Design: a mod-8 Counter

A mod-8 counter stores a integer value, and increments that value (say) on each clock tick, and wraps around to 0 if the previous stored value was 7.

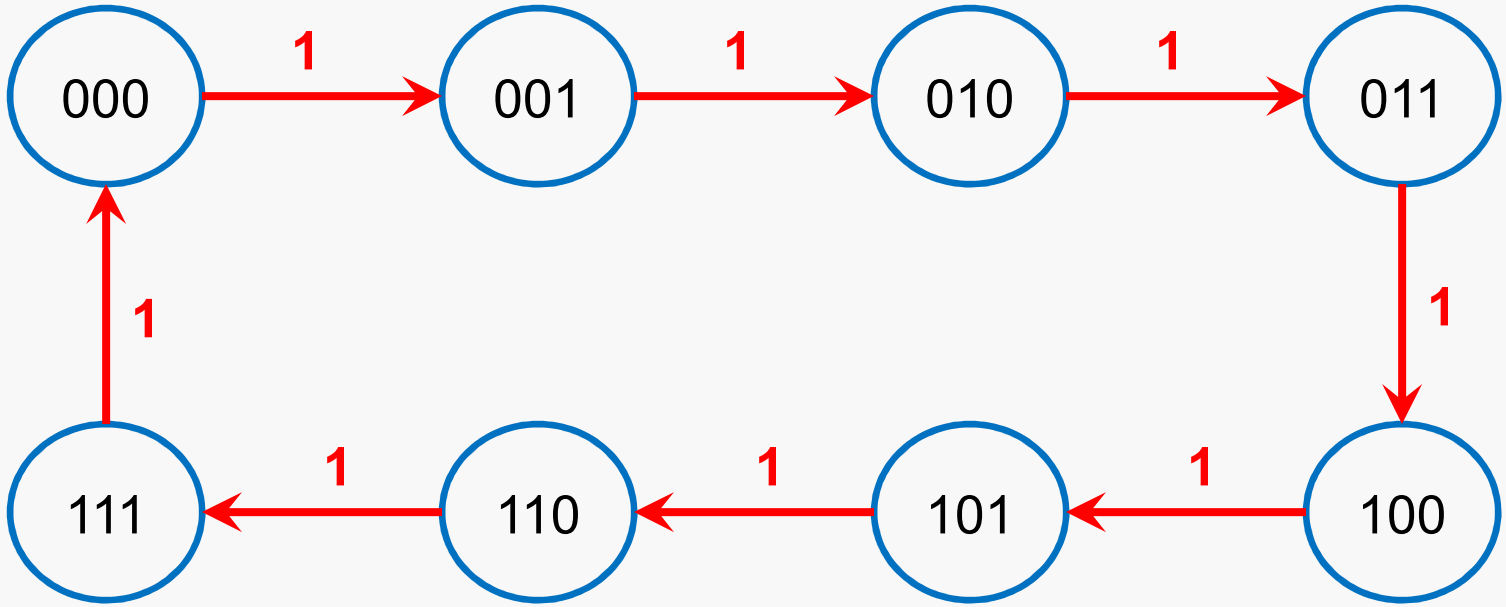
So, the stored value follows a cycle:

000 001 010 011 100 101 110 111

Design: State Machine

We need eight different states for our counter, one for each value from 0 to 7.

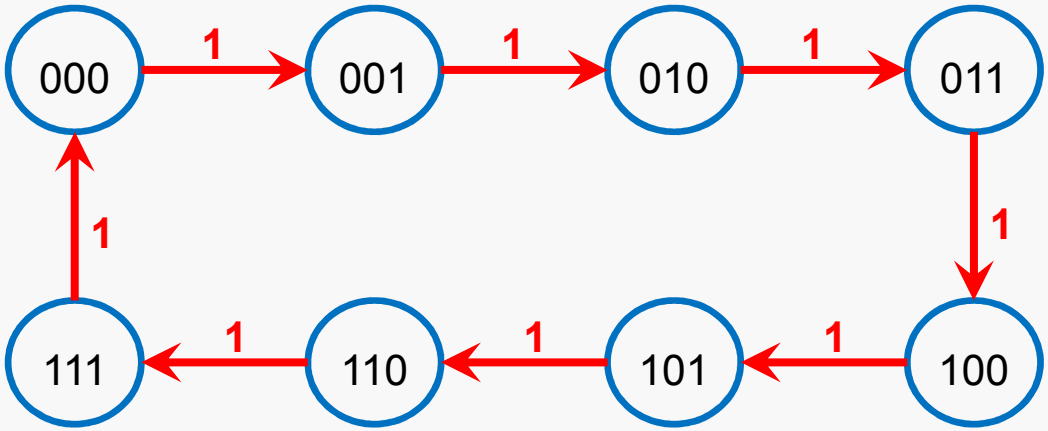
We can describe the operation by drawing a *state machine*. The nodes represent states and the edges represent transitions and are labeled with the input (clock in this case) that causes the transition to occur.



Design: State Table

A state table summarizes the state machine and is useful in deriving equations later:

Current State			Input	Next State		
0	0	0	1	0	0	1
0	0	1	1	0	1	0
0	1	0	1	0	1	1
0	1	1	1	1	0	0
1	0	0	1	1	0	1
1	0	1	1	1	1	0
1	1	0	1	1	1	1
1	1	1	1	0	0	0



Design: Deriving Equations

We will derive an equation for each of the next state functions:

Current State			Input	Next State		
C2	C1	C0		N2	N1	N0
0	0	0	1	0	0	1
0	0	1	1	0	1	0
0	1	0	1	0	1	1
0	1	1	1	1	0	0
1	0	0	1	1	0	1
1	0	1	1	1	1	0
1	1	0	1	1	1	1
1	1	1	1	0	0	0

$$N0 = \overline{C2}\cdot\overline{C1}\cdot\overline{C0} + \overline{C2}\cdot C1\cdot\overline{C0} + C2\cdot\overline{C1}\cdot\overline{C0} + C2\cdot C1\cdot\overline{C0} = \overline{C0}$$

$$N1 = \overline{C2}\cdot\overline{C1}\cdot C0 + \overline{C2}\cdot C1\cdot\overline{C0} + C2\cdot\overline{C1}\cdot C0 + C2\cdot C1\cdot\overline{C0} = \overline{C1}\cdot C0 + C1\cdot\overline{C0}$$

$$N2 = \overline{C2}\cdot C1\cdot C0 + C2\cdot\overline{C1}\cdot\overline{C0} + C2\cdot\overline{C1}\cdot C0 + C2\cdot C1\cdot\overline{C0}$$

$$= \overline{C2}\cdot C1\cdot C0 + C2\cdot\overline{C1} + C2\cdot\overline{C0}$$

Design: Mapping to D Flip-flops

Since each state is represented by a 3-bit integer, we can represent the states by using a collection of three flip-flops (more-or-less a mini-register).

We will implement the circuit using D flip-flops, which make for a simple translation from the state table because a D flip-flop simply accepts its input value.

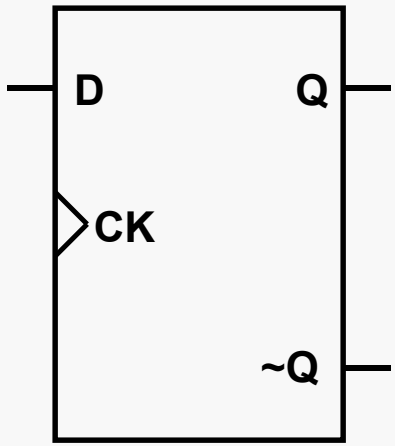
D	Q	~Q
0	0	0
0	1	0
1	0	1
1	1	1

So, we just need to feed each of the flip-flops the value of the appropriate next-state function equation derived earlier…

D Flip-flop

The D flip-flop takes one data input and updates its state Q, on a clock tick, according to the table:

D	Q	~Q
0	0	1
1	1	1

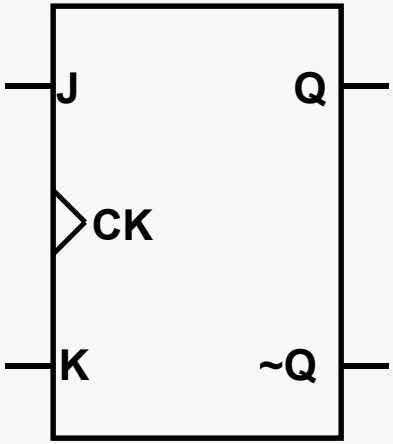


In the following Logisim diagrams, the D flip-flops update state on the falling edge (when the clock goes from high to low).

JK Flip-flop

The JK flip-flop takes two data inputs and updates its state Q, on a clock tick, according to the table:

J	K	Q	~Q
0	0	no change	
0	1	0	1
1	0	1	0
1	1	opposite	



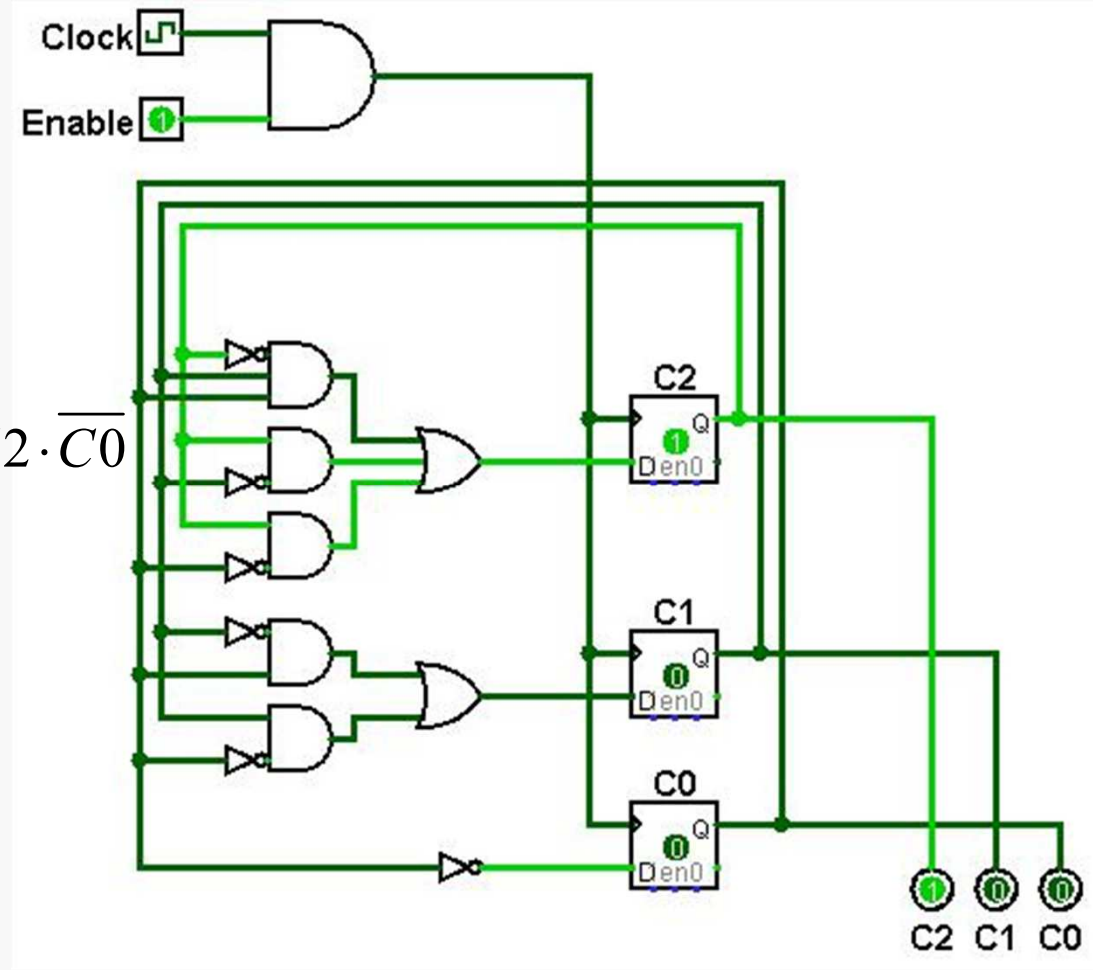
In the following Logisim diagrams, the JK flip-flops update state on the falling edge (when the clock goes from high to low).

Implementation

$$N2 = \overline{C2} \cdot C1 \cdot C0 + C2 \cdot \overline{C1} + C2 \cdot \overline{C0}$$

$$N1 = \overline{C1} \cdot C0 + C1 \cdot \overline{C0}$$

$$N0 = \overline{C0}$$



Design: Mapping to JK Flip-flops

We could also implement the circuit using JK flip-flops.

J	K	Q	~Q
0	0	no change	
0	1	0	1
1	0	1	0
1	1	opposite	

This will require a little more effort, since the inputs to the JK flip-flops cannot merely be set to equal the next state functions.

Design: Deriving JK Inputs

We must examine each current/next state pair and determine how/if the relevant flip-flop needs to change state:

Current State			Next State			Flip-flop Inputs J2	K2	J1	K1	J0	K0
0	0	0	0	0	1	0	0	0	0	1	1
0	0	1	0	1	0	0	0	1	1	1	1
0	1	0	0	1	1	0	0	0	0	1	1
0	1	1	1	0	0	1	1	1	1	1	1
1	0	0	1	0	1	0	0	0	0	1	1
1	0	1	1	1	0	0	0	1	1	1	1
1	1	0	1	1	1	0	0	0	0	1	1
1	1	1	0	0	0	1	1	1	1	1	1

For this simple circuit, we either want the JK flip-flop to hold state (both inputs 0) or to toggle state (both inputs 1).

Design: Deriving JK Input Equations

We can derive equations in the usual manner from the previous table:

Current State			Flip-flop Inputs					
			J2	K2	J1	K1	J0	K0
C2	C1	C0						
0	0	0	0	0	0	0	1	1
0	0	1	0	0	1	1	1	1
0	1	0	0	0	0	0	1	1
0	1	1	1	1	1	1	1	1
1	0	0	0	0	0	0	1	1
1	0	1	0	0	1	1	1	1
1	1	0	0	0	0	0	1	1
1	1	1	1	1	1	1	1	1

$$J0 = K0 = 1$$

$$J1 = K1 = \overline{C2} \cdot \overline{C1} \cdot C0 + \overline{C2} \cdot C1 \cdot C0 + C2 \cdot \overline{C1} \cdot C0 + C2 \cdot C1 \cdot C0$$

$$= C0$$

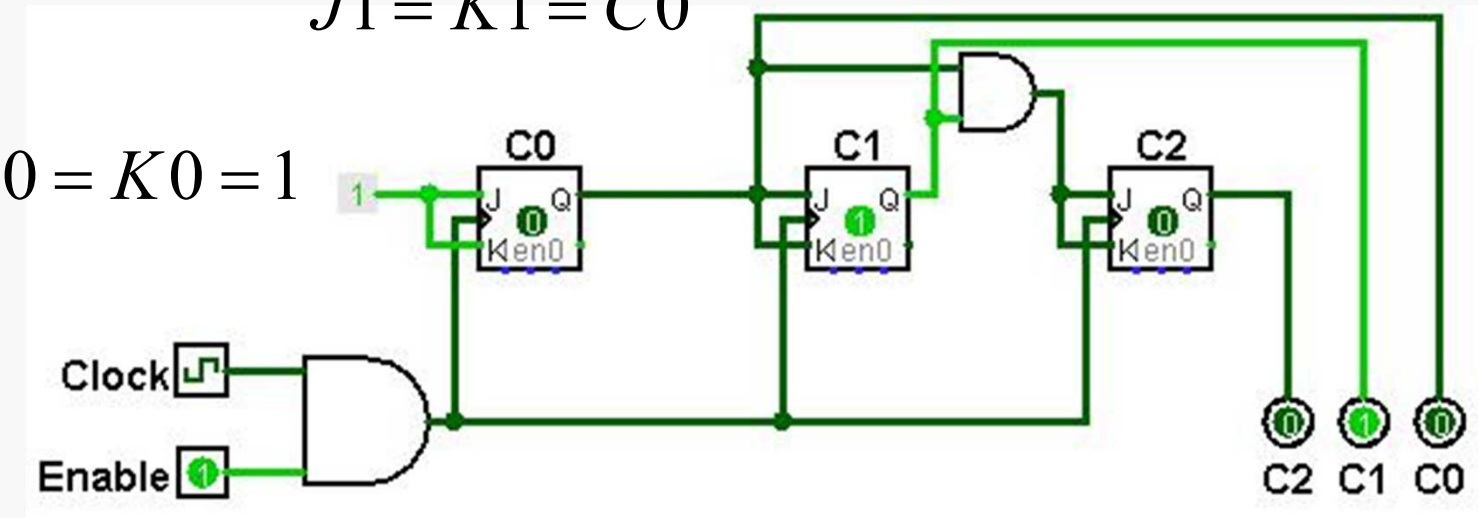
$$J2 = K2 = \overline{C2} \cdot C1 \cdot C0 + C2 \cdot C1 \cdot C0 = C1 \cdot C0$$

Implementation

$$J2 = K2 = C1 \cdot C0$$

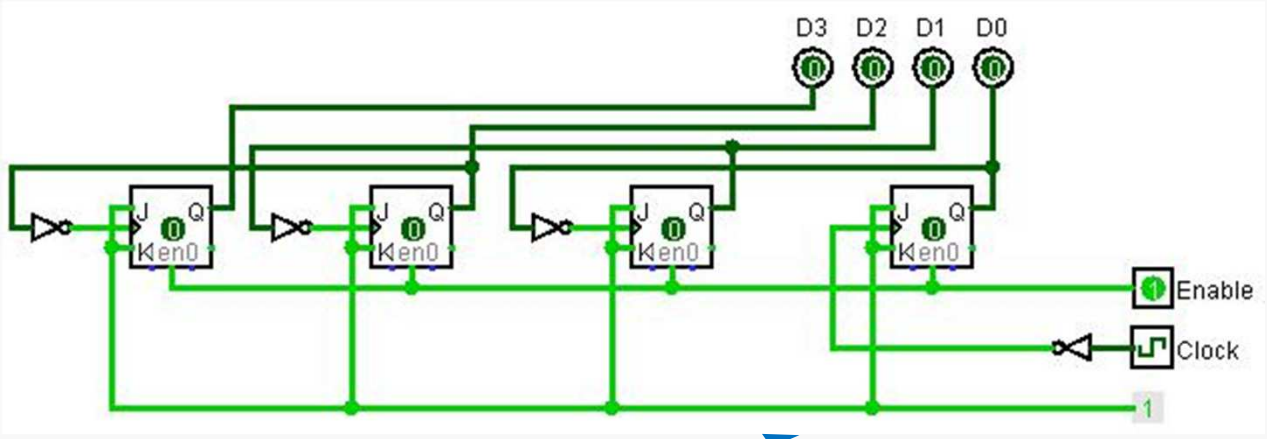
$$J1 = K1 = C0$$

$$J0 = K0 = 1$$



A mod-16 Counter

We can use JK flip-flops to implement a 4-bit counter:



Note that the J and K inputs are all set to the fixed value 1, so the flip-flops "toggle".

As the clock signal runs, the circuit will cycle its outputs through the values

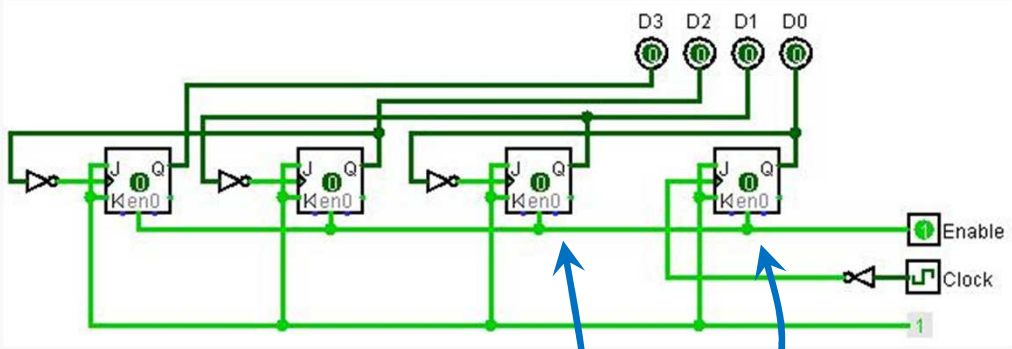
0000, 0001, 0010, . . . , 1111

and then repeat the pattern.

So, it counts clock ticks, modulo 16.

mod-16 Counter: first tick

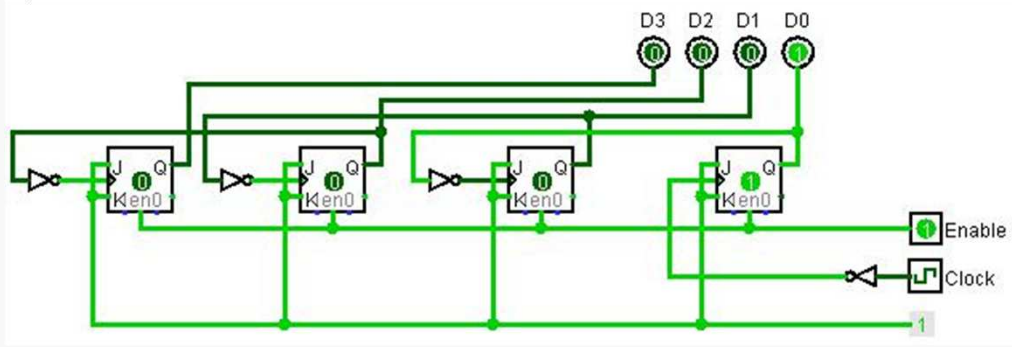
Suppose the counter is in the initial state shown below (output is 0000).



When the clock cycles from high to low:

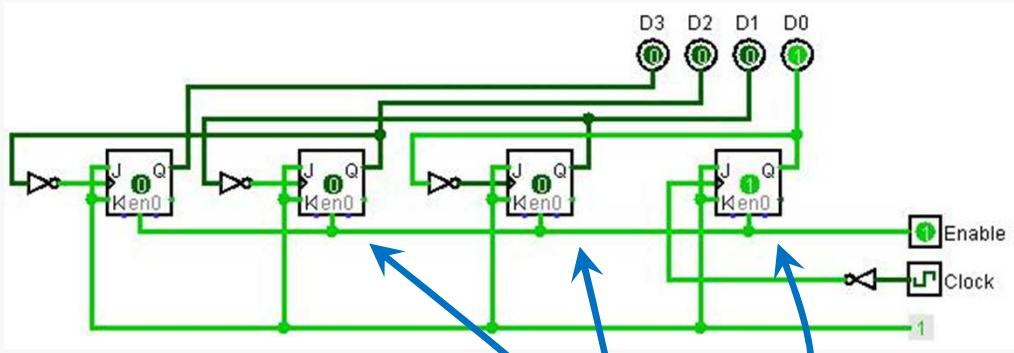
- the right-most sees its (inverted) clock signal go from low to high, and so it toggles its state to 1
- the next flip-flop sees its clock signal go from high to low, and so it doesn't toggle
- and so, neither do the other flip-flops…

So, the output is 0001:



mod-16 Counter: second tick

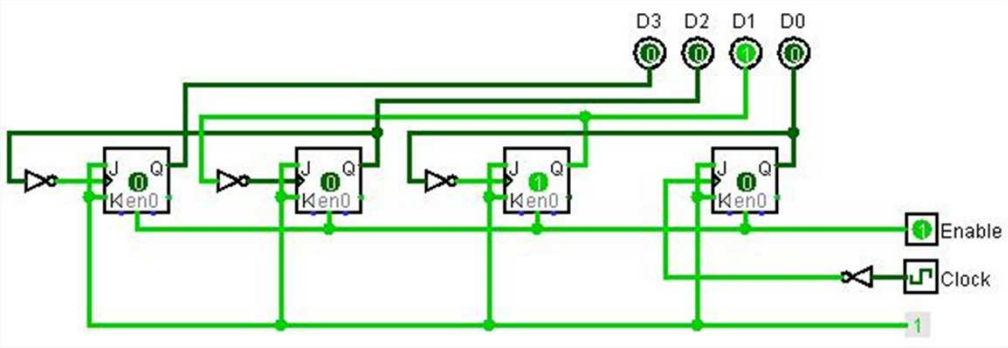
Suppose the counter is now in the state shown below (output is 0001).



When the clock cycles from high to low (2nd cycle):

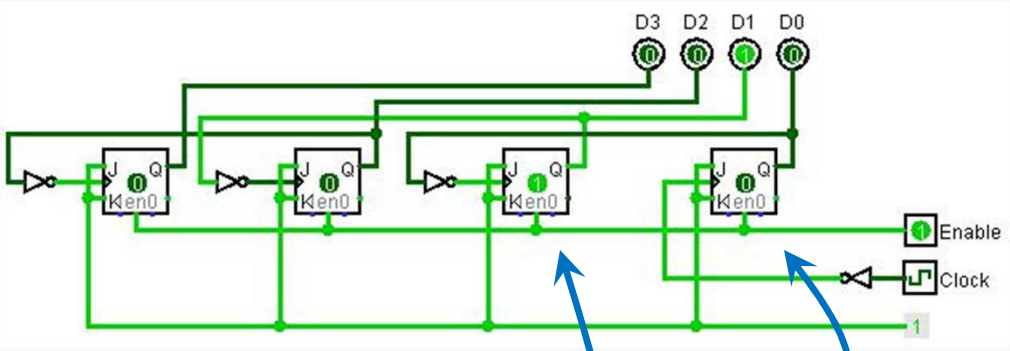
- the right-most sees its (inverted) clock signal go from low to high, and so it toggles its state to 0
- the next flip-flop sees its clock signal go from low to high, and so it toggles its state to 1
- the next flip-flop sees its clock signal go from high to low, so it doesn't toggle

So the output is 0010.



mod-16 Counter: third tick

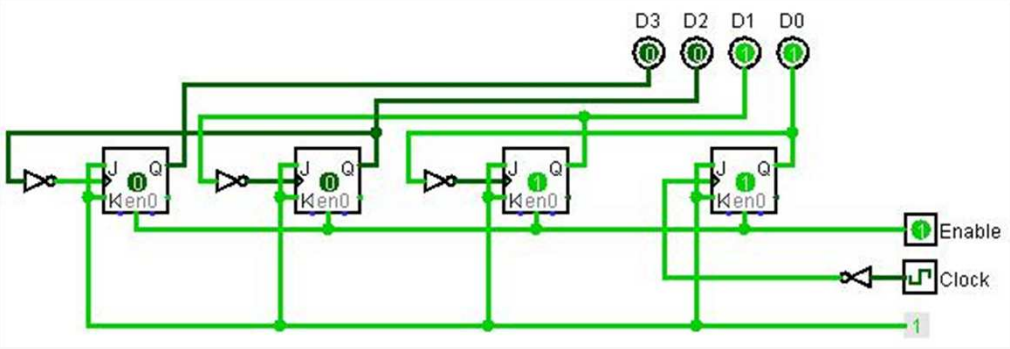
Suppose the counter is now in the state shown below (output is 0010).



When the clock cycles from high to low (3rd cycle):

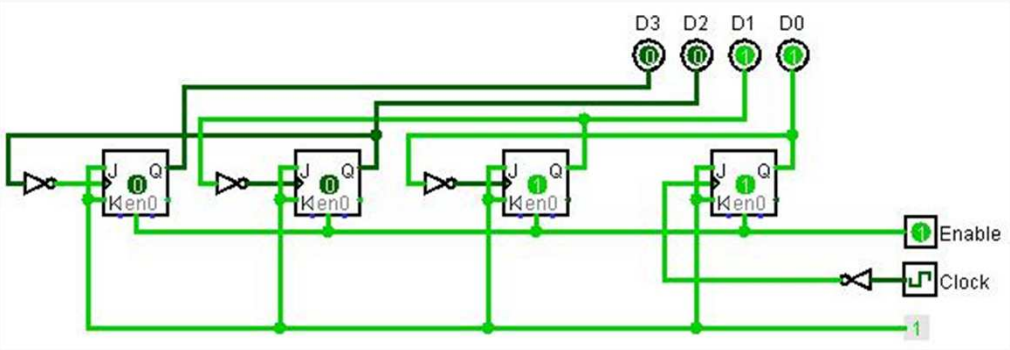
- the right-most sees its (inverted) clock signal go from low to high, and so it toggles its state to 1
- the next flip-flop sees its clock signal go from high to low, and so it doesn't toggle

So the output is 0011.



mod-16 Counter: fourth tick

Suppose the counter is now in the state shown below (output is 0011).



When the clock cycles from high to low (4th cycle):

- the right-most sees its (inverted) clock signal go from low to high, and so it toggles its state to 0
- the next flip-flop sees its clock signal go from low to high, and so it toggles its state to 0
- the next flip-flop sees its clock signal go from low to high, and so it toggles its state to 1

So the output is 0100.