# Sequential Logic Analysis

## Intro

### ILOs:

1. Understand latches, Flip-Flops, and timing diagrams. Specifically, the devices we'll care about are:
   1. SR Latch
   2. Data Latch (RS Flip-Flop with R tied to inverted S and an enable pin)
   3. D Flip-Flop (triggers on rising clock edge, sends D to Q)
   4. **Especially this one: JK Flip-Flop** (triggers on rising clock edge, holds if JK = 00, Set if 10, Reset if 01, toggle if 11)
   5. T Flip-Flop (JK with both inputs tied together so it only toggles)
2. Be able to produce timing diagrams, state transition tables, and state transition diagrams for circuits with arbitrary configurations of latches, flip-flops, and logic gates.

### Topic 7 Videos:

(Note: This was Topic #8 prior to 2020)

1. Analytical & Multisim: <https://www.youtube.com/watch?v=9BZV7hhDHb0&list=PLhbHWgMknRJT_eKLFXB843NkaNHfJ37Pw&index=33>
2. Lab Skills:
   1. "Sample Lab 8" <https://www.youtube.com/watch?v=zJ0iI5fk9A4&list=PLhbHWgMknRJT_eKLFXB843NkaNHfJ37Pw&index=34>
3. Hantek Lab Modification:
   1. Hardest part is making a clock signal, but we can do that with an external circuit; here's the video on how to do it: <https://youtu.be/5MMd51m0YhU>
      1. Update: You should buffer the output from the transistor by putting it through a gate (like the inverter shown below, or any other gate) prior to using it as a clock signal input for your flip-flops. This stops the output from going out of range from what the flip-flops are expecting (0 V to VCC) around state transitions.  
         Diagram, schematic

         Description automatically generated
      2. Note that you don't need specifically the 7406 inverter shown; the same idea will work with the inverters you have (SN74HC04), or any other gate you have where the output will toggle when the input toggles (i.e., NAND, NOR, AND, OR) by tying one input to the other or to high or low appropriately.
   2. Alternatively, you can lower VCC in your board to a low enough value that the Hantek AWG can generate a usable clock pulse. i.e., using 4 V as your input will make a 1.5 V offset 1 V amplitude square wave usable as a clock signal:   
      A picture containing text, electronics

      Description automatically generated

### Deliverables

Note: As always, your full objective for this topic is to review these notes, the videos, practice problems, live class sessions and forum content, then to write-up creating and tri-solving a variation problem of the week's topic that demonstrates you've mastered the content. With that all still in mind, following are some specific guidelines & tips for this week.

Tasks for your deliverable write-up:

1. Build a FSM using 4 flip-flops, and some combination of gates looping back on them.
   1. You don't need any external inputs to the FSM like the example in the first part of 7.1.4; inputs make the STT and STD much more complicated. Instead, you can build a circuit with no other external inputs like in the sample labs.
      1. i.e., all the J & K inputs to your flip-flops should come from either flip-flop outputs (Q or ~Q from any of your 4 flip-flops), constants (i.e., wired to ground or VCC) or logic gates (including chains of gates) which depend on only flip-flop outputs and constants.
   2. You should make all the clock inputs wired to the same clock signal; i.e., make it a **synchronous** FSM.
2. Analytically produce the full state-transition table (and state-transition diagram) for your logic showing all possible states the system could be in (considering all possible starting states). Identify all possible stable loops.
3. Use Multisim to check the state transition table and confirm all stable loops. Produce a timing diagram with multisim and show that it matches the analytical work (either by labeling states on the timing diagram or by first *analytically* producing a timing diagram and comparing them).
   1. ~~Also show a screen capture video of the circuit looping through the different states, indicating output with LEDs (or optionally a 7SD).~~ Instead of this, you can just show enough timing diagrams from different starting states (with explanations) to confirm your whole state transition diagram.
      1. i.e., start from enough different spots to have all possible states show up in your timing diagrams somewhere.
4. Physically build the circuit and do the same checks as you did using Multisim. Show results both using an oscilloscope-produced timing diagram, and a **video** of the circuit indicating output using LEDs.
   1. Note: you can manually clock the circuit for development, but watch out for switch bounce when doing so.
   2. Your final version this week should be automatically clocked; e.g., using the circuit in the video and companion PDF.
   3. Note: your scope only has two inputs, so you should show the clock and one (nontrivial) flip-flop output in your timing diagram (nontrivial means the bit doesn't just hold at one value).
      1. Option: *If possible*, you should try to have the LEDs in frame with the scope output so you can check the timing diagram with the output state.

Note: Including more logic gates can make your circuit more interesting, and actually make it easier because you will likely have less starting states that aren't part of loops.

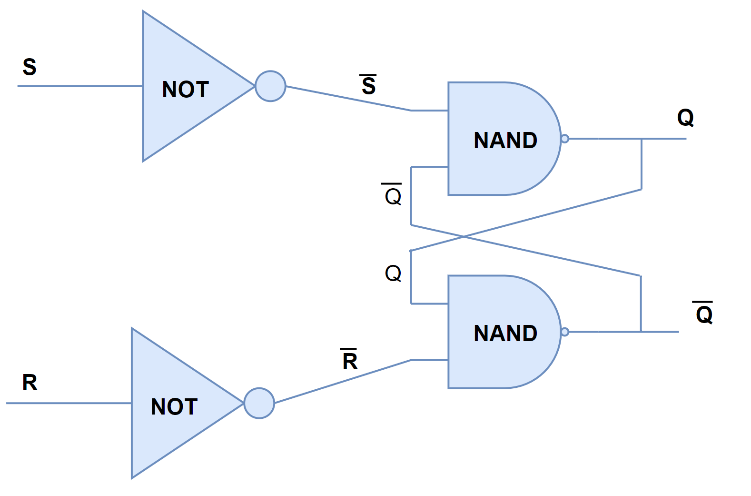
## Sequential Digital Logic Analysis

### Latches and Timing Diagrams

#### SR Latches

Cool video exploring latches: <https://youtu.be/KM0DdEaY5sY?t=604>

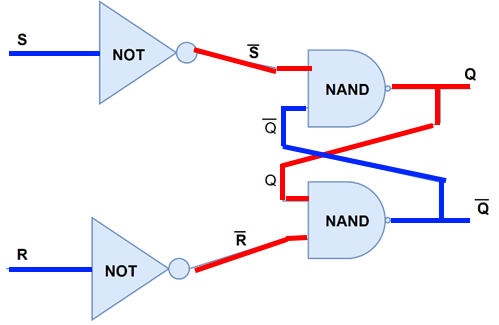
Consider this funky combination of gates with feedback:



The output *Q* depends not just on the inputs (*S* & *R*) but on what the *output* currently is! This configuration turns out to be one way to build an **SR-Latch**; the latch has the property that it holds its current output "state" *Q* at either high or low if the input "Set" and "Reset" signals both stay low; otherwise, the set pin being high can force the output *Q* high while the reset pin can force it to be low. Specifically, the output works like this:

|  |  |  |
| --- | --- | --- |
| S | R | Q |
| 0 | 0 | Present State |
| 0 | 1 | Reset |
| 1 | 0 | Set |
| 1 | 1 | Disallowed |

e.g., suppose at some instant in time both *S* & *R* are 0 and *Q* is 1 (and  is 0). It takes very some small but nonzero time for a changed input state to change the output state of the gates, so let's analyze what each NAND gate would do if this were the current state of inputs and outputs. Using red for high (1) and blue for low (0):

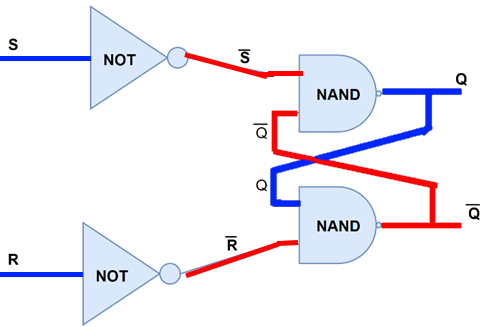


The top NAND gate will then try to output 

while the bottom one will try to output 

both of which are the current output states. Thus, this state is ***stable***, it will hold at this high value.

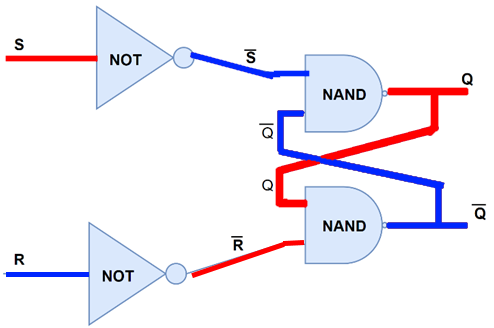
The SR latch would also hold its output *Q* to 0 if both inputs stay 0:



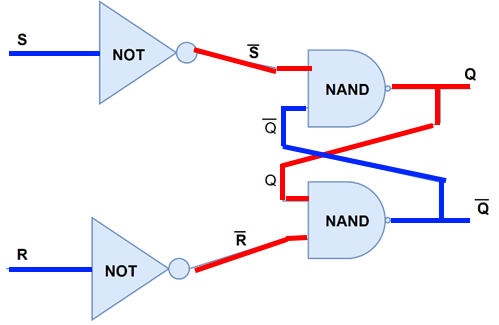
Now the top gate produces  while the bottom produces , so this state is stable as well!

*The SR latch stably holds its current output state as long as both S & R inputs stay low.*

Now suppose *S* goes high, making . Regardless of the current state of *Q* and , this will make at least one of the inputs to the top NAND gate a 0, which will force Q to be 1. As long as  is still high, that will then force  to be low:



Then after S returns low, this state will remain:

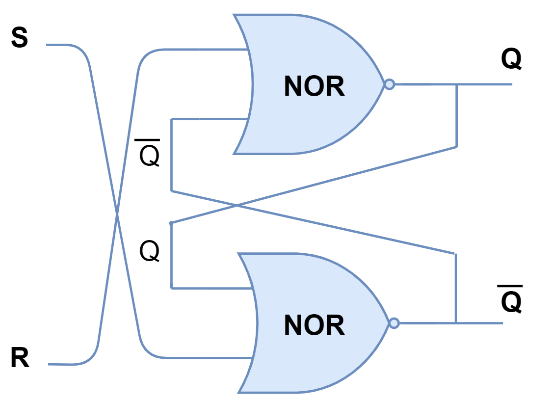


For that reason, the *S* input is called the "set" input - it's used to "set" the output state as high.

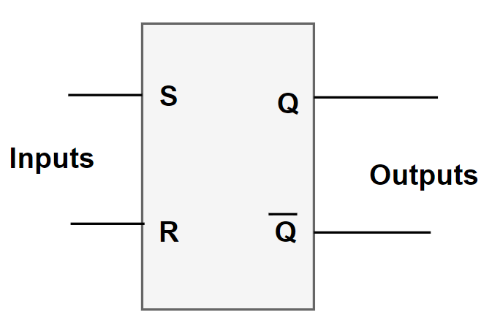
Similarly, if *R* goes high (while S stays low) then it will force the latch into the low output state (). For this reason, the *R* input is called the "reset" input - it's used to "reset" the output state to low.

Note that BOTH *R* and *S* being high at the same time is not allowed; that would make the latch end up with both  and  high, meaning that  is no longer the inverse of .

It's also possible to build an SR latch with NOR gates as follows:



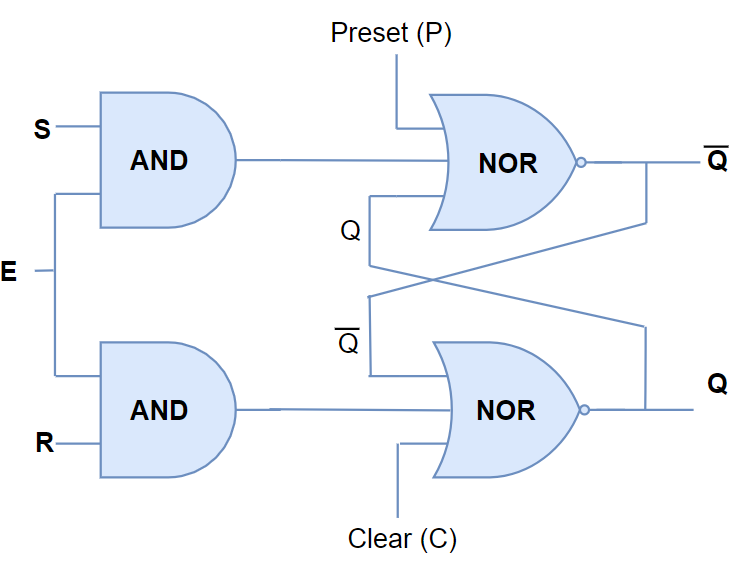
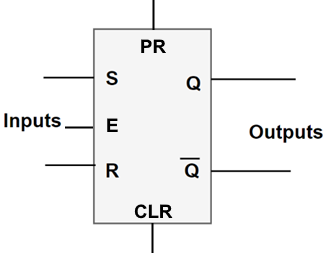
Regardless of which gates make it up inside, we can collect an SR latch into a single symbol as follows:



This saves us from needing to think about the gates, and instead lets us think in terms of the resulting behaviour:

|  |  |  |
| --- | --- | --- |
| S | R | Q |
| 0 | 0 | Present State |
| 0 | 1 | Reset |
| 1 | 0 | Set |
| 1 | 1 | Disallowed |

More elaborate SR latches have extra pins



These new pins do the following:

E: Enable - ignore signals on S & R unless E is high.

PR (or P): Preset - force the output to 1 regardless of the inputs

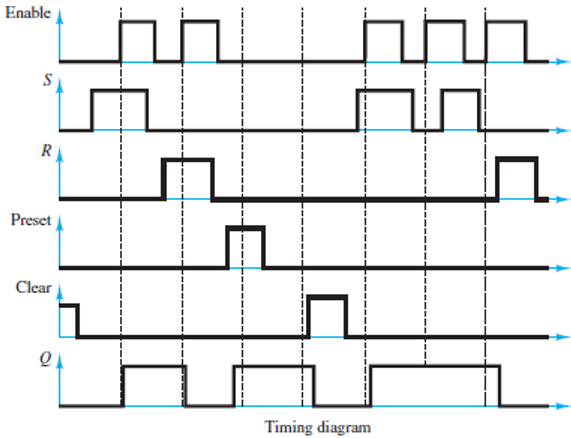
CLR (or C): Clear - force the output to 0 regardless of the inputs

Preset and Clear are normally set to 0, but can override the other behaviour and set or reset the the output state regardless of the other pins.

Often latches come with these pins whether you need them or not, so you need to be careful to wire them up rather than to leave them floating.

#### Timing Diagrams

We can explore how the output *Q* from an SR-latch depends on all 5 of these "inputs" by plotting the output and inputs vs. time in a **Timing Diagram** (this plots voltage of various signals on the vertical axis vs. time on the horizontal axis, just like you'd get with a multiple-input oscilloscope after shifting the signals vertically so they don't overlap)

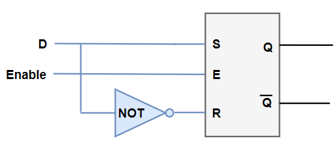


*Timing diagram for an SR-latch with an enable pin from* [1]

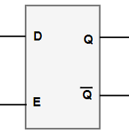
Note the slight delay between input signals changing and the output changing in response. This is due to switching time in the transistors that make up the latch's internal logic gates.

#### D-Latches

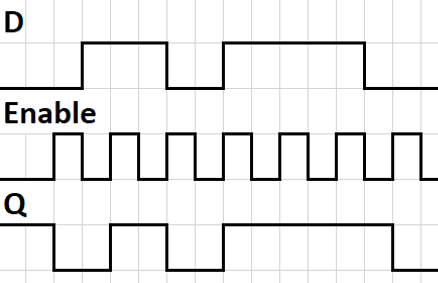
With an enable pin, the latch only responds to a 1 on the set or reset pin if the E pin is also set (and if both PR and CLR are low). With the enable pin, we can combine the function of the S & R pins into one single pin for data, the D pin:



Doing this creates a **Data latch** (AKA **Delay**, AKA **Gated Latch**):



With a D-latch, the input D gets transferred to the output only when the enable pin is also high; otherwise, the output holds its state and ignores changes in the input:



(Really there should be some small time delay between the inputs changing and the outputs changing, as before).

Note that if D changes at exactly the same time as E goes to 0 the latch may not catch the change in D, and will need to wait until the next time the enable pin is high before changing.

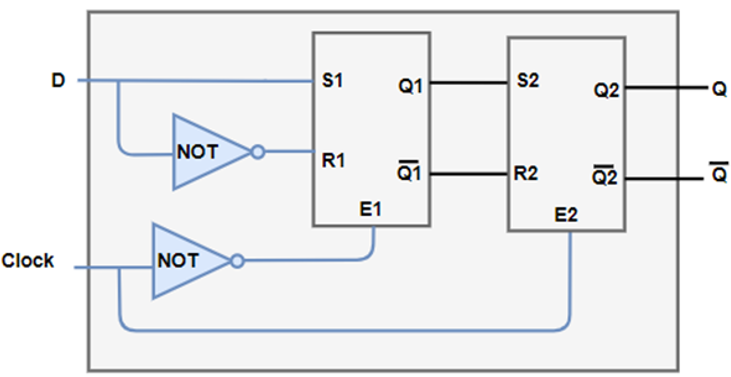
Data latches are the key to making computer memory: <https://www.youtube.com/watch?v=fpnE6UAfbtU>

### Flip-Flops

Cool video on flip-flops: <https://youtu.be/Hi7rK0hZnfc?t=404> (Note: sometimes, as in this video, latches are also referred to as flip-flops)

#### D Flip-Flops

Now suppose we take two latches and cascade them:



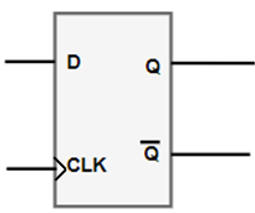
When Clock is low, E1 is high so Q1 will be set to whatever D is. However, at this time E2 is low so it will ignore any change in Q1; in this Clock state, only the first latch is enabled, while the second one is holding its state.

Similarly, when Clock is high only the second latch is enabled, meaning Q2 will become equal to Q1, and Q1 will stop changing in response to changing D values.

All together, this creates a device that will only change the overall output Q (= internal Q2) to the input value of *D* on a ***rising clock edge***: when the clock input changes from low to high.

This type of "edge-triggered" device (as opposed to "level-triggered" like a latch) is called a **flip-flop**.

Specifically, this flip-flop constructed from two data latches is a **D flip-flop**:

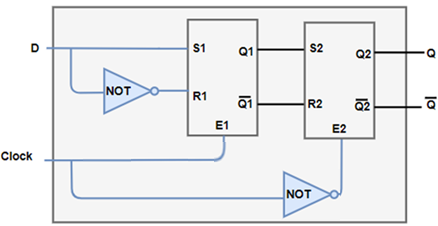


We can specify the behaviour on rising clock for the D flip-flop as follows:

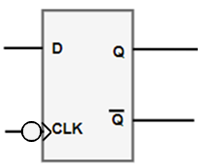
|  |  |  |
| --- | --- | --- |
| **D** | **CLK** | **Q** |
| **0** | **^** | **0** |
| **1** | **^** | **1** |

This indicates that the D flip-flop will send D to the output on a rising clock edge (but will leave the output the same any other time).

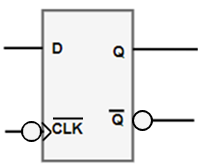
Often flip-flops trigger on falling clock edges instead. We can build this by making the not-gate apply to the second latch's enable pin instead:



And can symbolize this by placing a circle in front of the clock input:



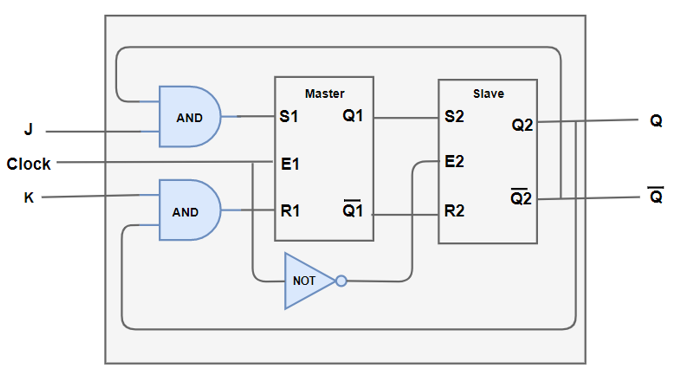
Note that in these situations often diagrams will show both the circle and the overbar to emphasize the wire's signal is inverted:



(This doesn't mean the clock and Q signals are double-inverted; it's just showing the inversion explicitly by the circle, and labeling wire from the perspective *outside* the chip; i.e.,  is what will appear on that wire out, not .)

#### JK Flip-Flops

A more complicated but much more versatile design is the **JK Flip-Flop.**



This flip-flop has two inputs (J and K) and makes use of all 4 possible input values for JK (00, 01, 10, & 11) when determining what to do on a clock edge:

|  |  |  |
| --- | --- | --- |
| Jn | Kn | Qn+1 |
| 0 | 0 | Qn (hold) |
| 0 | 1 | 0 (reset) |
| 1 | 0 | 1 (set) |
| 1 | 1 | !Qn (toggle) |

That is, on a clock edge (in this case a *falling* clock edge)

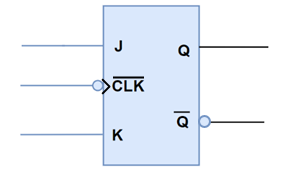
-If J and K are both 0, the flip-flop will **hold** its current output (keep it at whatever it was before the clock edge).

-If J is 0 while K is 1, it will **reset** the output to zero

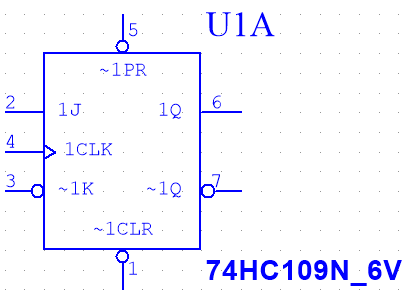
-If J is 1 while K is 0, it will **set** the output to 1

-If J and K are both 1, it will **toggle** its current output (switch it to the opposite of what it was before the clock edge.

The circuit symbol for a JK flip-flop is similar to that of a D flip-flop, but with two inputs:



Flip-flops can also be rising clock-edge triggered and come with preset and clear pins, like the popular 74HC109 we have in the lab:

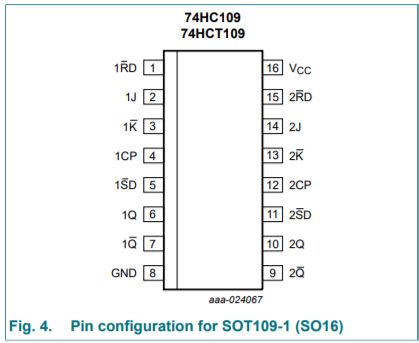


Like with the SR-latch, the preset and clear pins override the rest of the operation (including ignoring waiting for a clock edge) to "instantly"\* make the output high and low, respectively.

\*still with some transistor switching delay, but not waiting for a clock edge.

Note that the preset and clear pins on the 74HC109 are inverted too, meaning that the input pins are more like "Don't Preset" and "Don't clear" - they need to be wired to high or they'll override the flip-flop output to 1 and 0 (respectively), so don't forget to do this!

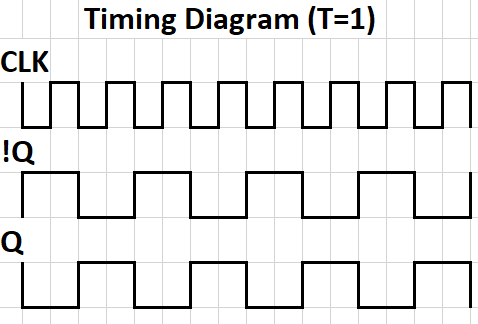
The 74HC109N has two flip-flops in one package. Compare the Multisim image for U1A above to the pinout from the datasheet:

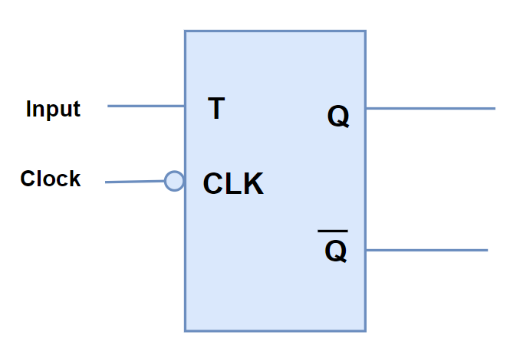


Pinout on 74HC109N, from [2]

#### T Flip-Flop:

If you were to wire *K* to , the *JK* flip-flop would behave like a D-flip-flop: always put *J* onto the output on the clock edge. But what if you wire  to *K*? This makes the JK flip-flop into a T-flip-flop: it either holds its current state or toggles it, depending on the value of the common input:





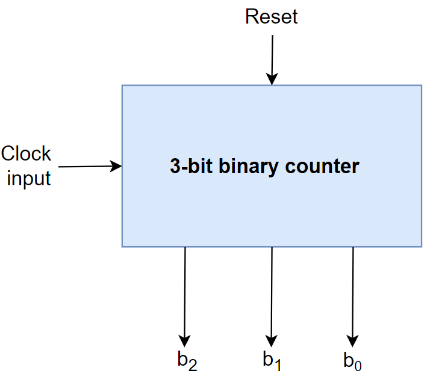
Flip-flops form the basis of some very useful digital devices, like counters and registers (memory).

### Digital Counters and Registers

Key Devices for this section:

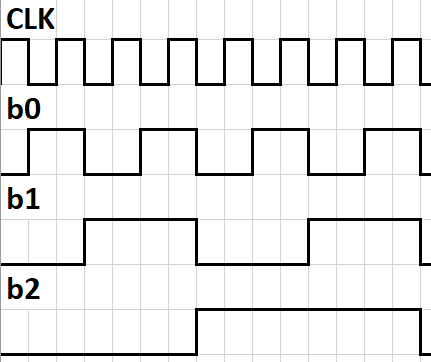
1. 4-bit counter
2. Ripple counter
3. decade counter (and why propagation delays can make some designs not work like you'd think)
4. synchronous counter
5. Registers

A counter is a device that keeps count of how many input pulses (like a falling clock edge) have happened:

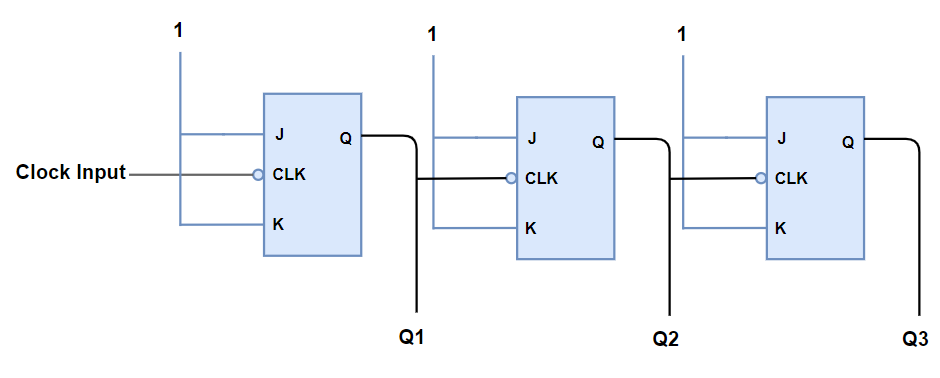


|  |  |  |  |
| --- | --- | --- | --- |
| Input Pulses | b2 | b1 | b0 |
| 0 | 0 | 0 | 0 |
| 1 | 0 | 0 | 1 |
| 2 | 0 | 1 | 0 |
| 3 | 0 | 1 | 1 |
| 4 | 1 | 0 | 0 |
| 5 | 1 | 0 | 1 |
| 6 | 1 | 1 | 0 |
| 7 | 1 | 1 | 1 |
| 8 | 0 | 0 | 0 |
| 9 | 0 | 0 | 1 |
| … |  |  |  |

The timing diagram helps give a hint of how to build such a thing:

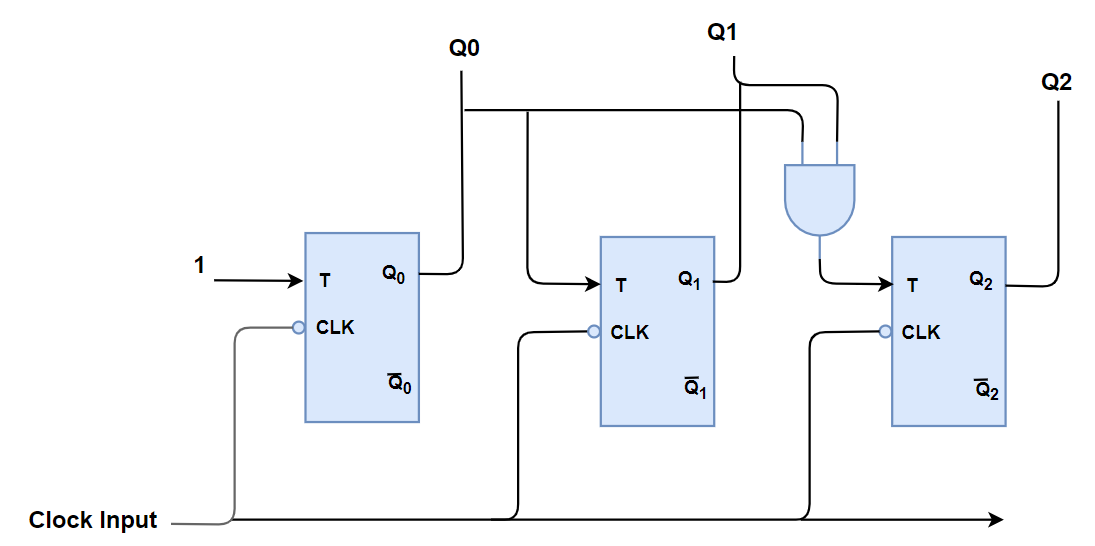


One way is by taking the output from three falling-edge-triggered flip-flops set to toggle:



This setup is called a **ripple counter**. In a ripple counter, each flip-flop only gets a signal to change when the one before it has changed, so the change *ripples* through from left to right. This means that in the ripple counter not all the bits change at the same time (because there's switching delays due to rise times of capacitances within the transistors that make up the logic gates, and the second flip-flop doesn't get the signal to change until the first one finishes changes its output), so intermediate values can be strange.

Another implementation is the **synchronous counter**:



Now, all flip-flops share the same clock signal so they all change at the same time. It still counts though, because the toggle input of higher bits is only set if *all* lower bits are high.

**Decade Counter:**

A decade counter is a counter that counts up to 9 and then resets instead of counting ten:

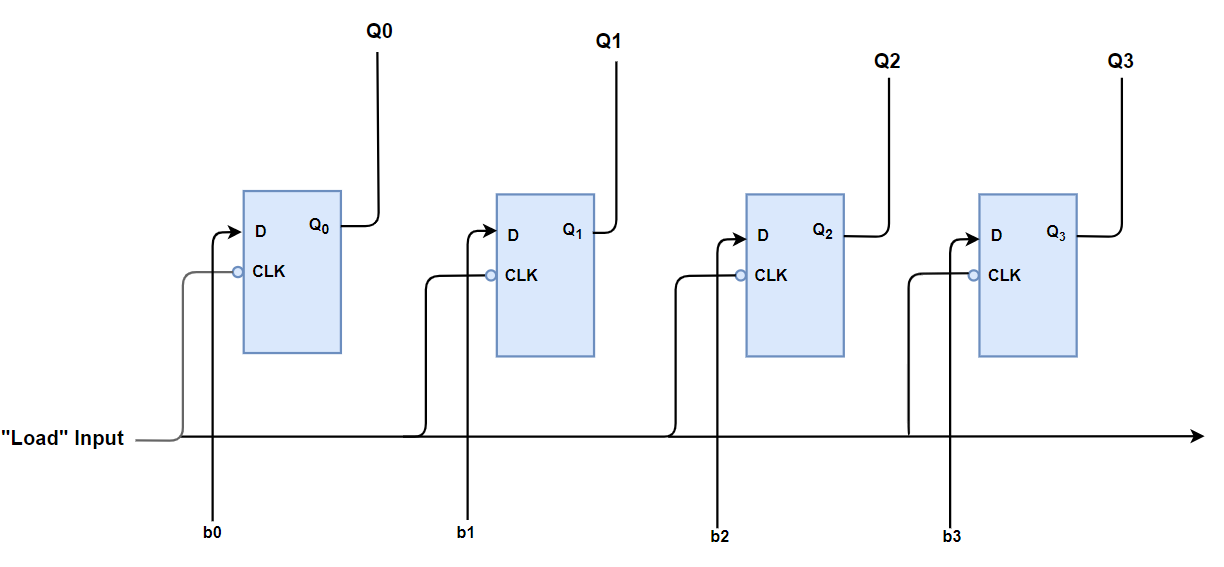
Timing Table:

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| Input Pulses | b3 | b2 | b1 | b0 |
| 0 | 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 | 0 | 0 | 0 | 0 |

You can build such a thing by checking if the input state is 9 and forcing the next state to be 0, or by using the clear pins to force it to 0 as soon as it would hit 10.

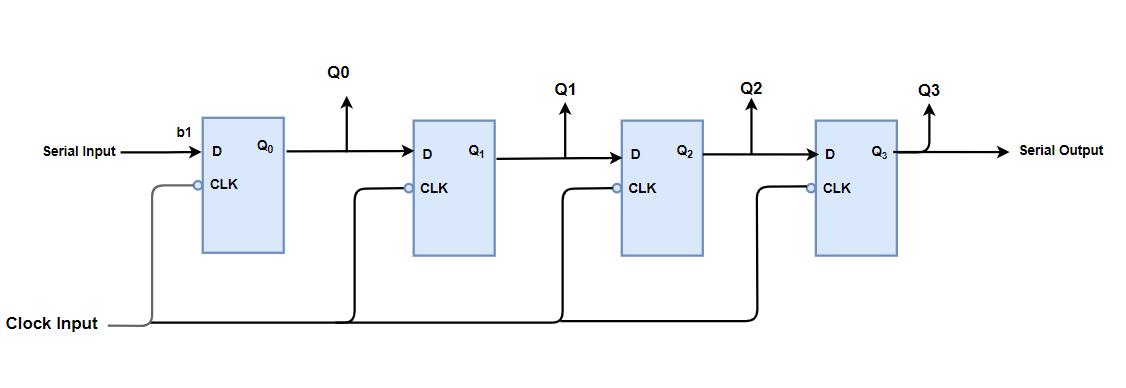
**Parallel Register:**

A register is a flip-flop device for storing data rather than counting. Whenever the "load input" sends a falling edge (moves from high to low) the parallel register below will load the data on b0b1b2b3 into the memory Q0Q1Q2Q3.



**Shift Register:**

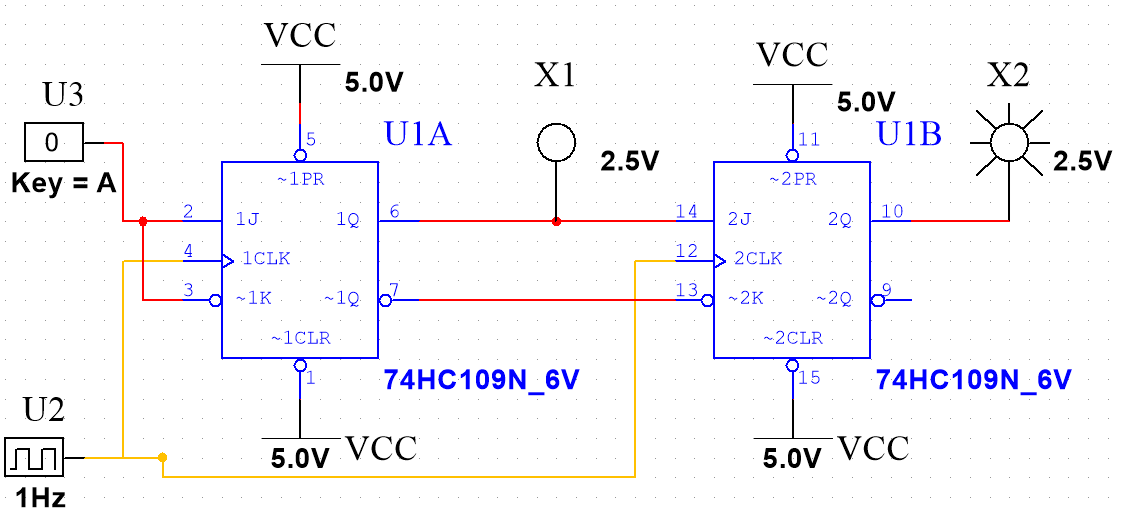
This type of register transfers the data from left to right on the clock edges, and is useful for transmitting data serially: using only one data line (i.e., one wire).



### Finite State Machines

Counters and registers are only two examples of a general class of devices based on flip-flops and gates called Finite-State Machines (FSMs). FSMs are so-called because they have a finite number of states they can be in at any time. FSMs are deterministic, in that the next state they'll enter is uniquely determined by the current state and the set of external inputs (if any).

For example, consider this combination of two flip-flops:



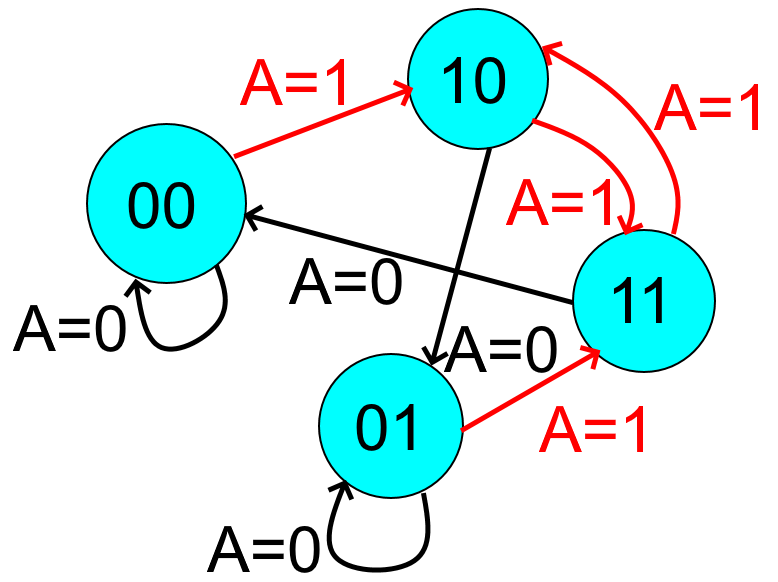
Considering the Q values (but not the input A or the clock signal) as the current state, this system has 4 possible states it can be in at any time: Q1Q2 = 00, 01, 10, or 11.

On each rising clock pulse it will move to another state depending on the current state it is in. The first flip-flop has both inputs tied to A, so Q1 will either reset if A is a 0, or set if A is a 1.

The second flip flop has its inputs J2 = Q1 and ~K2 = ~Q1, i.e., K2 = Q1, so is either holding or toggling. Specifically, Q2 will hold when Q1 is a 0, and will toggle if Q1 is a 1. With that determined, you can complete the **state transition table** for this circuit as follows:

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| Current State | | Inputs | Next State | |
| q1 | q2 | A | Q1 | Q2 |
| 0 | 0 | 0 | 0 | 0 |
| 0 | 0 | 1 | 1 | 0 |
| 0 | 1 | 0 | 0 | 1 |
| 0 | 1 | 1 | 1 | 1 |
| 1 | 0 | 0 | 0 | 1 |
| 1 | 0 | 1 | 1 | 1 |
| 1 | 1 | 0 | 0 | 0 |
| 1 | 1 | 1 | 1 | 0 |

Often FSMs are visualized via state-transition diagrams, with circles indicating each state, and arrows indicating the next state (and which input would get it there, if applicable):

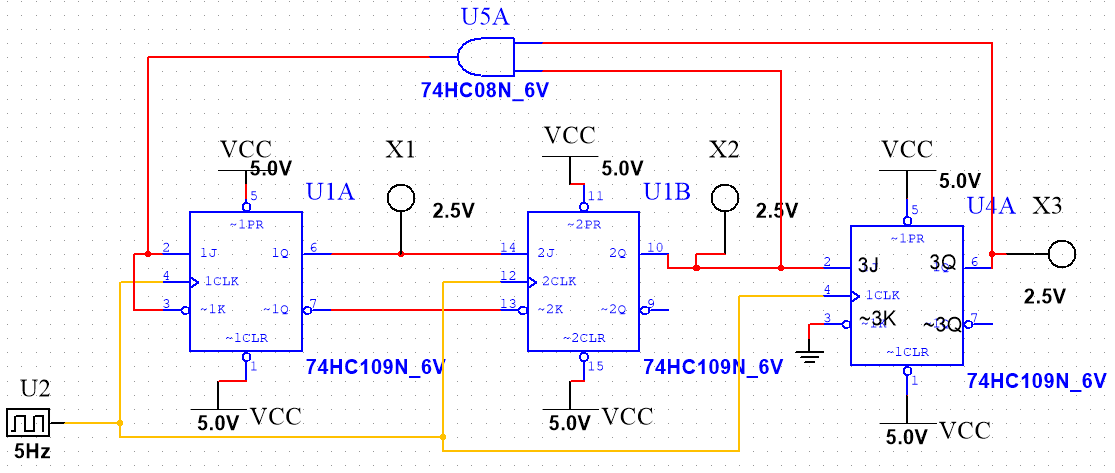


In this case, if A holds still at 0, the FSM will find its way to either 00 or 01 and hold, depending on where it started. If instead A holds at 1, it will cycle back and forth between 10 and 11.

A cycling state like this is called a **stable loop**.

#### Example with no external inputs

Consider the following set of flip-flops with no external inputs:



Create a state transition table and state transition diagram for all possible states. Identify any stable loops.

Solution:

It can help to write logic for each of the input J and K values, then think through what the flip-flop would do as a result.

Call the outputs Q1, Q2, and Q3 going left to right.

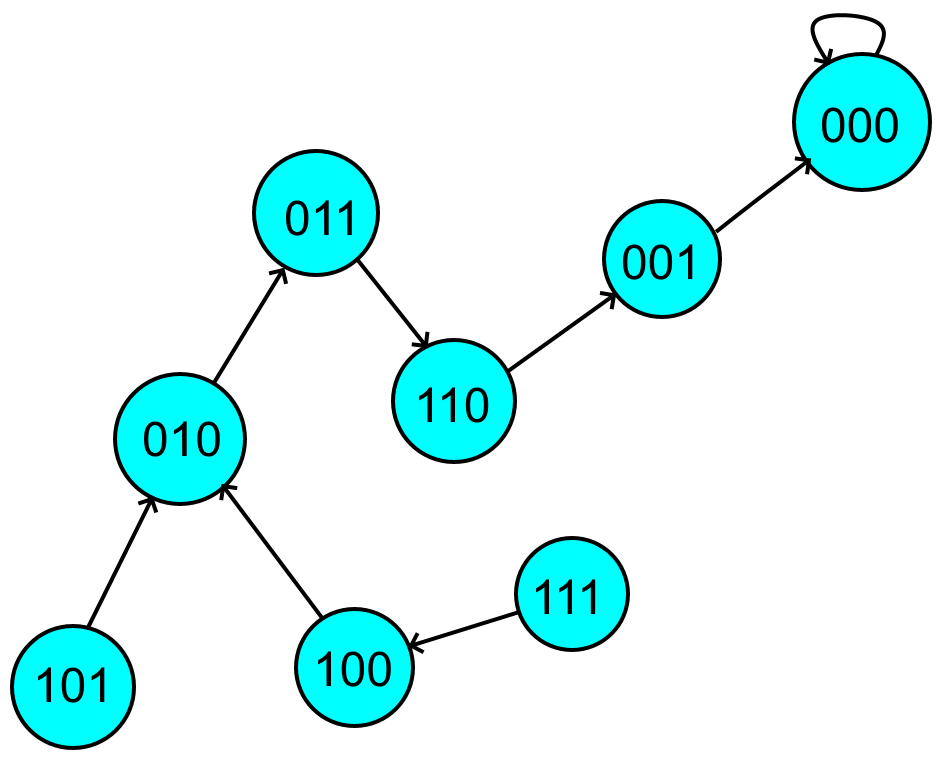
Then J1 = ~K1 = . Because , this flip-flop is operating like a D-flip-flop; only setting (if both Q2 & Q3 are 1) and resetting otherwise.

The second flip-flop has J2 = Q1, ~K2 = ~Q1, so J2 = K2 = Q1 → this flip-flop will toggle if Q1 is a 1, and hold otherwise.

The third flip-flop sees ~K3 = 0 → K3 = 1, meaning this flip-flop can only toggle (if J3 = Q2 =1) and reset (if J3 = Q2 = 0).

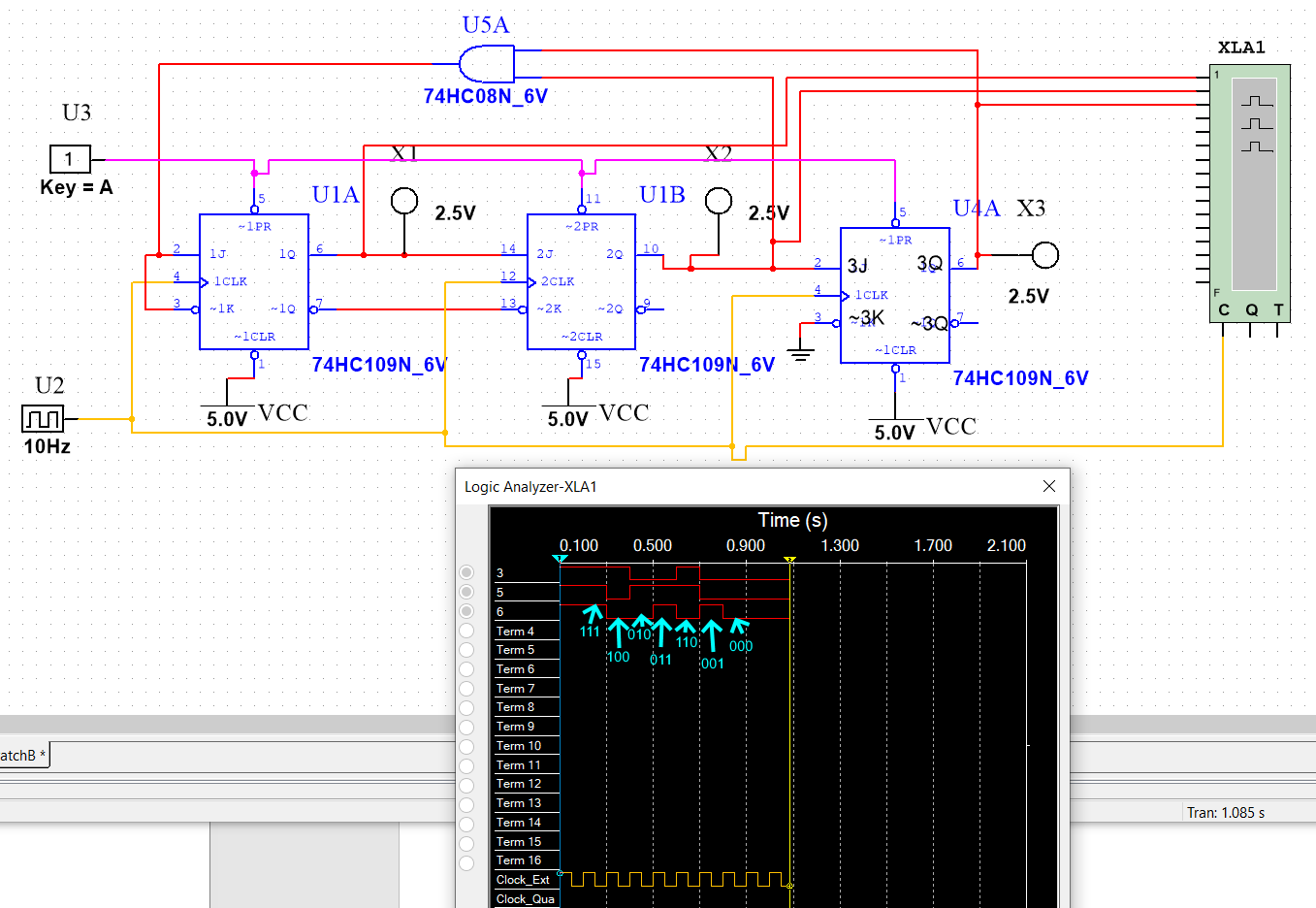
This lets us draw the state transition table & diagram:

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| Current State | | | Next State | | |
| q1 | q2 | q3 | Q1 | Q2 | Q3 |
| 0 | 0 | 0 | 0 | 0 | 0 |
| 0 | 0 | 1 | 0 | 0 | 0 |
| 0 | 1 | 0 | 0 | 1 | 1 |
| 0 | 1 | 1 | 1 | 1 | 0 |
| 1 | 0 | 0 | 0 | 1 | 0 |
| 1 | 0 | 1 | 0 | 1 | 0 |
| 1 | 1 | 0 | 0 | 0 | 1 |
| 1 | 1 | 1 | 1 | 0 | 0 |



Thus, regardless of the starting state, this set of flip-flops ends up in the one-state stable loop of 000. The longest possible sequence is 7 numbers long; 111→100→010→011→110→001→000

We can verify this in Multisim by hooking up the ~PR pin to a toggle switch to start it in the 111 state, then hook up the logic analyzer (set trigger to external clock) and observe the timing diagram after stopping the ~PR override:



Hurrah!

##### Tip: J~K Flip-Flop

When dealing exclusively with flip-flops whose inputs are actually J and ~K rather than J and K, it can be helpful to consider the JK-flip-flop operation based on ~K

|  |  |  |
| --- | --- | --- |
| Jn | Kn | Qn+1 |
| 0 | 0 | Qn |
| 0 | 1 | 0 (reset) |
| 1 | 0 | 1 (set) |
| 1 | 1 | !Qn (toggle) |

Therefore,

|  |  |  |
| --- | --- | --- |
| Jn | ~Kn | Qn+1 |
| 0 | 1 | Qn |
| 0 | 0 | 0 (reset) |
| 1 | 1 | 1 (set) |
| 1 | 0 | !Qn (toggle) |