From latches to flip-flops
COMSM1302 Overview of Computer Architecture

John Lapinskas, University of Bristol

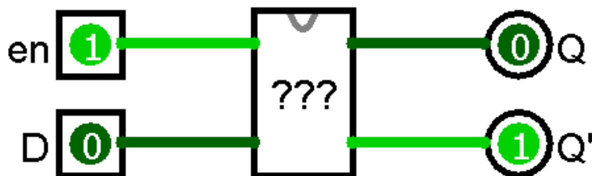
Building a better latch

The R-S latch is nice, but it has problems:

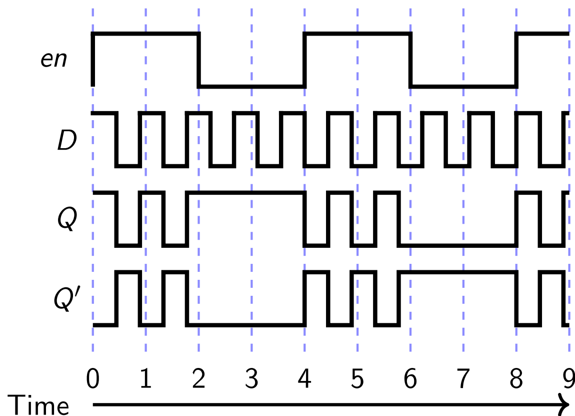
- We have to be careful not to activate set and reset at the same time.
- Timing issues would be much easier to think about if we could choose *what* the next state will be separately from *when* it will change.
- Things get messy if we loop the output back to the inputs!

Let's solve the first two problems first, with a **D latch**. We'll solve the third later with a **D flip-flop**.

[Demonstration of D latch behaviour in Logisim — see video.]



Desired behaviour



en	D	Q	Q'
0	0	"Hold"	"Hold"
0	1	"Hold"	"Hold"
1	0	0	1
1	1	1	0

en is an active high input. When it's active, Q takes the value of D. Otherwise, Q stays constant.

We maintain $Q' = \neg Q$.

Can you see how to build a D latch from an R-S latch?

The D-latch

We can go back to basics: combinatorial logic, using *en* and D to specify inputs for an R-S latch!

en	D	Q	Q'
0	0	“Hold”	“Hold”
0	1	“Hold”	“Hold”
1	0	0	1
1	1	1	0

R'	S'	Q	Q'
0	0	X	X
0	1	0	1
1	0	1	0
1	1	“Hold”	“Hold”

The D-latch

We can go back to basics: combinatorial logic, using *en* and *D* to specify inputs for an R-S latch!

en	D	R'	S'	Q	Q'
0	0	1	1	"Hold"	"Hold"
0	1	1	1	"Hold"	"Hold"
1	0	0	1	0	1
1	1	1	0	1	0

The D-latch

We can go back to basics: combinatorial logic, using *en* and D to specify inputs for an R-S latch!

en	D	R'	S'	Q	Q'
0	0	1	1	"Hold"	"Hold"
0	1	1	1	"Hold"	"Hold"
1	0	0	1	0	1
1	1	1	0	1	0

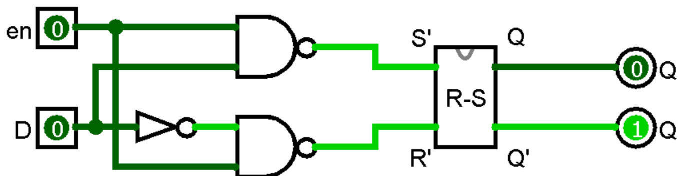
We can write this as $S' = \neg(en \wedge D)$ and $R' = \neg(en \wedge \neg D)$, so...

The D-latch

We can go back to basics: combinatorial logic, using *en* and D to specify inputs for an R-S latch!

en	D	R'	S'	Q	Q'
0	0	1	1	"Hold"	"Hold"
0	1	1	1	"Hold"	"Hold"
1	0	0	1	0	1
1	1	1	0	1	0

We can write this as $S' = \neg(en \wedge D)$ and $R' = \neg(en \wedge \neg D)$, so...

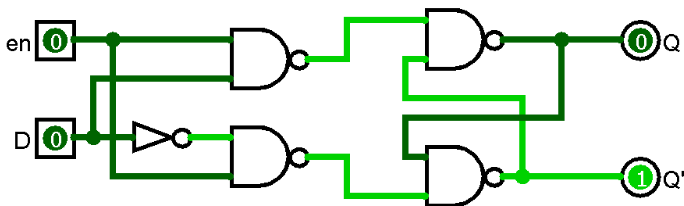


The D-latch

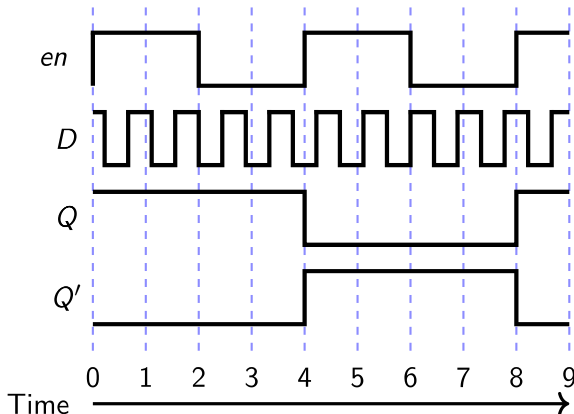
We can go back to basics: combinatorial logic, using *en* and D to specify inputs for an R-S latch!

en	D	R'	S'	Q	Q'
0	0	1	1	"Hold"	"Hold"
0	1	1	1	"Hold"	"Hold"
1	0	0	1	0	1
1	1	1	0	1	0

We can write this as $S' = \neg(en \wedge D)$ and $R' = \neg(en \wedge \neg D)$, so...



Level-triggering and edge-triggering

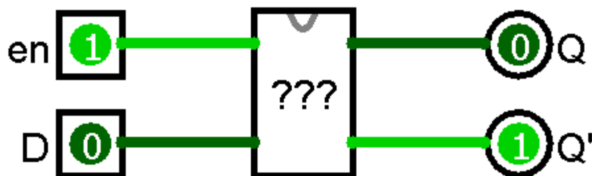


A D latch is **level-triggered** — for as long as *en* stays high, any change to *D* will be reflected in the output.

This can work, if we are very careful about how long *en* stays high for. But it would be much easier to use if it were **edge-triggered**: if *Q* took on the value of *D only* at the exact moment *en* went high.

(This is **positive** edge triggering. If the trigger was *en* going low instead of high, it would be **negative** edge triggering.)

[Demonstration of D flip-flop behaviour in Logisim — see video.]

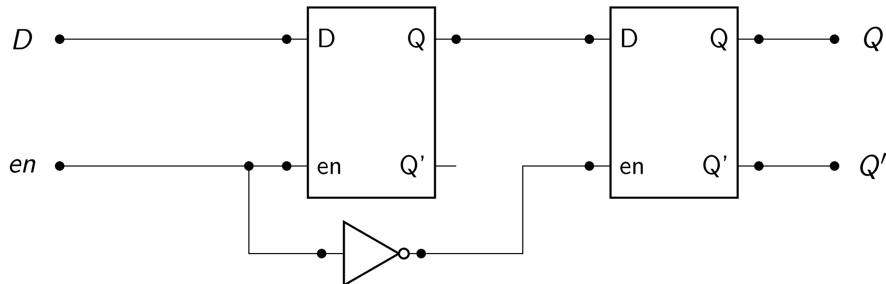


How a D flip-flop works

Negative edge-triggering is easier, so let's start with that.

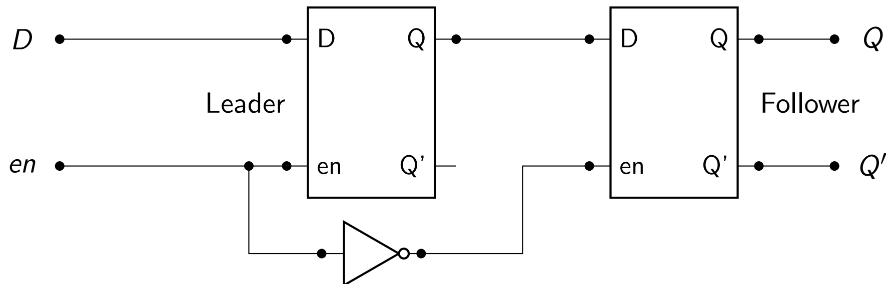
How a D flip-flop works

Negative edge-triggering is easier, so let's start with that.



How a D flip-flop works

Negative edge-triggering is easier, so let's start with that.

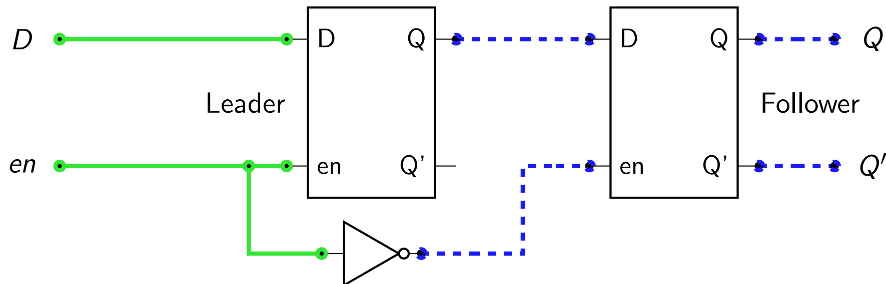


The left D latch is sometimes called the **leader** or **primary**, and the right is sometimes called the **follower** or **secondary**.

Historically the left latch was called the **master** and the right latch the **slave**, but these terms have unpleasant connotations and are losing popularity.

How a D flip-flop works

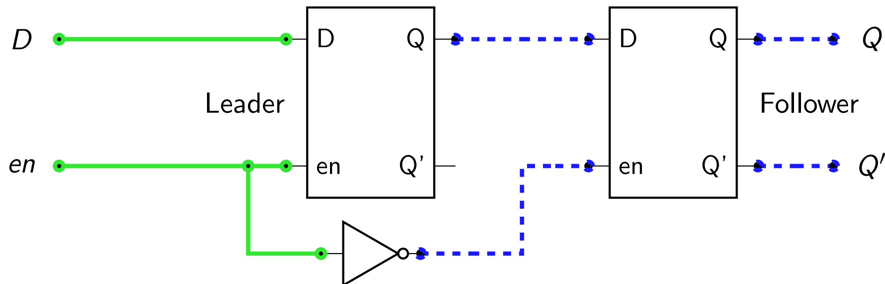
Negative edge-triggering is easier, so let's start with that.



Let's say (for example) that *en* starts high.

How a D flip-flop works

Negative edge-triggering is easier, so let's start with that.

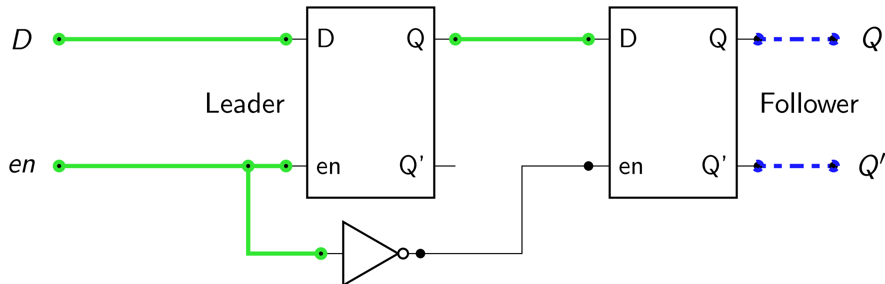


Let's say (for example) that *en* starts high.

Then the leader must output D, while the follower is disabled (so its output is unaffected).

How a D flip-flop works

Negative edge-triggering is easier, so let's start with that.

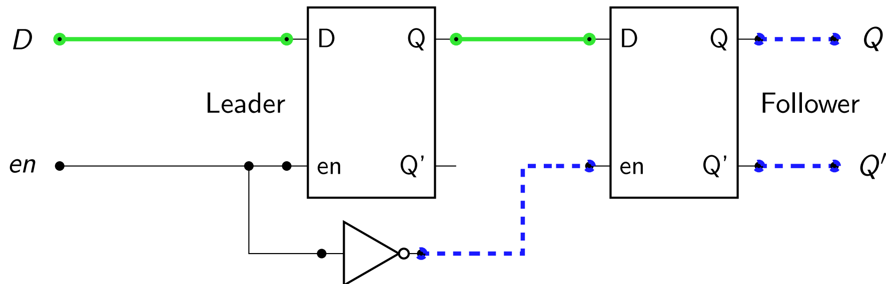


Let's say (for example) that *en* starts high.

Then the leader must output D, while the follower is disabled (so its output is unaffected).

How a D flip-flop works

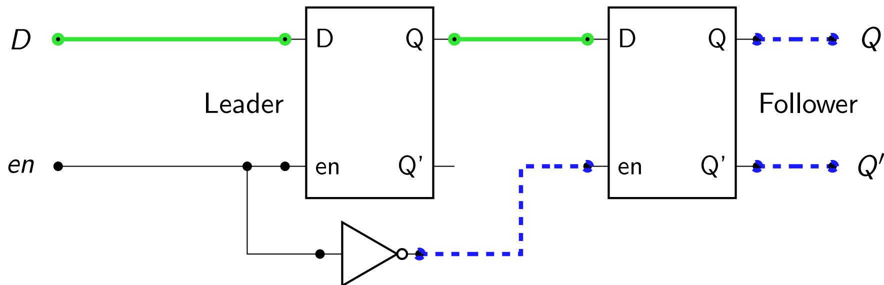
Negative edge-triggering is easier, so let's start with that.



Now suppose *en* falls low.

How a D flip-flop works

Negative edge-triggering is easier, so let's start with that.

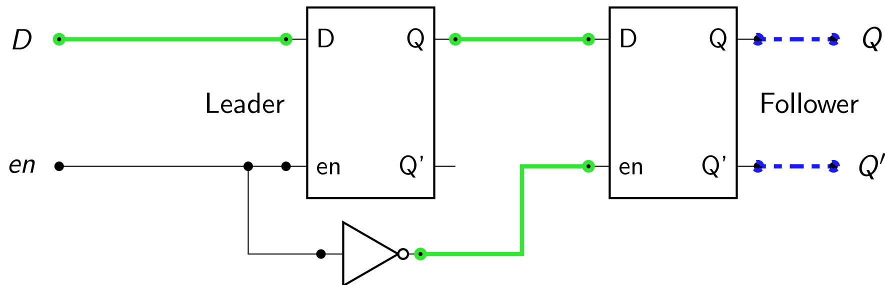


Now suppose *en* falls low.

Then the follower's *en* input goes high, so it takes on *D*'s *current* value.

How a D flip-flop works

Negative edge-triggering is easier, so let's start with that.

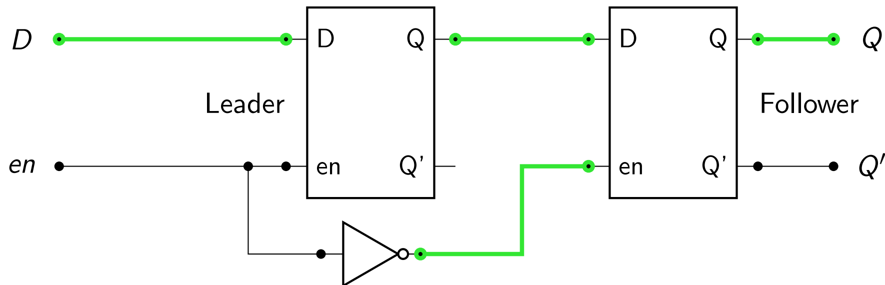


Now suppose *en* falls low.

Then the follower's *en* input goes high, so it takes on *D*'s *current* value.

How a D flip-flop works

Negative edge-triggering is easier, so let's start with that.

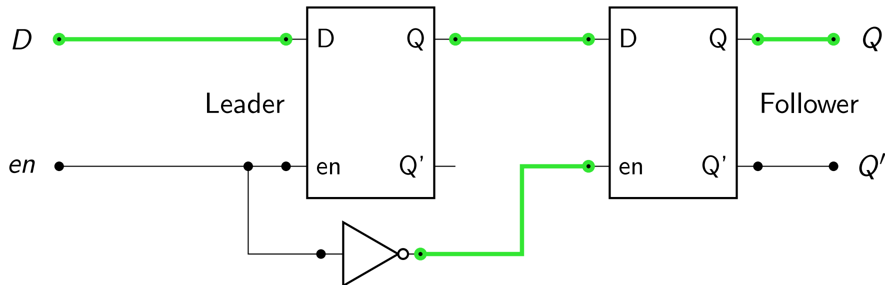


Now suppose *en* falls low.

Then the follower's *en* input goes high, so it takes on *D*'s *current* value.

How a D flip-flop works

Negative edge-triggering is easier, so let's start with that.



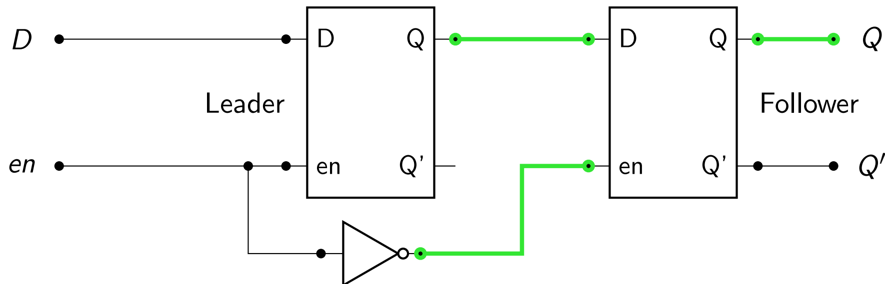
Now suppose *en* falls low.

Then the follower's *en* input goes high, so it takes on *D*'s *current* value.

Subsequent changes to *D* have no effect, since *en* is low.

How a D flip-flop works

Negative edge-triggering is easier, so let's start with that.



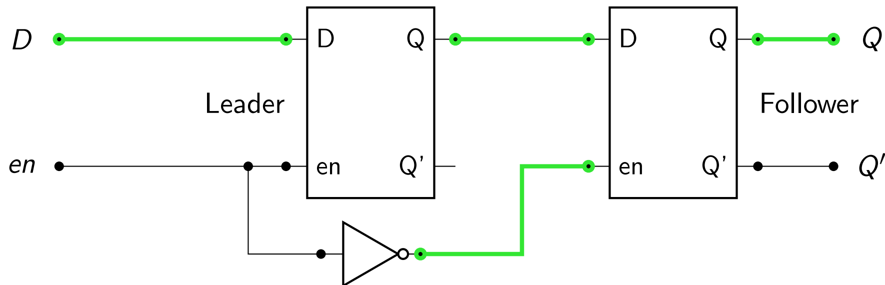
Now suppose *en* falls low.

Then the follower's *en* input goes high, so it takes on *D*'s *current* value.

Subsequent changes to *D* have no effect, since *en* is low.

How a D flip-flop works

Negative edge-triggering is easier, so let's start with that.



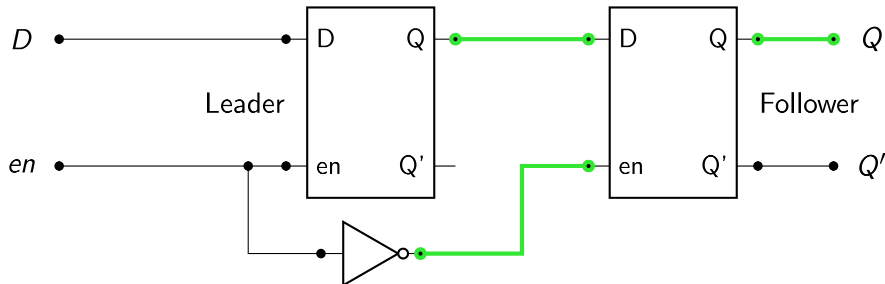
Now suppose *en* falls low.

Then the follower's *en* input goes high, so it takes on *D*'s *current* value.

Subsequent changes to *D* have no effect, since *en* is low.

How a D flip-flop works

Negative edge-triggering is easier, so let's start with that.



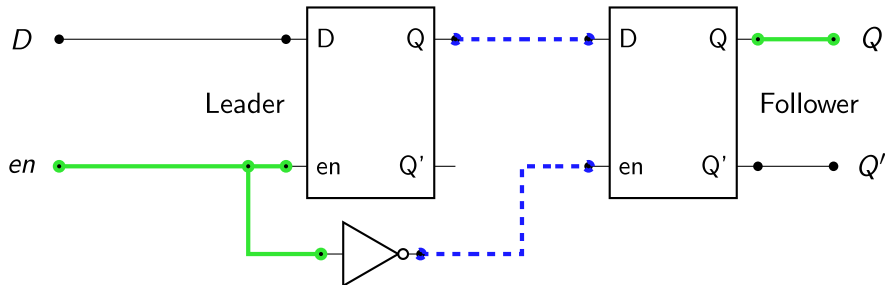
Now suppose *en* falls low.

Then the follower's *en* input goes high, so it takes on *D*'s *current* value.

Subsequent changes to *D* have no effect, since *en* is low.

How a D flip-flop works

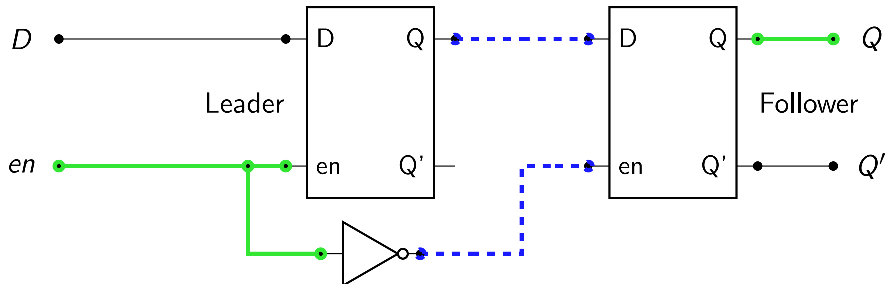
Negative edge-triggering is easier, so let's start with that.



Now suppose *en* rises again.

How a D flip-flop works

Negative edge-triggering is easier, so let's start with that.



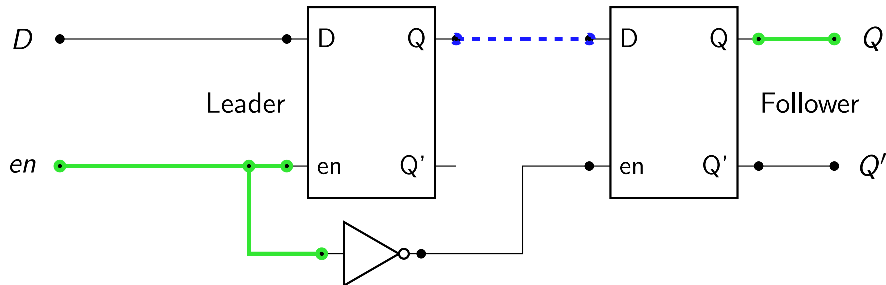
Now suppose *en* rises again.

Crucially, even if *D* has changed, this new signal will propagate through the NOT gate faster than through the leader.

So the follower's *en* input goes low *before* the leader's output finishes updating, and the follower's output is unchanged.

How a D flip-flop works

Negative edge-triggering is easier, so let's start with that.



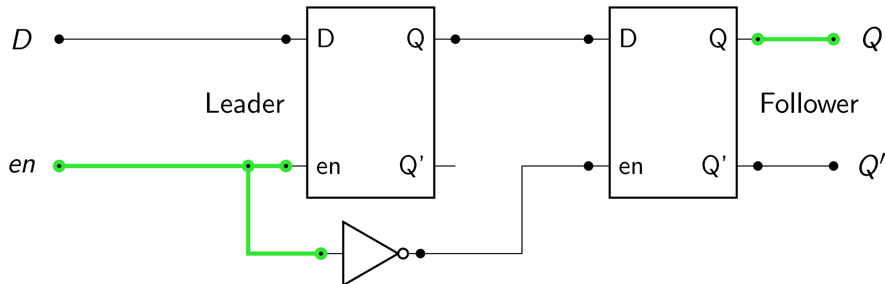
Now suppose *en* rises again.

Crucially, even if *D* has changed, this new signal will propagate through the NOT gate faster than through the leader.

So the follower's *en* input goes low *before* the leader's output finishes updating, and the follower's output is unchanged.

How a D flip-flop works

Negative edge-triggering is easier, so let's start with that.



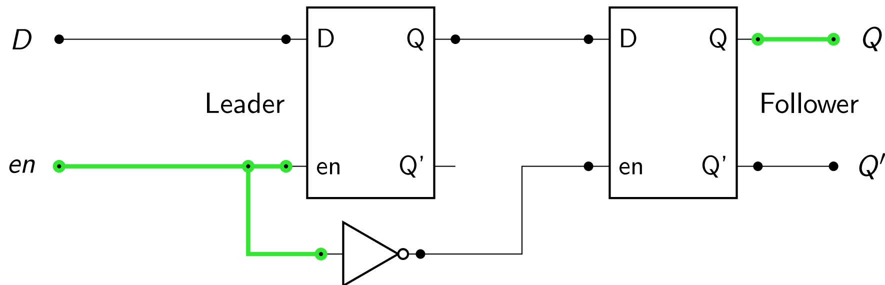
Now suppose *en* rises again.

Crucially, even if *D* has changed, this new signal will propagate through the NOT gate faster than through the leader.

So the follower's *en* input goes low *before* the leader's output finishes updating, and the follower's output is unchanged.

How a D flip-flop works

Negative edge-triggering is easier, so let's start with that.



Now suppose *en* rises again.

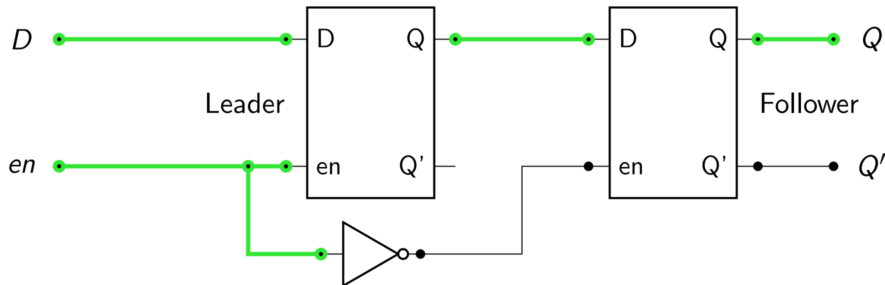
Crucially, even if *D* has changed, this new signal will propagate through the NOT gate faster than through the leader.

So the follower's *en* input goes low *before* the leader's output finishes updating, and the follower's output is unchanged.

Subsequent changes to *D* again have no effect.

How a D flip-flop works

Negative edge-triggering is easier, so let's start with that.



Now suppose *en* rises again.

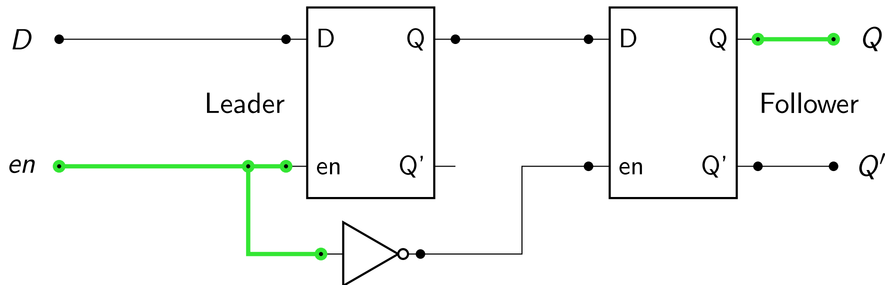
Crucially, even if *D* has changed, this new signal will propagate through the NOT gate faster than through the leader.

So the follower's *en* input goes low *before* the leader's output finishes updating, and the follower's output is unchanged.

Subsequent changes to *D* again have no effect.

How a D flip-flop works

Negative edge-triggering is easier, so let's start with that.



Now suppose *en* rises again.

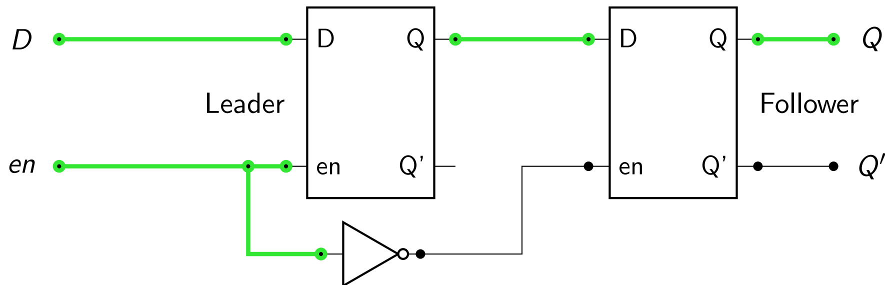
Crucially, even if *D* has changed, this new signal will propagate through the NOT gate faster than through the leader.

So the follower's *en* input goes low *before* the leader's output finishes updating, and the follower's output is unchanged.

Subsequent changes to *D* again have no effect.

How a D flip-flop works

Negative edge-triggering is easier, so let's start with that.



Now suppose *en* rises again.

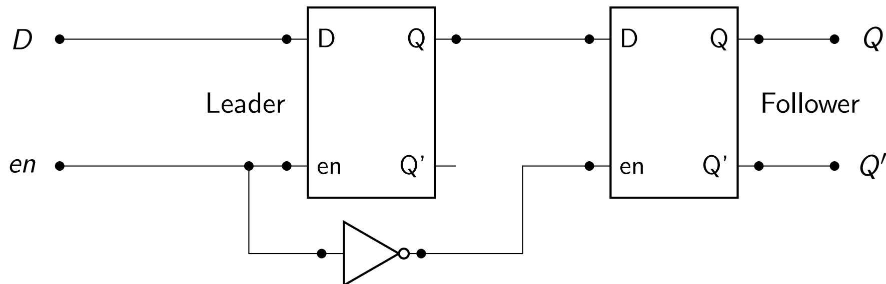
Crucially, even if *D* has changed, this new signal will propagate through the NOT gate faster than through the leader.

So the follower's *en* input goes low *before* the leader's output finishes updating, and the follower's output is unchanged.

Subsequent changes to *D* again have no effect.

How a D flip-flop works

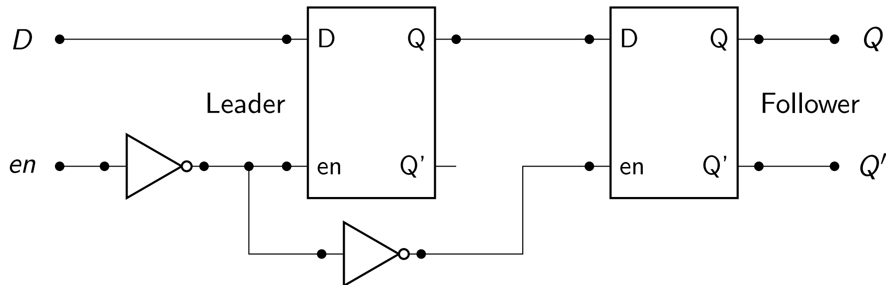
Negative edge-triggering is easier, so let's start with that.



So how do we trigger this on positive edges rather than negative edges?

How a D flip-flop works

Negative edge-triggering is easier, so let's start with that.

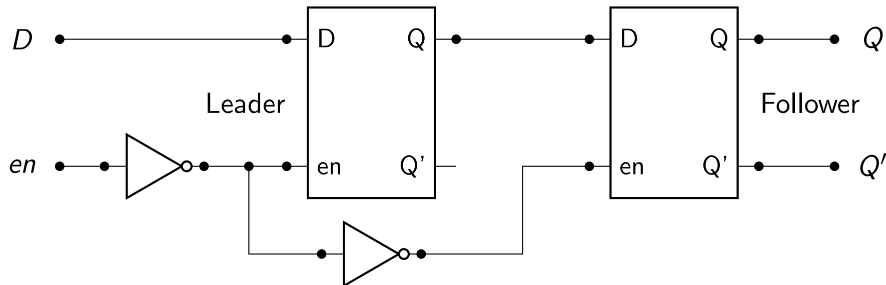


So how do we trigger this on positive edges rather than negative edges?

Just add another NOT gate!

How a D flip-flop works

Negative edge-triggering is easier, so let's start with that.



So how do we trigger this on positive edges rather than negative edges?

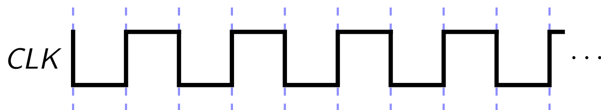
Just add another NOT gate!

This is a (positive edge-triggered) **D flip-flop**.

The difference between a **latch** and a **flip-flop** is that latches are level-triggered while flip-flops are edge-triggered.

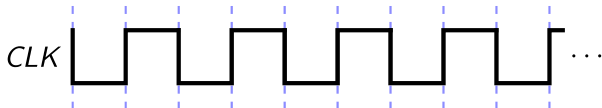
The clock

Now we're in a position to solve all our timing issues! We will drive our circuits with a *single* square wave: the **clock**, often denoted CLK.

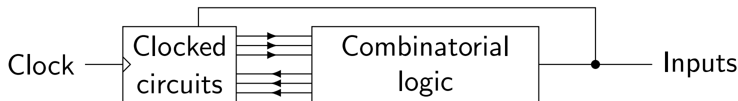


The clock

Now we're in a position to solve all our timing issues! We will drive our circuits with a *single* square wave: the **clock**, often denoted CLK.



We then use D flip-flops (and relatives) as a buffer against timing issues:

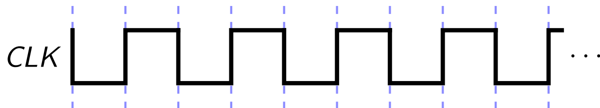


On each rising edge, the flip-flops update. Their outputs stay *constant* while changes propagate through the logic — even if their inputs change.

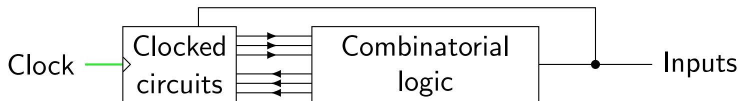
So as long as the gap between rising edges is at least as long as the maximum propagation delay, we can ignore it entirely and work with logic!

The clock

Now we're in a position to solve all our timing issues! We will drive our circuits with a *single* square wave: the **clock**, often denoted CLK.



We then use D flip-flops (and relatives) as a buffer against timing issues:

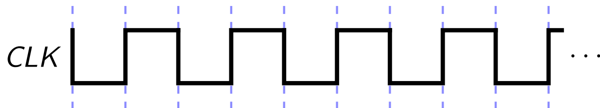


On each rising edge, the flip-flops update. Their outputs stay *constant* while changes propagate through the logic — even if their inputs change.

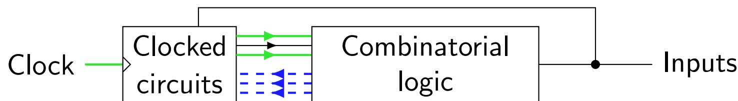
So as long as the gap between rising edges is at least as long as the maximum propagation delay, we can ignore it entirely and work with logic!

The clock

Now we're in a position to solve all our timing issues! We will drive our circuits with a *single* square wave: the **clock**, often denoted CLK.



We then use D flip-flops (and relatives) as a buffer against timing issues:

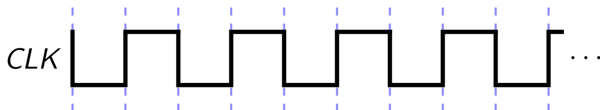


On each rising edge, the flip-flops update. Their outputs stay *constant* while changes propagate through the logic — even if their inputs change.

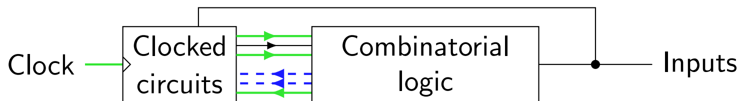
So as long as the gap between rising edges is at least as long as the maximum propagation delay, we can ignore it entirely and work with logic!

The clock

Now we're in a position to solve all our timing issues! We will drive our circuits with a *single* square wave: the **clock**, often denoted CLK.



We then use D flip-flops (and relatives) as a buffer against timing issues:

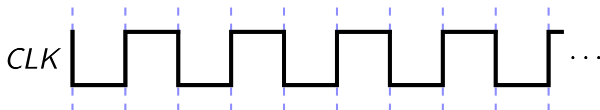


On each rising edge, the flip-flops update. Their outputs stay *constant* while changes propagate through the logic — even if their inputs change.

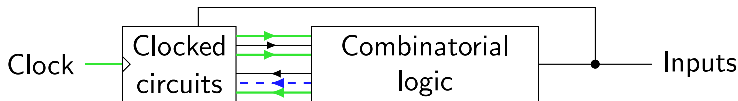
So as long as the gap between rising edges is at least as long as the maximum propagation delay, we can ignore it entirely and work with logic!

The clock

Now we're in a position to solve all our timing issues! We will drive our circuits with a *single* square wave: the **clock**, often denoted CLK.



We then use D flip-flops (and relatives) as a buffer against timing issues:

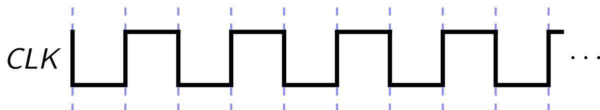


On each rising edge, the flip-flops update. Their outputs stay *constant* while changes propagate through the logic — even if their inputs change.

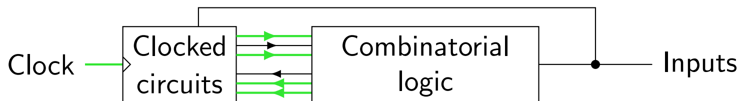
So as long as the gap between rising edges is at least as long as the maximum propagation delay, we can ignore it entirely and work with logic!

The clock

Now we're in a position to solve all our timing issues! We will drive our circuits with a *single* square wave: the **clock**, often denoted CLK.



We then use D flip-flops (and relatives) as a buffer against timing issues:

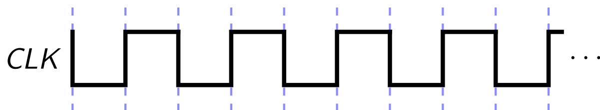


On each rising edge, the flip-flops update. Their outputs stay *constant* while changes propagate through the logic — even if their inputs change.

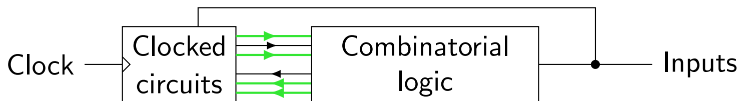
So as long as the gap between rising edges is at least as long as the maximum propagation delay, we can ignore it entirely and work with logic!

The clock

Now we're in a position to solve all our timing issues! We will drive our circuits with a *single* square wave: the **clock**, often denoted CLK.



We then use D flip-flops (and relatives) as a buffer against timing issues:

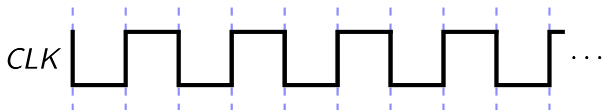


On each rising edge, the flip-flops update. Their outputs stay *constant* while changes propagate through the logic — even if their inputs change.

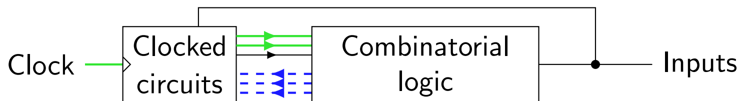
So as long as the gap between rising edges is at least as long as the maximum propagation delay, we can ignore it entirely and work with logic!

The clock

Now we're in a position to solve all our timing issues! We will drive our circuits with a *single* square wave: the **clock**, often denoted CLK.



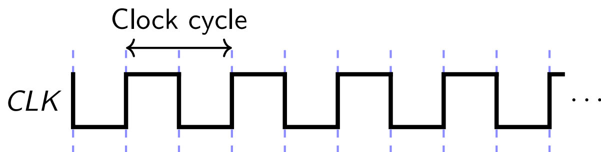
We then use D flip-flops (and relatives) as a buffer against timing issues:



On each rising edge, the flip-flops update. Their outputs stay *constant* while changes propagate through the logic — even if their inputs change.

So as long as the gap between rising edges is at least as long as the maximum propagation delay, we can ignore it entirely and work with logic!

The clock: Terminology



An interval of time between successive rising edges is called a **clock cycle**.

The length of time one clock cycle takes is called the clock's **time period**.

The number of clock cycles per second is the clock's **frequency**, measured in hertz (Hz). A frequency of 1Hz means one cycle per second, with

$$\text{Frequency (in hertz)} = 1/\text{Time period (in seconds)}.$$

For example, a clock with a frequency of 20kHz has a time period of $1/20,000$ seconds, i.e. $50\mu s$.

Clock signals are usually generated using **piezoelectric crystals** in a dedicated circuit using our arch-enemy, physics.

An aside: SI units reference

Base 10	Symbol	Name
	⋮	
10^{15}	P	Peta-
10^{12}	T	Tera-
10^{9}	G	Giga-
10^{6}	M	Mega-
10^{3}	k	Kilo-
10^{0}	N/A	N/A
10^{-3}	m	Milli-
10^{-6}	μ	Micro-
10^{-9}	n	Nano-
10^{-12}	p	Pico-
	⋮	

For example, 1GHz is $10^9 = 1,000,000,000$ hertz. There's one annoying caveat for bytes specifically that we'll discuss later.