ECE199JL: Introduction to Computer Engineering **Fall 2012**
Notes Set 2.6

Sequential Logic

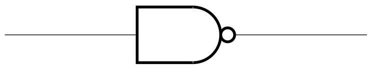
These notes introduce logic components for storing bits, building up from the idea of a pair of cross-coupled inverters through an implementation of a flip-flop, the storage abstractions used in most modern logic design processes. We then introduce simple forms of timing diagrams, which we use to illustrate the behavior of a logic circuit. After commenting on the benefits of using a clocked synchronous abstraction when designing systems that store and manipulate bits, we illustrate timing issues and explain how these are abstracted away in clocked synchronous designs. *Sections marked with an asterisk are provided solely for your interest, but you probably need to learn this material in later classes.*

Storing One Bit

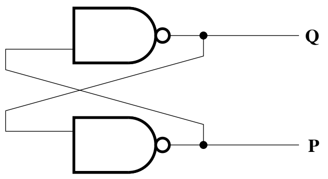
So far we have discussed only implementation of Boolean functions: given some bits as input, how can we design a logic circuit to calculate the result of applying some function to those bits? The answer to such questions is called **combinational logic** (sometimes **combinatorial logic**), a name stemming from the fact that we are combining existing bits with logic, not storing new bits.

You probably already know, however, that combinational logic alone is not sufficient to build a computer. We need the ability to store bits, and to change those bits. Logic designs that make use of stored bits—bits that can be changed, not just wires to high voltage and ground—are called **sequential logic**. The name comes from the idea that such a system moves through a sequence of stored bit patterns (the current stored bit pattern is called the **state** of the system).

Consider the diagram to the right. What is it? A 1-input NAND gate, or an inverter drawn badly? If you think carefully about how these two gates are built, you will realize that they are the same thing. Conceptually, we use two inverters to store a bit, but in most cases we make use of NAND gates to simplify the mechanism for changing the stored bit.



Take a look at the design to the right. Here we have taken two inverters (drawn as NAND gates) and coupled each gate's output to the other's input. What does the circuit do? Let's make some guesses and see where they take us. Imagine that the value at Q is 0. In that case, the lower gate drives P to 1. But P drives the upper gate, which forces Q to 0. In other words, this combination forms a stable state of the system: once the gates reach this state, they continue to hold these values. The first row of the truth table to the right (outputs only) shows this state.



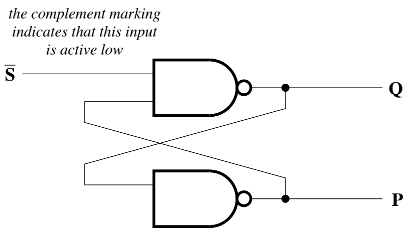
Q	P
0	1
1	0

What if $Q = 1$, though? In this case, the lower gate forces P to 0, and the upper gate in turn forces Q to 1. Another stable state! The $Q = 1$ state appears as the second row of the truth table.

We have identified all of the stable states.[1] Notice that our cross-coupled inverters can store a bit. Unfortunately, we have no way to specify which value should be stored, nor to change the bit's value once the gates have settled into a stable state. What can we do?

[1]Most logic families also allow unstable states in which the values alternate rapidly between 0 and 1. These metastable states are beyond the scope of our class, but ensuring that they do not occur in practice is important for real designs.

Let's add an input to the upper gate, as shown to the right. We call the input $\bar{S}$. The "S" stands for set—as you will see, our new input allows us to **set** our stored bit Q to 1. The use of a complemented name for the input indicates that the input is **active low**. In other words, the input performs its intended task (setting Q to 1) when its value is 0 (not 1).


the complement marking indicates that this input is active low
S̄
Q
P

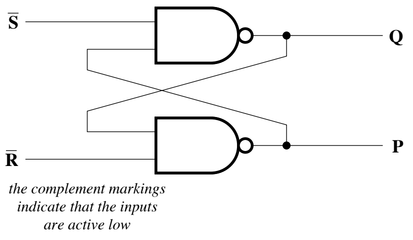
$\bar{S}$	Q	P
1	0	1
1	1	0
0	1	0

Think about what happens when the new input is not active, $\bar{S} = 1$. As you know, ANDing any value with 1 produces the same value, so our new input has no effect when $\bar{S} = 1$. The first two rows of the truth table are simply a copy of our previous table: the circuit can store either bit value when $\bar{S} = 1$. What happens when $\bar{S} = 0$? In that case, the upper gate's output is forced to 1, and thus the lower gate's is forced to 0. This third possibility is reflected in the last row of the truth table.

Now we have the ability to force bit Q to have value 1, but if we want $Q = 0$, we just have to hope that the circuit happens to settle into that state when we turn on the power. What can we do?

As you probably guessed, we add an input to the other gate, as shown to the right. We call the new input $\bar{R}$: the input's purpose is to **reset** bit Q to 0, and the input is active low. We extend the truth table to include a row with $\bar{R} = 0$ and $\bar{S} = 1$, which forces $Q = 0$ and $P = 1$.

an R̄–S̄ latch (stores a single bit)

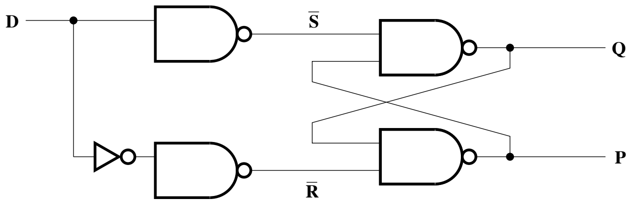

S̄
Q
P
R̄
the complement markings indicate that the inputs are active low

$\bar{R}$	$\bar{S}$	Q	P
1	1	0	1
1	1	1	0
1	0	1	0
0	1	0	1
0	0	1	1

The circuit that we have drawn has a name: an **R̄-S̄ latch**. One can also build R-S latches (with active high set and reset inputs). The textbook also shows an R̄-S̄ latch (labeled incorrectly). Can you figure out how to build an R-S latch yourself?

Let's think a little more about the R̄-S̄ latch. What happens if we set $\bar{S} = 0$ and $\bar{R} = 0$ at the same time? Nothing bad happens immediately. Looking at the design, both gates produce 1, so $Q = 1$ and $P = 1$. The bad part happens later: if we raise both $\bar{S}$ and $\bar{R}$ back to 1 at around the same time, the stored bit may end up in either state.[2]

We can avoid the problem by adding gates to prevent the two control inputs ($\bar{S}$ and $\bar{R}$) from ever being 1 at the same time. A single inverter might technically suffice, but let's build up the structure shown below, noting that the two inverters in sequence connecting D to $\bar{R}$ have no practical effect at the moment. A truth table is shown to the right of the logic diagram. When $D = 0$, $\bar{R}$ is forced to 0, and the bit is reset. Similarly, when $D = 1$, $\bar{S}$ is forced to 0, and the bit is set.

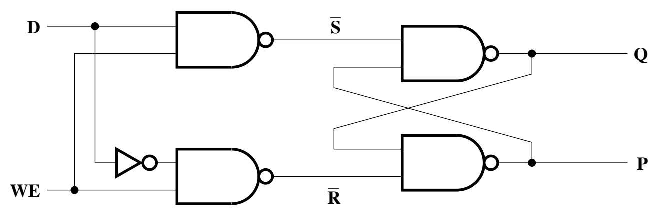


D	$\bar{R}$	$\bar{S}$	Q	P
0	0	1	0	1
1	1	0	1	0

Unfortunately, except for some interesting timing characteristics, the new design has the same functionality as a piece of wire. And, if you ask a circuit designer, thin wires also have some interesting timing characteristics. What can we do? Rather than having Q always reflect the current value of D, let's add some extra inputs to the new NAND gates that allow us to control when the value of D is copied to Q, as shown on the next page.

[2]Or, worse, in a metastable state, as mentioned earlier.

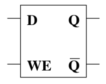
a gated D latch (stores a single bit)



WE	D	$\bar{R}$	$\bar{S}$	Q	P
1	0	0	1	0	1
1	1	1	0	1	0
0	0	1	1	0	1
0	1	1	1	0	1
0	0	1	1	1	0
0	1	1	1	1	0

The WE (write enable) input controls whether or not Q mirrors the value of D. The first two rows in the truth table are replicated from our "wire" design: a value of $WE = 1$ has no effect on the first two NAND gates, and $Q = D$. A value of $WE = 0$ forces the first two NAND gates to output 1, thus $\bar{R} = 1$, $\bar{S} = 1$, and the bit Q can occupy either of the two possible states, regardless of the value of D, as reflected in the lower four lines of the truth table.

gated D latch symbol



P and Q were always opposites, so we now just write $\bar{Q}$ (often omitted entirely in drawings)

The circuit just shown is called a **gated D latch**, and is an important mechanism for storing state in sequential logic. (Random-access memory uses a slightly different technique to connect the cross-coupled inverters, but latches are used for nearly every other application of stored state.) The "D" stands for "data," meaning that the bit stored is matches the value of the input. Other types of latches (including S-R latches) have been used historically, but D latches are used predominantly today, so we omit discussion of other types. The "gated" qualifier refers to the presence of an enable input (we called it WE) to control when the latch copies its input into the stored bit. A symbol for a gated D latch appears to the right. Note that we have dropped the name P in favor of $\bar{Q}$, since $P = \bar{Q}$ in a gated D latch.

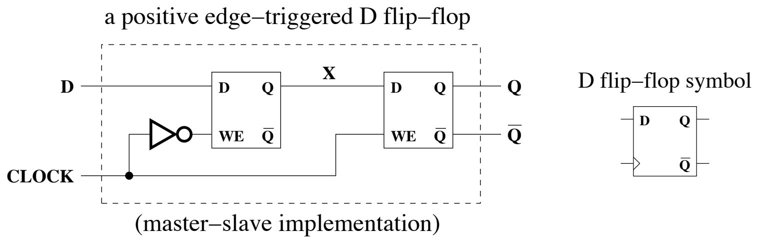
The Clock Abstraction

High-speed logic designs often use latches directly. Engineers specify the number of latches as well as combinational logic functions needed to connect one latch to the next, and the CAD tools optimize the combinational logic. The enable inputs of successive groups of latches are then driven by what we call a clock signal, a single bit line distributed across most of the chip that alternates between 0 and 1 with a regular period. While the clock is 0, one set of latches holds its bit values fixed, and combinational logic uses those latches as inputs to produce bits that are copied into a second set of latches. When the clock switches to 1, the second set of latches stops storing their data inputs and retains their bit values in order to drive other combinational logic, the results of which are copied into a third set of latches. Of course, some of the latches in the first and third sets may be the same.

The timing of signals in such designs plays a critical role in their correct operation. Fortunately, we have developed powerful abstractions that allow engineers to ignore much of the complexity while thinking about the Boolean logic needed for a given design.

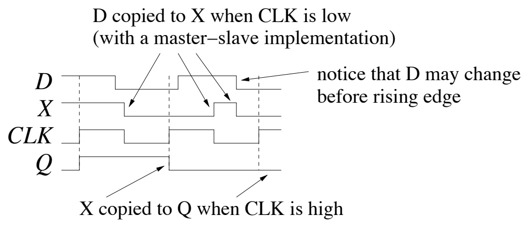
Towards that end, we make a simplifying assumption for the rest of our class, and for most of your career as an undergraduate: the clock signal is a **square wave** delivered uniformly across a chip. For example, if the period of a clock is 0.5 nanoseconds (2 GHz), the clock signal is a 1 for 0.25 nanoseconds, then a 0 for 0.25 nanoseconds. We assume that the clock signal changes instantaneously and at the same time across the chip. Such a signal can never exist in the real world: voltages do not change instantaneously, and the phrase "at the same time" may not even make sense at these scales. However, circuit designers can usually provide a clock signal that is close enough, allowing us to forget for now that no physical signal can meet our abstract definition.

The device shown to the right is a **master-slave** implementation of a **positive edge-triggered** D flip-flop. As you can see, we have constructed it from two gated D latches with opposite senses of write enable. The "D" part of the name has the same meaning as with a gated D latch: the bit stored is the same as the one delivered to the input. Other variants of flip-flops have also been built, but this type dominates designs today. Most are actually generated automatically from hardware "design" languages (that is, computer programming languages for hardware design).



When the clock is low (0), the first latch copies its value from the flip-flop's D input to the midpoint (marked X in our figure, but not usually given a name). When the clock is high (1), the second latch copies its value from X to the flip-flop's output Q. Since X can not change when the clock is high, the result is that the output changes each time the clock changes from 0 to 1, which is called the **rising edge** or **positive edge** (the derivative) of the clock signal. Hence the qualifier "positive edge-triggered," which describes the flip-flop's behavior. The "master-slave" implementation refers to the use of two latches. In practice, flip-flops are almost never built this way. To see a commercial design, look up 74LS74, which uses six 3-input NAND gates and allows set/reset of the flip-flop (using two extra inputs).

The **timing diagram** to the right illustrates the operation of our flip-flop. In a timing diagram, the horizontal axis represents (continuous) increasing time, and the individual lines represent voltages for logic signals. The relatively simple version shown here uses only binary values for each signal. One can also draw transitions more realistically (as taking finite time). The dashed vertical lines here represent the times at which the clock rises. To make the example interesting, we have varied D over two clock cycles. Notice that even though D rises and falls during the second clock cycle, its value is not copied to the output of our flip-flop. One can build flip-flops that "catch" this kind of behavior (and change to output 1), but we leave such designs for later in your career.



Circuits such as latches and flip-flops are called **sequential feedback** circuits, and the process by which they are designed is beyond the scope of our course. The "feedback" part of the name refers to the fact that the outputs of some gates are fed back into the inputs of others. Each cycle in a sequential feedback circuit can store one bit. Circuits that merely use latches and flip-flops as building blocks are called **clocked synchronous sequential circuits**. Such designs are still sequential: their behavior depends on the bits currently stored in the latches and flip-flops. However, their behavior is substantially simplified by the use of a clock signal (the "clocked" part of the name) in a way that all elements change at the same time ("synchronously").

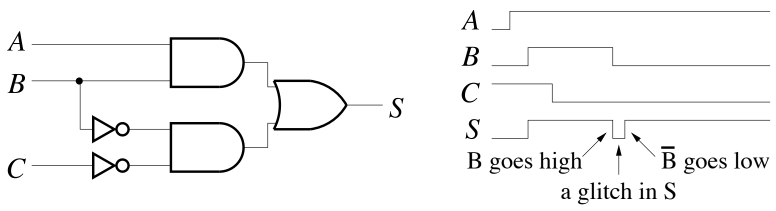
The value of using flip-flops and assuming a square-wave clock signal with uniform timing may not be clear to you yet, but it bears emphasis. With such assumptions, *we can treat time as having discrete values.* In other words, time "ticks" along discretely, like integers instead of real numbers. We can look at the state of the system, calculate the inputs to our flip-flops through the combinational logic that drives their D inputs, and be confident that, when time moves to the next discrete value, we will know the new bit values stored in our flip-flops, allowing us to repeat the process for the next clock cycle without worrying about exactly when things change. Values change only on the rising edge of the clock!

Real systems, of course, are not so simple, and we do not have one clock to drive the universe, so engineers must also design systems that interact even though each has its own private clock signal (usually with different periods).

Static Hazards: Causes and Cures*

Before we forget about the fact that real designs do not provide perfect clocks, let's explore some of the issues that engineers must sometimes face. We discuss these primarily to ensure that you appreciate the power of the abstraction that we use in the rest of our course. In later classes (probably our 298, which will absorb material from 385), you may be required to master this material. *For now, we provide it simply for your interest.*

Consider the circuit shown below, for which the output is given by the equation $S = AB + \bar{B}\bar{C}$.

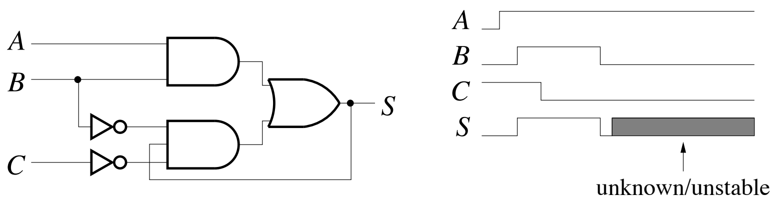


The timing diagram on the right shows a **glitch** in the output when the input shifts from $ABC = 110$ to 100, that is, when B falls. The problem lies in the possibility that the upper AND gate, driven by B, might go low before the lower AND gate, driven by $\bar{B}$, goes high. In such a case, the OR gate output S falls until the second AND gate rises, and the output exhibits a glitch.

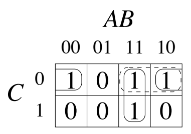
A circuit that might exhibit a glitch in an output that functionally remains stable at 1 is said to have a **static-1 hazard**. The qualifier "static" here refers to the fact that we expect the output to remain static, while the "1" refers to the expected value of the output.

The presence of hazards in circuits can be problematic in certain cases. In domino logic, for example, an output is precharged and kept at 1 until the output of a driving circuit pulls it to 0, at which point it stays low (like a domino that has been knocked over). If the driving circuit contains static-1 hazards, the output may fall in response to a glitch.

Similarly, hazards can lead to unreliable behavior in sequential feedback circuits. Consider the addition of a feedback loop to the circuit just discussed, as shown in the figure below. The output of the circuit is now given by the equation $S^* = AB + \bar{B}\bar{C}S$, where S^* denotes the state after S feeds back through the lower AND gate. In the case discussed previously, the transition from $ABC = 110$ to 100, the glitch in S can break the feedback, leaving S low or unstable. The resulting sequential feedback circuit is thus unreliable.



Eliminating static hazards from two-level circuits is fairly straightforward. The Karnaugh map to the right corresponds to our original circuit; the solid lines indicate the implicants selected by the AND gates. A static-1 hazard is present when two adjacent 1s in the K-map are not covered by a common implicant. Static-0 hazards do not occur in two-level SOP circuits.



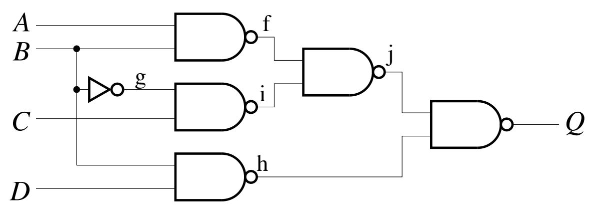
Eliminating static hazards requires merely extending the circuit with consensus terms in order to ensure that some AND gate remains high through every transition between input states with output 1.[3] In the K-map shown, the dashed line indicates the necessary consensus term, $A\bar{C}$.

[3]Hazard elimination is not in general simple; we have considered only two-level circuits.

Dynamic Hazards*

Consider an input transition for which we expect to see a change in an output. Under certain timing conditions, the output may not transition smoothly, but instead bounce between its original value and its new value before coming to rest at the new value. A circuit that might exhibit such behavior is said to contain a **dynamic hazard**. The qualifier "dynamic" refers to the expected change in the output.

Dynamic hazards appear only in more complex circuits, such as the one shown below. The output of this circuit is defined by the equation $Q = \bar{A}B + \bar{A}\bar{C} + \bar{B}\bar{C} + BD$.



Consider the transition from the input state $ABCD = 1111$ to 1011, in which B falls from 1 to 0. For simplicity, assume that each gate has a delay of 1 time unit. If B goes low at time $T = 0$, the table shows the progression over time of logic levels at several intermediate points in the circuit and at the output Q. Each gate merely produces the appropriate output based on its inputs in the previous time step. After one delay, the three gates with B as a direct input change their outputs (to stable, final values). After another delay, at $T = 2$, the other three gates respond to the initial changes and flip their outputs. The resulting changes induce another set of changes at $T = 3$, which in turn causes the output Q to change a final time at $T = 4$.

T	f	g	h	i	j	Q
0	0	0	0	1	1	1
1	1	1	1	1	1	1
2	1	1	1	0	0	0
3	1	1	1	0	1	1
4	1	1	1	0	1	0

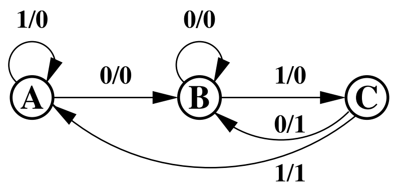
The output column in the table illustrates the possible impact of a dynamic hazard: rather than a smooth transition from 1 to 0, the output drops to 0, rises back to 1, and finally falls to 0 again. The dynamic hazard in this case can be attributed to the presence of a static hazard in the logic that produces intermediate value j.

Essential Hazards*

Essential hazards are inherent to the function of a circuit and may appear in any implementation. In sequential feedback circuit design, they must be addressed at a low level to ensure that variations in logic path lengths (**timing skew**) through a circuit do not expose them. With clocked synchronous circuits, essential hazards are abstracted into a single form: **clock skew**, or disparate clock edge arrival times at a circuit's flip-flops.

An example demonstrates the possible effects: consider the construction of a clocked synchronous circuit to recognize 0-1 sequences on an input IN. Output Q should be held high for one cycle after recognition, that is, until the next rising clock edge. A description of states and a state diagram for such a circuit appear below.

S_1S_0	state	meaning
00	A	nothing, 1, or 11 seen last
01	B	0 seen last
10	C	01 recognized (output high)
11	unused	



For three states, we need two ($= \lceil \log_2 3 \rceil$) flip-flops. Denote the internal state S_1S_0. The specific internal state values for each logical state (A, B, and C) simplify the implementation and the example. A **state table** and K-maps for the next-state logic appear below. The state table uses one line per state with separate columns for each input combination, making the table more compact than one with one line per state/input combination. Each column contains the full next-state information, including output. Using this form of the state table, the K-maps can be read directly from the table.

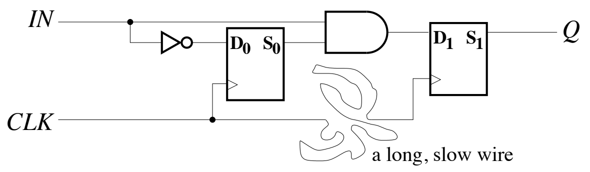
	IN	
S_1S_0	0	1
00	01/0	00/0
01	01/0	10/0
11	x	x
10	01/1	00/1

S_1^+	S_1S_0 00	01	11	10
IN 0	0	0	x	0
1	0	1	x	0

S_0^+	S_1S_0 00	01	11	10
IN 0	1	1	x	1
1	0	0	x	0

Q	S_1S_0 00	01	11	10
IN 0	0	0	x	1
1	0	0	x	1

Examining the K-maps, we see that the excitation and output equations are $S_1^+ = IN \cdot S_0$, $S_0^+ = \overline{IN}$, and $Q = S_1$. An implementation of the circuit using two D flip-flops appears below. Imagine that mistakes in routing or process variations have made the clock signal's path to flip-flop 1 much longer than its path into flip-flop 0, as illustrated.



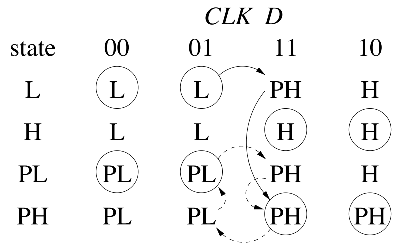
Due to the long delays, we cannot assume that rising clock edges arrive at the flip-flops at the same time. The result is called clock skew, and can make the circuit behave improperly by exposing essential hazards. In the logical B to C transition, for example, we begin in state $S_1S_0 = 01$ with $IN = 1$ and the clock edge rising. Assume that the edge reaches flip-flop 0 at time $T = 0$. After a flip-flop delay ($T = 1$), S_0 goes low. After another AND gate delay ($T = 2$), input D_1 goes low, but the second flip-flop has yet to change state! Finally, at some later time, the clock edge reaches flip-flop 1. However, the output S_1 remains at 0, leaving the system in state A rather than state C.

Fortunately, in clocked synchronous sequential circuits, all essential hazards are related to clock skew. This fact implies that we can eliminate a significant amount of complexity from circuit design by doing a good job of distributing the clock signal. It also implies that, as a designer, you should avoid specious addition of logic in a clock path, as you may regret such a decision later, as you try to debug the circuit timing.

Proof Outline for Clocked Synchronous Design*

This section outlines a proof of the claim made regarding clock skew being the only source of essential hazards for clocked synchronous sequential circuits. A **proof outline** suggests the form that a proof might take and provides some of the logical arguments, but is not rigorous enough to be considered a proof. Here we use a D flip-flop to illustrate a method for identifying essential hazards (*the D flip-flop has no essential hazards, however*), then argue that the method can be applied generally to collections of flip-flops in a clocked synchronous design to show that essential hazards occur only in the form of clock skew.

state		
low	L	clock low, last input low
high	H	clock high, last input low
pulse low	PL	clock low, last input high (output high, too)
pulse high	PH	clock high, last input high (output high, too)



	CLK D			
state	00	01	11	10
L	L	L	PH	H
H	L	L	H	H
PL	PL	PL	PH	H
PH	PL	PL	PH	PH

Consider the sequential feedback state table for a positive edge-triggered D flip-flop, shown above. In designing and analyzing such circuits, we assume that only one input bit changes at a time. The state table consists of one row for each state and one column for each input combination. Within a row, input combinations that have no effect on the internal state of the circuit (that is, those that do not cause any change in the state) are said to be stable; these states are circled. Other states are unstable, and the circuit changes state in response to changes in the inputs.

For example, given an initial state L with low output, low clock, and high input D, the solid arcs trace the reaction of the circuit to a rising clock edge. From the 01 input combination, we move along the column to the 11 column, which indicates the new state, PH. Moving down the column to that state's row, we see that the new state is stable for the input combination 11, and we stop. If PH were not stable, we would continue to move within the column until coming to rest on a stable state.

An essential hazard appears in such a table as a difference between the final state when flipping a bit once and the final state when flipping a bit thrice in succession. The dashed arcs in the figure illustrate the concept: after coming to rest in the PH state, we reset the input to 01 and move along the PH row to find a new state of PL. Moving up the column, we see that the state is stable. We then flip the clock a third time and move back along the row to 11, which indicates that PH is again the next state. Moving down the column, we come again to rest in PH, the same state as was reached after one flip. Flipping a bit three times rather than once evaluates the impact of timing skew in the circuit; if a different state is reached after two more flips, timing skew could cause unreliable behavior. As you can verify from the table, a D flip-flop has no essential hazards.

A group of flip-flops, as might appear in a clocked synchronous circuit, can and usually does have essential hazards, but only dealing with the clock. As you know, the inputs to a clocked synchronous sequential circuit consist of a clock signal and other inputs (either external of fed back from the flip-flops). Changing an input other than the clock can change the internal state of a flip-flop (of the master-slave variety), but flip-flop designs do not capture the number of input changes in a clock cycle beyond one, and changing an input three times is the same as changing it once. Changing the clock, of course, results in a synchronous state machine transition.

The detection of essential hazards in a clocked synchronous design based on flip-flops thus reduces to examination of the state machine. If the next state of the machine has any dependence on the current state, an essential hazard exists, as a second rising clock edge moves the system into a second new state. For a single D flip-flop, the next state is independent of the current state, and no essential hazards are present.