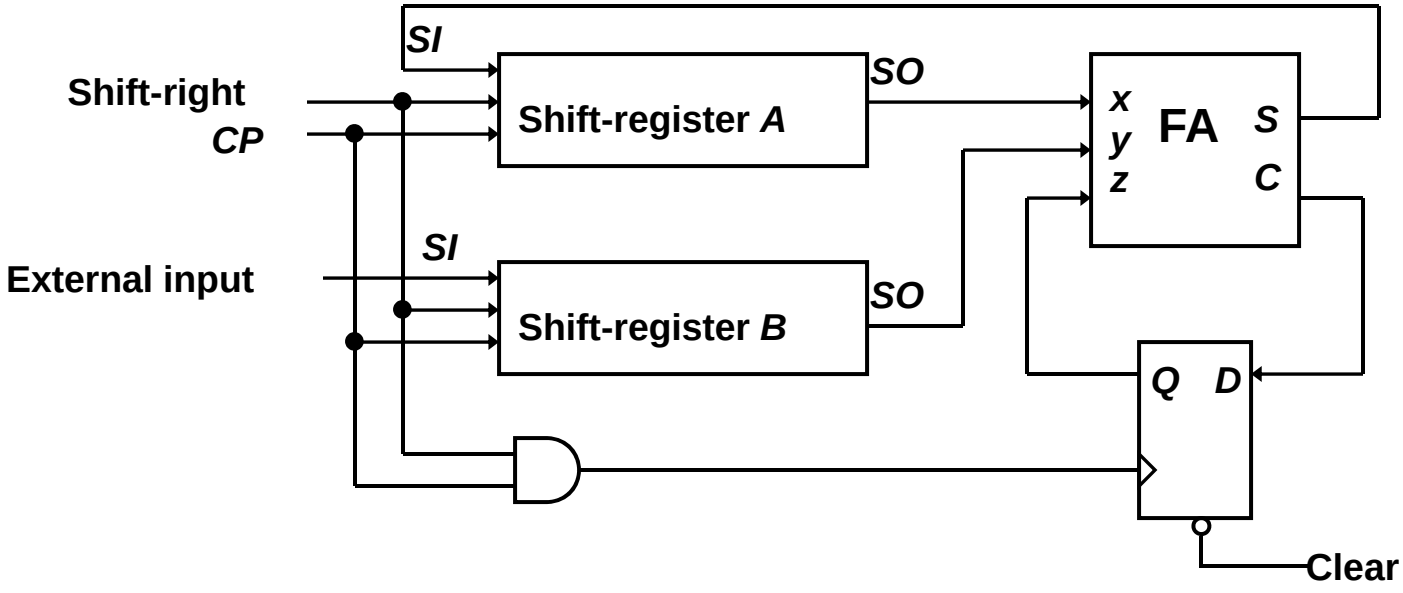
Bidirectional Shift Registers

- 4-bit bidirectional shift register with parallel load.

Mode Control		*Register Operation*
s_1	s_0	
0	0	No change
0	1	Shift right
1	0	Shift left
1	1	Parallel load

An Application – Serial Addition

- Most operations in digital computers are done in parallel. Serial operations are slower but require less equipment.
- A serial adder is shown below. $A \leftarrow A + B$.



An Application – Serial Addition

- A = 0100; B = 0111. $A + B$ = 1011 is stored in A after 4 clock pulses.

Initial:	***A*: 0 1 0 0** ***B*: 0 1 1 1**	***Q*: 0**
Step 1: 0 + 1 + 0 **S = 1, C = 0**	***A*: 1 0 1 0** ***B*: x 0 1 1**	***Q*: 0**
Step 2: 0 + 1 + 0 **S = 1, C = 0**	***A*: 1 1 0 1** ***B*: x x 0 1**	***Q*: 0**
Step 3: 1 + 1 + 0 **S = 0, C = 1**	***A*: 0 1 1 0** ***B*: x x x 0**	***Q*: 1**
Step 4: 0 + 0 + 1 **S = 1, C = 0**	***A*: 1 0 1 1** ***B*: x x x x**	***Q*: 0**

Shift Register Counters

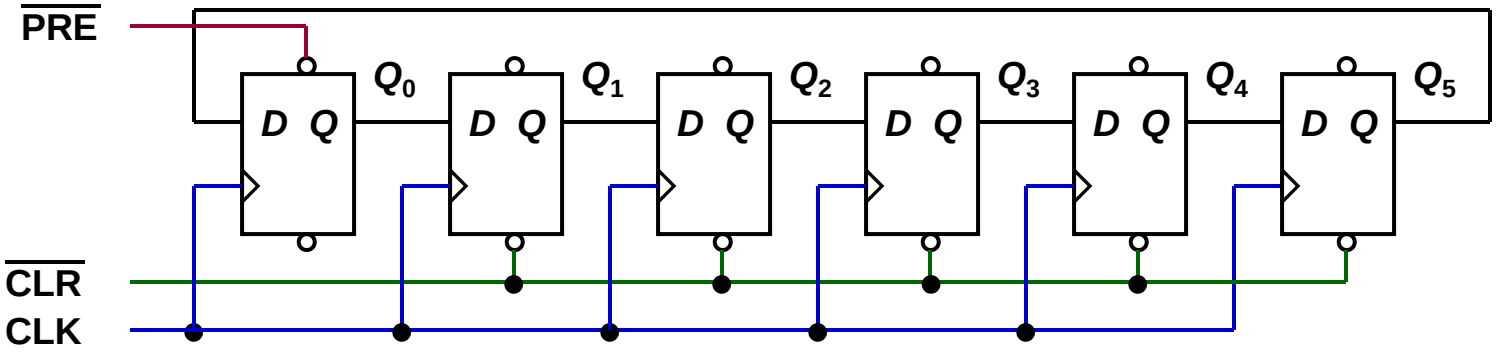
- Shift register counter: a shift register with the serial output connected back to the serial input.
- They are classified as counters because they give a specified sequence of states.
- Two common types: the *Johnson counter* and the *Ring counter*.

Ring Counters

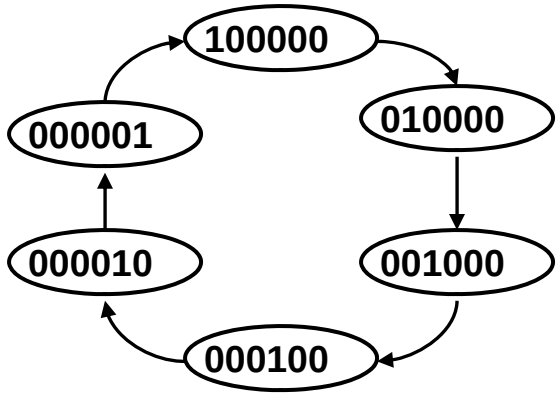
- One flip-flop (stage) for each state in the sequence.
- The output of the last stage is connected to the D input of the first stage.
- An *n*-bit ring counter cycles through *n* states.
- No decoding gates are required, as there is an output that corresponds to every state the counter is in.

Ring Counters

- Example: A 6-bit (MOD-6) ring counter.



Clock	Q_0	Q_1	Q_2	Q_3	Q_4	Q_5
0	1	0	0	0	0	0
1	0	1	0	0	0	0
2	0	0	1	0	0	0
3	0	0	0	1	0	0
4	0	0	0	0	1	0
5	0	0	0	0	0	1

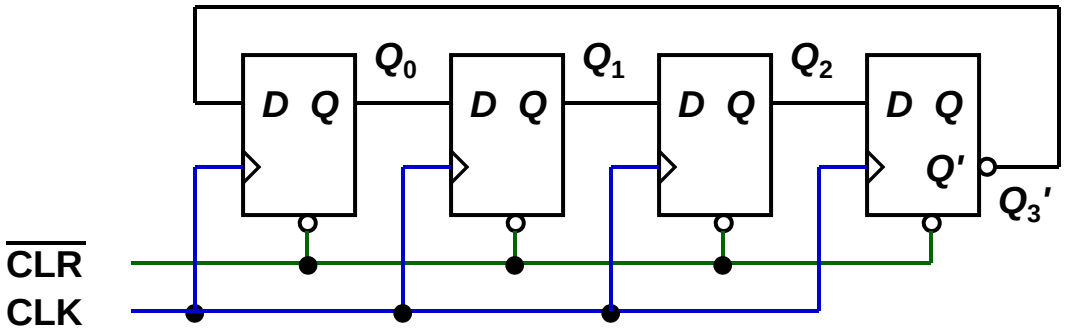


Johnson Counters

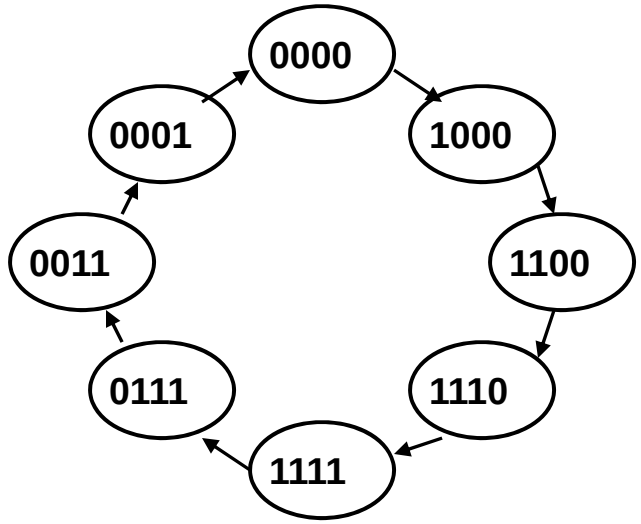
- The complement of the output of the last stage is connected back to the D input of the first stage.
- Also called the *twisted-ring counter*.
- Require fewer flip-flops than ring counters but more flip-flops than binary counters.
- An *n*-bit Johnson counter cycles through 2*n* states.
- Require more decoding circuitry than ring counter but less than binary counters.

Johnson Counters

- Example: A 4-bit (MOD-8) Johnson counter.



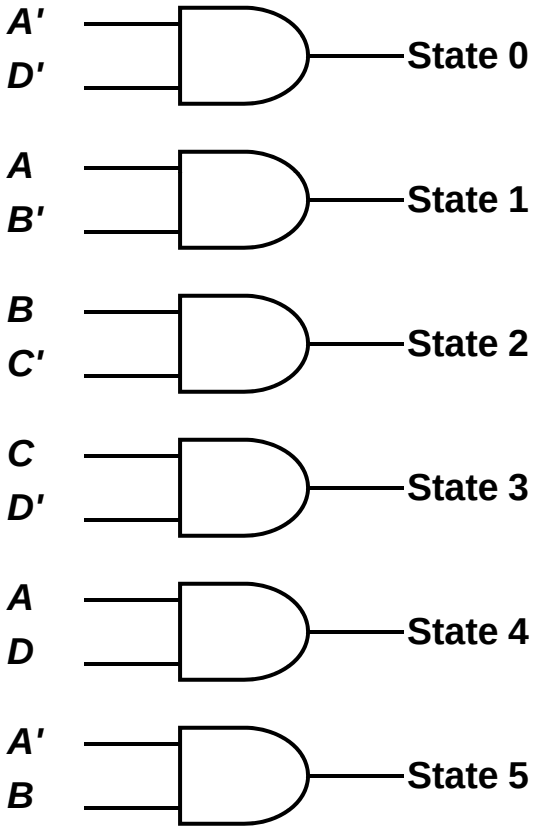
Clock	Q_0	Q_1	Q_2	Q_3
→ 0	0	0	0	0
1	1	0	0	0
2	1	1	0	0
3	1	1	1	0
4	1	1	1	1
5	0	1	1	1
6	0	0	1	1
7	0	0	0	1



Johnson Counters

- Decoding logic for a 4-bit Johnson counter.

Clock	A	B	C	D	Decoding
→ 0	0	0	0	0	A'.D'
1	1	0	0	0	A.B'
2	1	1	0	0	B.C'
3	1	1	1	0	C.D'
4	1	1	1	1	A.D
5	0	1	1	1	A'.B
6	0	0	1	1	B'.C
7	0	0	0	1	C'.D

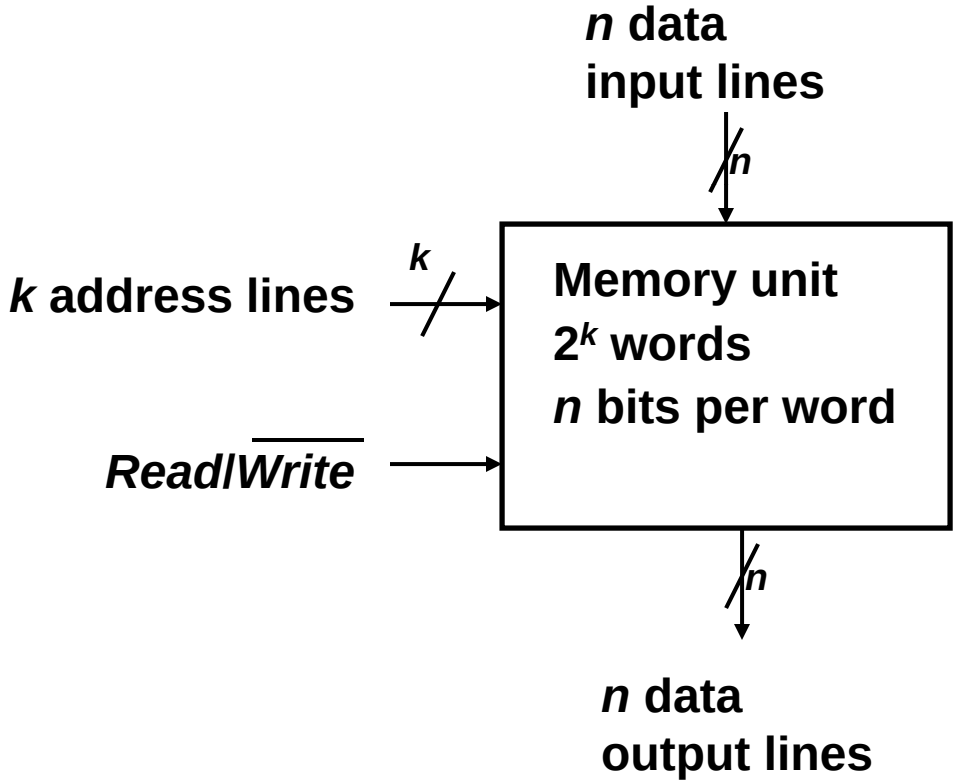


Random Access Memory (RAM)

- A memory unit stores binary information in groups of bits called *words*.
- The data consists of *n* lines (for *n*-bit words). Data input lines provide the information to be stored (*written*) into the memory, while data output lines carry the information out (*read*) from the memory.
- The address consists of *k* lines which specify which word (among the 2^k words available) to be selected for reading or writing.
- The control lines *Read* and *Write* (usually combined into a single control line *Read/Write*) specifies the direction of transfer of the data.

Random Access Memory (RAM)

- Block diagram of a memory unit:



Random Access Memory (RAM)

- Content of a 1024 x 16-bit memory:

Memory address		Memory content
binary	**decimal**	
0000000000	0	1011010111011101
0000000001	1	1010000110000110
0000000010	2	0010011101110001
:	:	:
:	:	:
1111111101	1021	1110010101010010
1111111110	1022	0011111010101110
1111111111	1023	1011000110010101

Random Access Memory (RAM)

- The Write operation:
  - ❖ Transfers the address of the desired word to the address lines
  - ❖ Transfers the data bits (the word) to be stored in memory to the data input lines
  - ❖ Activates the *Write* control line (set *Read*/*Write* to 0)
- The Read operation:
  - ❖ Transfers the address of the desired word to the address lines
  - ❖ Activates the *Read* control line (set *Read*/*Write* to 1)

Random Access Memory (RAM)

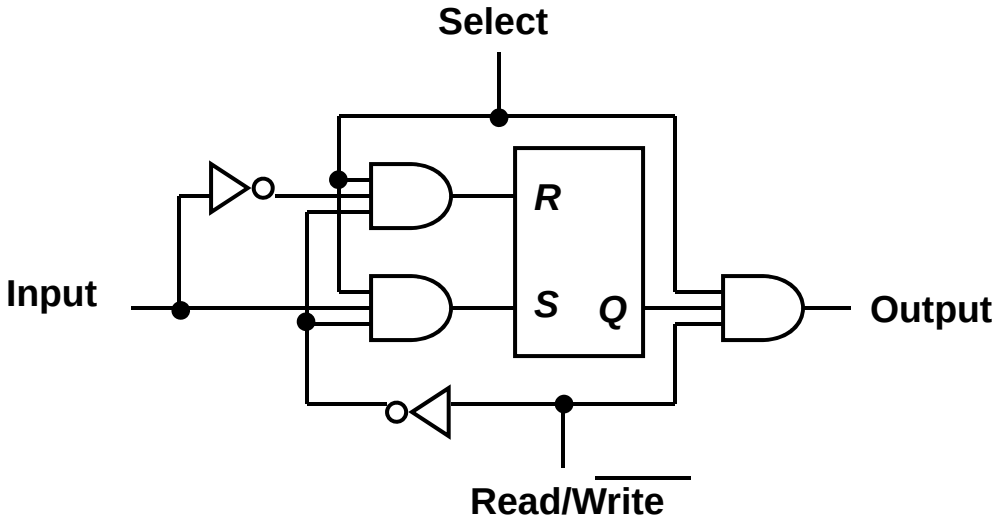
- The Read/Write operation:

Memory Enable	*Read/Write*	Memory Operation
0	X	None
1	0	Write to selected word
1	1	Read from selected word

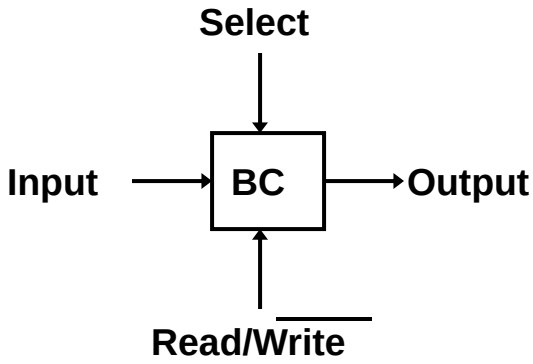
- **Two types of RAM: Static and dynamic.**
  - **Static RAMs use flip-flops as the memory cells.**
  - **Dynamic RAMs use capacitor charges to represent data. Though simpler in circuitry, they have to be constantly refreshed.**

Random Access Memory (RAM)

- A single memory cell of the static RAM has the following logic and block diagrams.



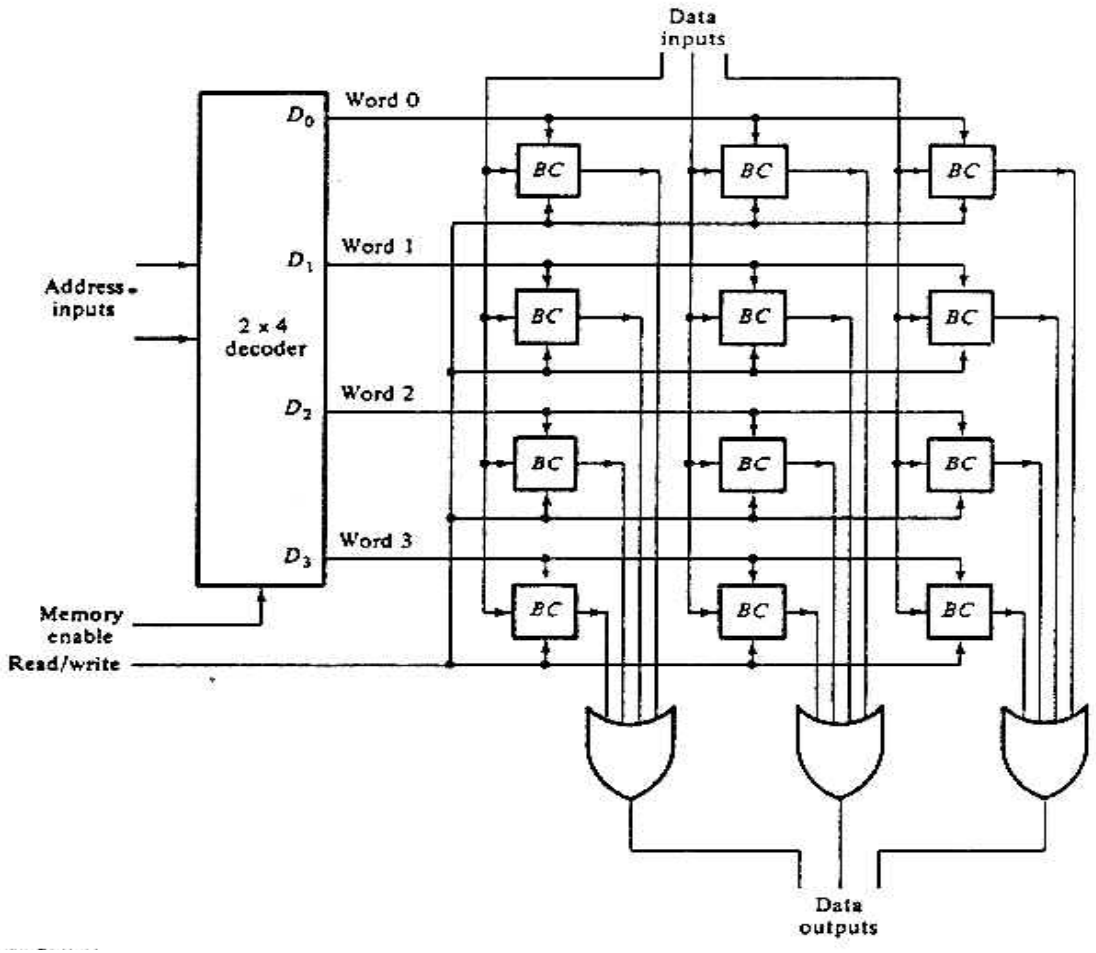
Logic diagram



Block diagram

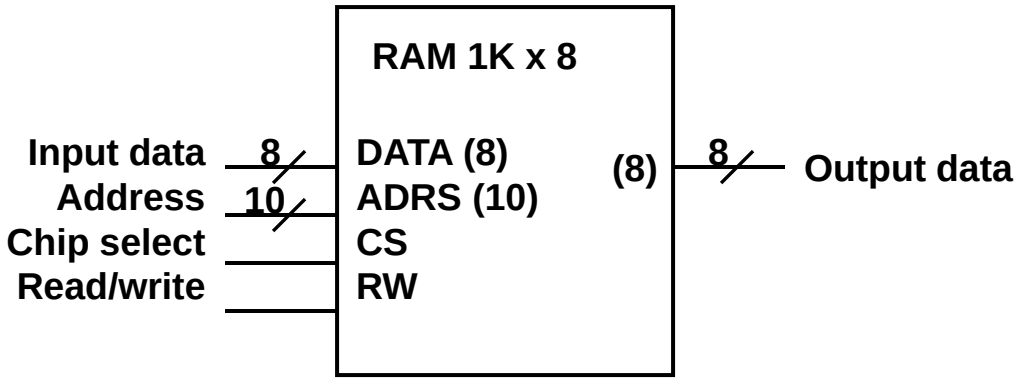
Random Access Memory (RAM)

- Logic construction of a 4 x 3 RAM (with decoder and OR gates):



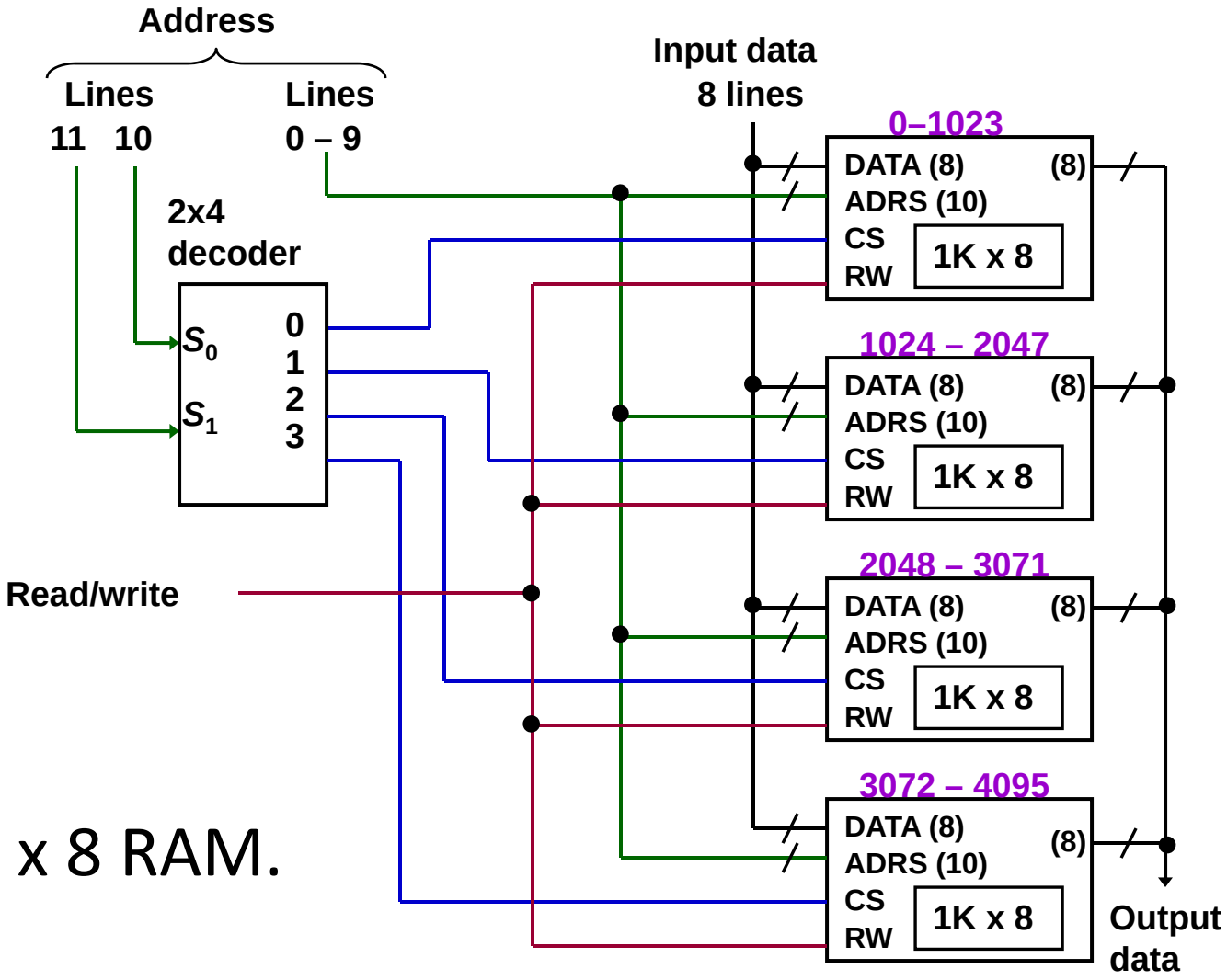
Random Access Memory (RAM)

- An array of RAM chips: memory chips are combined to form larger memory.
- A 1K x 8-bit RAM chip:



Block diagram of a 1K x 8 RAM chip

Random Access Memory (RAM)



- 4K x 8 RAM.

End of segment