CHAPTER 3

Sequential Logic Design: Controllers

3.1 INTRODUCTION

The output of a combinational circuit is a function of the circuit's present inputs. A combinational circuit has no ***memory***—the feature of a circuit storing new bits and retaining those bits over time for later use. Combinational circuits alone are of limited usefulness. Designers typically use combinational circuits as part of larger circuits called sequential circuits—circuits that have memory. A ***sequential circuit*** is a circuit whose output depends not only on the circuit's present inputs, but also on the circuit's present ***state***, which is all the bits currently stored in the circuit. The circuit's state in turn depends on the past *sequence* of the circuit's input values.

An everyday sequential system example is a lamp that toggles (changes from off to on, or from on to off) when its button is pressed, as was shown in Figure 2.1(c). After plugging in the lamp, push the lamp's button (the input) a first time, and the lamp turns on. Push the button a second time, and the lamp turns off. Push the button a third time, and the lamp turns on again. The system's output (whether the lamp is on or off) depends on the input and on whether the system is currently in the *state* of the lamp being on or off. That state in turn depends on the past sequence of input values since the system was initially powered on. In contrast, an everyday combinational system example is a basic doorbell, as was shown in Figure 2.1(a). Push the button (the input) now, and the bell (the output) rings. Push the button again, and the bell rings again. Push the button tomorrow and the bell rings the same each time. A basic doorbell has no state—its output value (whether the bell rings or not) depends solely on its present input value (whether the button is pressed or not).

Most digital systems with which you are likely familiar involve sequential circuits. A calculator contains a sequential circuit to store the numbers you enter, in order to operate on those numbers. A digital camera stores pictures. A traffic light controller stores information indicating which light is presently green. A kitchen timer that counts down from a set time to zero stores the present count value, to know what the next value should be.

This chapter describes sequential circuit building blocks called flips-flops and registers, which can store bits. It then introduces a sequential circuit design process in which a designer first captures desired behavior, and then converts that behavior to a type of sequential circuit known as a controller, comprised of a register and combinational logic.

▶ 3.2 STORING ONE BIT—FLIP-FLOPS

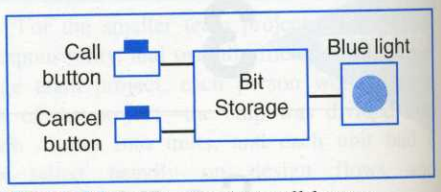


Figure 3.1 Flight attendant call-button system. Pressing *Call* turns on the light, which stays on after *Call* is released. Pressing *Cancel* turns off the light.

Sequential circuit design is aided by a building block that enables storing of a bit, much like combinational circuit design was aided by the AND, OR, and NOT gate building blocks. Storing a bit means that we can save either a `0` or a `1` in the block and later come back to see what was saved. For example, consider designing the flight attendant call-button system in Figure 3.1. An airline passenger can push the *Call* button to turn on a small blue light above the passenger's seat, indicating to a flight attendant that the passenger needs service. The light stays on even after the call button is released. The light can be turned off by pressing the *Cancel* button. Because the light must stay on even after the call button is released, a mechanism is needed to "remember" that the call button was pressed. That mechanism can be a bit storage block, in which a `1` will be stored when the call button is pressed, and a `0` stored when the cancel button is pressed. The inputs of this bit storage block will be connected to the call and cancel buttons, and the output to the blue light, as in Figure 3.1. The light illuminates when the block's output is `1`.

This section introduces the internal design of such a bit storage block by introducing several increasingly complex circuits able to store a bit—a basic SR latch, a level-sensitive SR latch, a level-sensitive D latch, and an edge-triggered D flip-flop. The D flip-flop will then be used to create a block capable of storing multiple bits, known as a register, which will serve as the main bit storage block in the rest of the book. Each successive circuit eliminates some problem of the previous one. Be aware that designers today rarely use bit storage blocks other than D flip-flops. We introduce the other blocks to provide the reader with an underlying intuition of the D flip-flop's internal design.

Feedback—The Basic Storage Method

The basic method used to store a bit in a digital circuit is ***feedback***. You've surely experienced feedback in the form of audio feedback, when someone talking into a microphone stood in front of the speaker, causing a loud continuous humming sound to come out of the speakers (in turn causing everyone to cover their ears and snicker). The talker generated a sound that was picked up by the microphone, came out of the speakers (amplified), was picked up *again* by the microphone, came out the speakers again (amplified even more), etc. That's feedback.

Feedback in audio systems is annoying, but in digital systems is extremely useful. Intuitively, we know that somehow the output of a logic gate must feed back into the gate itself, so that the stored bit ends up looping around and around, like a dog chasing its own tail. We might try the circuit in Figure 3.2.

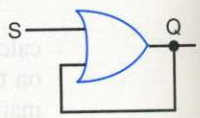


Figure 3.2 First (failed) attempt at using feedback to store a bit.

Suppose initially `Q` is `0` and `S` is `0`. At some point, suppose we set `S` to `1`. That causes `Q` to become `1`, and that `1` feeds back into the OR gate, causing `Q` to be `1`, etc. So even when `S` returns

Basic SR

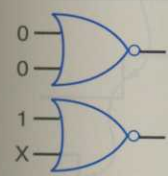
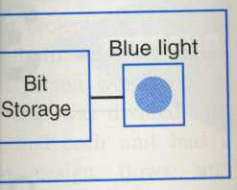


Figure 3.5 NOR behavior.



ndant call-button
l turns on the light,
Call is released.
s off the light.

a flight attendant that
button is released. The
light must stay on even
ber" that the call button
a 1 will be stored when
pressed. The inputs of
tons, and the output to
ck's output is 1.
ge block by introducing
R latch, a level-sensitive
op. The D flip-flop will
own as a register, which
successive circuit elim-
ers today rarely use bit
ks to provide the reader

You've surely experienced
o a microphone stood in
me out of the speakers (in
nerated a sound that was
as picked up *again* by the
That's feedback.

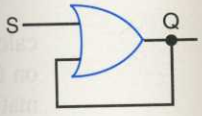


Figure 3.2 First (failed) attempt at using feedback to store a bit.

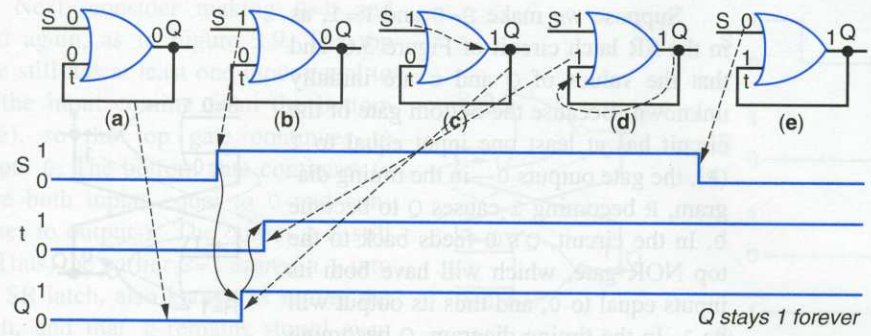


Figure 3.3 Tracing the behavior of our first attempt at bit storage.

to 0, Q stays 1. Unfortunately, Q stays 1 from then on, and we have no way of resetting Q to 0. But hopefully you understand the basic idea of feedback now—we did successfully store a 1 using feedback, but we couldn't store a 0 again.

Figure 3.3 shows the timing diagram for the feedback circuit of Figure 3.2. Initially, we set both OR gate inputs to 0 (Figure 3.3(a)). Then we set S to 1 (Figure 3.3(b)), which causes Q to become 1 slightly later (Figure 3.3(c)), assuming the OR gate has a small delay as discussed in Section 2.10. Q becoming 1 causes t to become 1 slightly later (Figure 3.3(d)), assuming the wire has a small delay too. Q will stay at 1. Finally, when we change S back to 0 (Figure 3.3(e)), Q will continue to stay 1 because t is 1. The first curved line with an arrow indicates that the event of S changing from 0 to 1 causes the event of Q changing from 0 to 1. An ***event*** is any change on a bit signal from 0 to 1 or from 1 to 0. The second curved line with an arrow indicates that the event of Q changing from 0 to 1 in turn causes the event of t changing from 0 to 1. That 1 then continues to loop around, forever, with no way for S to reset Q to 0.

Basic SR Latch

It turns out that the simple circuit in Figure 3.4, called a ***basic SR latch***, implements the bit storage building block that we seek. The circuit consists of a pair of cross-coupled NOR gates. Making the circuit's S input equal to 1 causes Q to become 1, while making R equal to 1 causes Q to become 0. Making both S and R equal to 0 causes Q's current value to keep looping around. In other words, S "sets" the latch to 1, and R "resets" the latch to 0—hence the letters S (for *set*) and R (for *reset*).

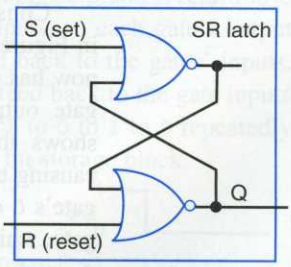


Figure 3.4 Basic SR latch.

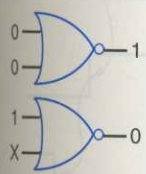


Figure 3.5 NOR behavior.

Let's see why the basic SR latch works as it does. Recall that a NOR gate outputs 1 only when all the gate's inputs equal 0, as shown in Figure 3.5; if at least one input equals 1, the NOR gate outputs 0.

Suppose we make S=0 and R=1, as in the SR latch circuit of Figure 3.6, and that the values of Q and t are initially unknown. Because the bottom gate of the circuit has at least one input equal to 1 (R), the gate outputs 0—in the timing diagram, R becoming 1 causes Q to become 0. In the circuit, Q's 0 feeds back to the top NOR gate, which will have both its inputs equal to 0, and thus its output will be 1. In the timing diagram, Q becoming 0 causes t to become 1. In the circuit, that 1 feeds back to the bottom NOR gate, which has at least one input (actually, both) equal to 1, so the bottom gate continues to output 0. Thus the output Q equals 0, and all values are ***stable,*** meaning the values won't change as long as no external input changes.

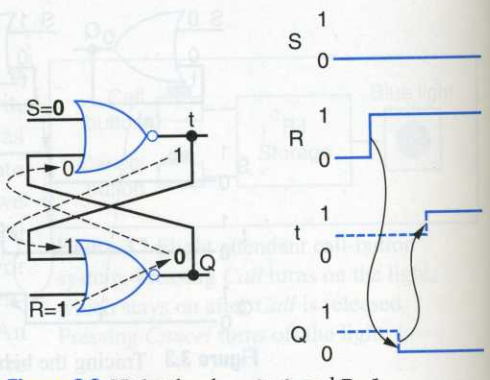


Figure 3.6 SR latch when S=0 and R=1.

Now suppose we keep S=0 and change R from 1 back to 0, as in Figure 3.7. The bottom gate still has at least one input equal to 1 (the input coming from the top gate), so the bottom gate continues to output 0. The top gate continues to have both inputs equal to 0 and continues to output 1. The output Q will thus still be 0. Therefore, the earlier R=1 *stored* a 0 into the SR latch, also known as *resetting* the latch, and that 0 remains stored even when R is changed back to 0. Note that R=1 will reset the latch regardless of the initial value of Q.

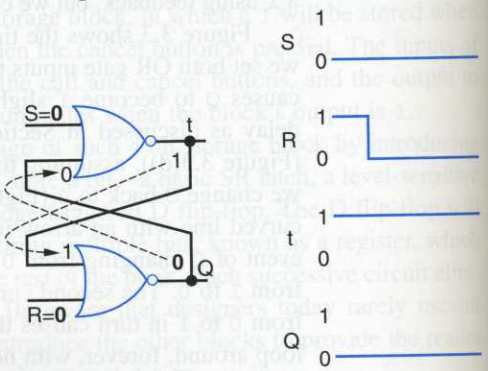


Figure 3.7 SR latch when S=0 and R=0, after R was previously 1.

Consider making S=1 and R=0, as in Figure 3.8. The top gate in the circuit now has one input equal to 1, so the top gate outputs a 0—the timing diagram shows the change of S from 0 to 1, causing t to change from 1 to 0. The top gate's 0 output feeds back to the bottom gate, which now has both inputs equal to 0 and thus outputs 1—the timing diagram shows the change of t from 1 to 0, causing Q to change from 0 to 1. The bottom gate's 1 output (Q) feeds back to the top gate, which has at least one input (actually, both of its inputs) equal to 1, so the top gate continues to output 0. The output Q therefore equals 1, and all values are stable.

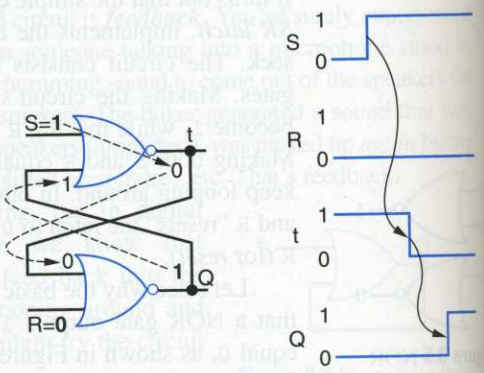
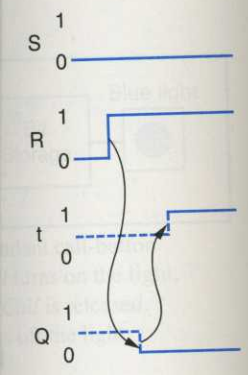
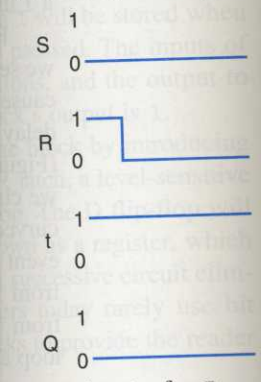


Figure 3.8 SR latch when S=1 and R=0.

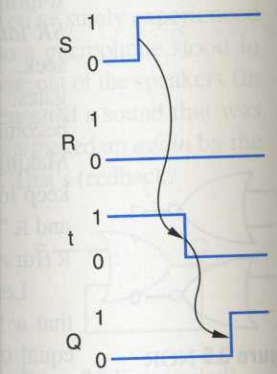


S=0 and R=1.

output Q equals 0, and
external input changes.



n S=0 and R=0, after R



hen S=1 and R=0.

herefore equals 1, and all

Next, consider making S=0 and R=0 again, as in Figure 3.9. The top gate still has at least one input equal to 1 (the input coming from the bottom gate), so the top gate continues to output 0. The bottom gate continues to have both inputs equal to 0 and continues to output 1. The output Q is still 1. Thus, the earlier S=1 *stored* a 1 into the SR latch, also known as *setting* the latch, and that 1 remains stored even when we return S to 0. Note that S=1 will set the latch regardless of the initial value of Q.

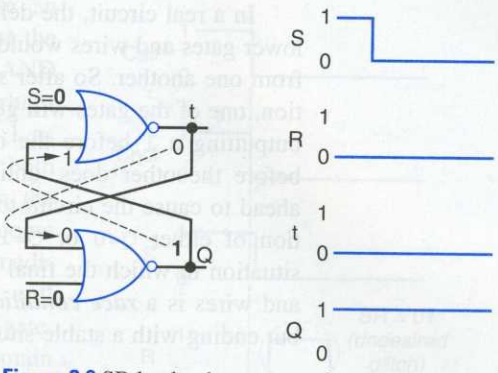


Figure 3.9 SR latch when S=0 and R=0, after S was previously 1.

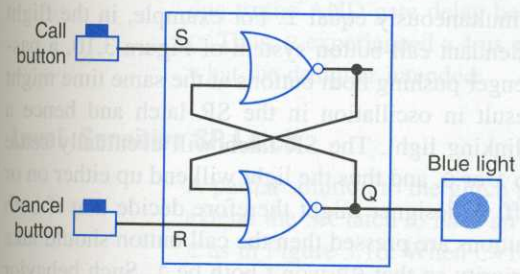


Figure 3.10 Flight attendant call-button system using a basic SR latch.

The basic SR latch can be used to implement the flight attendant call-button system as shown in Figure 3.10, by connecting the call button to S, the cancel button to R, and Q to the light. Pressing the call button sets Q to 1, thus turning on the light. Q stays 1 even when the call button is released. Pressing the cancel button resets Q to 0, thus turning off the light. Q stays 0 even when the cancel button is released.

Problem when SR=11 in a Basic SR latch

A problem with the basic SR latch is that if S and R are both 1, undefined behavior results—the latch might store a 1, it might store a 0, or its output might oscillate, changing from 1 to 0 to 1 to 0, and so on. In particular, if S = 1 and R = 1 (written as "SR=11" for short), both the NOR gates have at least one input equal to 1, and thus both gates output 0, as in Figure 3.11(a). A problem occurs when S and R are made 0 again. Suppose S and R return to 0 at the same time. Then both gates will have 0s at all their inputs, so each gate's output will change from 0 to 1, as in Figure 3.11(b). Those 1s feed back to the gates' inputs, causing the gates to output 0s, as in Figure 3.11(c). Those 0s feed back to the gate inputs again, causing the gates to output 1s. And so on. Going from 1 to 0 to 1 to 0 repeatedly is called ***oscillation***. Oscillation is not a desirable feature of a bit storage block.

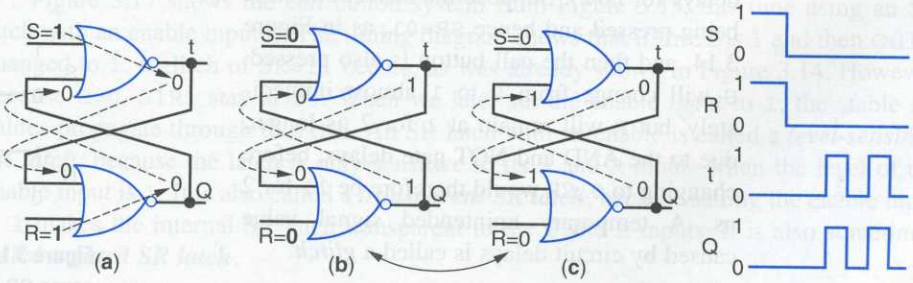


Figure 3.11 The situation of S=1 and R=1 causes problems—Q oscillates when SR return to 00.

In a real circuit, the delays of the upper and lower gates and wires would be slightly different from one another. So after some time of oscillation, one of the gates will get ahead of the other, outputting a `1` before the other does, then a `0` before the other does, until it gets far enough ahead to cause the circuit to enter a stable situation of either `Q=0` or `Q=1`. Which situation will happen is unknown beforehand. A situation in which the final output of a sequential circuit depends on the delays of gates and wires is a ***race condition***. Figure 3.12 shows a race condition involving oscillation but ending with a stable situation of `Q=1`.

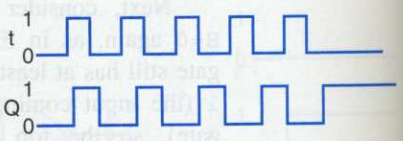


Figure 3.12 Q eventually settles to either `0` or `1`, due to race condition.

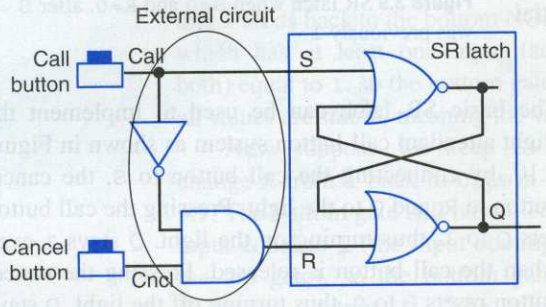


Figure 3.13 Circuit added external to SR latch striving to prevent `SR=11` when both buttons are pressed.

Therefore, `S` and `R` must *never* be allowed to simultaneously equal `1` in an SR latch. A designer using an SR latch should add a circuit external to the SR latch that strives to ensure that `S` and `R` never simultaneously equal `1`. For example, in the flight attendant call-button system of Figure 3.10, a passenger pushing both buttons at the same time might result in oscillation in the SR latch and hence a blinking light. The SR latch will eventually settle to `1` or `0`, and thus the light will end up either on or off. A designer might therefore decide that if both buttons are pressed then the call button should take priority so that `SR` won't both be `1`. Such behavior can be achieved using a combinational circuit in front of `S` and `R`, as shown in Figure 3.13. `S` should be `1` if the call button (denoted as `Call`) is pressed and either the cancel button (`Cncl`) is pressed or not pressed, so `S = Call*Cncl + Call*Cncl' = Call`. `R` should be `1` only if the cancel button is pressed *and* the call button is *not* pressed, meaning `R = Cncl * Call'`. The circuit in Figure 3.13 is derived directly from these equations.

Even with such an external circuit, `S` and `R` could still inadvertently both become `1` due to the delay of real gates (see Section 2.10). Assume the AND and NOT gates in Figure 3.13 have delays of 1 ns each (ignore wire delays for now). Suppose the cancel button is being pressed and hence `SR=01`, as in Figure 3.14, and then the call button is also pressed. `S` will change from `0` to `1` almost immediately, but `R` will remain at `1` for 2 ns longer, due to the AND and NOT gate delays, before changing to `0`. `SR` would therefore be `11` for 2 ns. A temporary unintended signal value caused by circuit delays is called a ***glitch***.

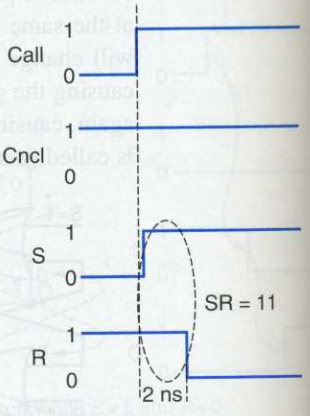


Figure 3.14 Gate delays can cause `SR=11`.

Significantly, glitches can also cause an unintended latch set or reset. Assume that the wire connecting the cancel button to the AND gate in Figure 3.13 has a delay of 4 ns (perhaps the wire is very long), in addition to the 1 ns AND and NOT gate delays. Suppose both buttons are pressed, so SR=10, and then the buttons are both released—SR should become 00. S will indeed change to 0 almost immediately. The top input of the AND gate will become 1 after the 1 ns delay of the NOT gate. The bottom input of that AND gate will remain 1 for 3 ns more, due to the 4 ns wire delay, thus causing R to change 1. After that bottom input finally changes to 0, yet another 1 ns will pass due to the AND gate delay before R returns to 0. Thus, R experienced a 4 ns glitch, which resets the latch to 0—yet a reset is clearly not what the designer intended.

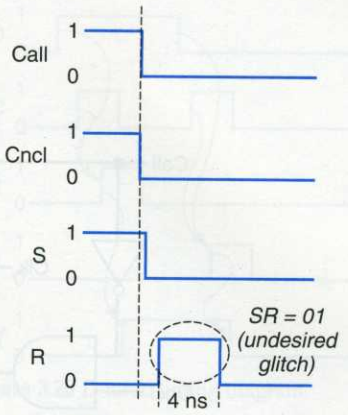


Figure 3.15 Wire delay leading to a glitch causing a reset.

Level -Sensitive SR Latch

A partial solution to the glitch problem is to extend the SR latch to have an *enable* input C as in Figure 3.16. When C=1, the S and R signals pass through the two AND gates to the S1 and R1 inputs of the basic SR latch, because S*1=S and R*1=R. The latch is enabled. But when C=0, the two AND gates cause S1 and R1 to be 0, regardless of the values of S and R. The latch is disabled. The enable input can be set to 0 when S and R might change so that glitches won't propagate through to S1 and R1, and then set to 1 only when S and R are stable. The question then remains of when to set the enable input to 1. That question will be answered in the upcoming sections.

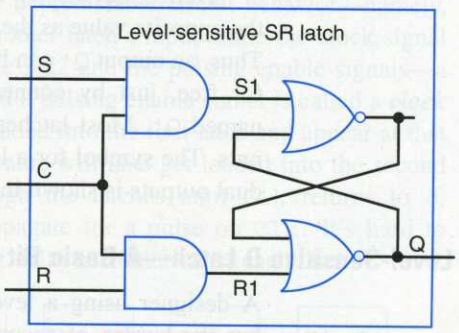
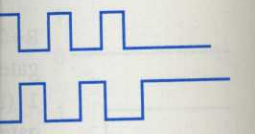


Figure 3.16 Level-sensitive SR latch—an SR latch with enable input C.

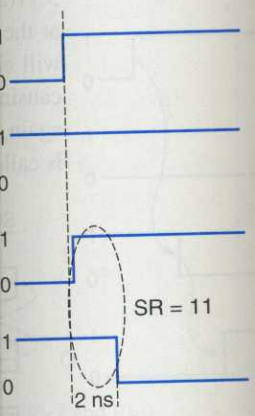
Figure 3.17 shows the call button system from Figure 3.13, this time using an SR latch with an enable input C. The timing diagram shows that if Cncl is 1 and then Call is changed to 1, a glitch of SR=11 occurs, as was already shown in Figure 3.14. However, because C=0, S1R1 stay at 00. When we later set the enable input to 1, the stable SR values propagate through to S1R1. An SR latch with an enable is called a ***level-sensitive SR latch***, because the latch is only sensitive to its S and R inputs when the level of the enable input is 1. It is also called a ***transparent SR latch***, because setting the enable input to 1 makes the internal SR latch transparent to the S and R inputs. It is also sometimes called a ***gated SR latch***.



ntually settles to either
e condition.

known beforehand. A
on the delays of gates
n involving oscillation

er be allowed to simul-
atch. A designer using
cuit external to the SR
that S and R never
example, in the flight
of Figure 3.10, a pas-
at the same time might
SR latch and hence a
n will eventually settle
will end up either on or
fore decide that if both
call button should take
th be 1. Such behavior
ombinational circuit in
call button (denoted as
sed or not pressed, so
y if the cancel button is
Call'. The circuit in



e 3.14 Gate delays can
e SR=11.

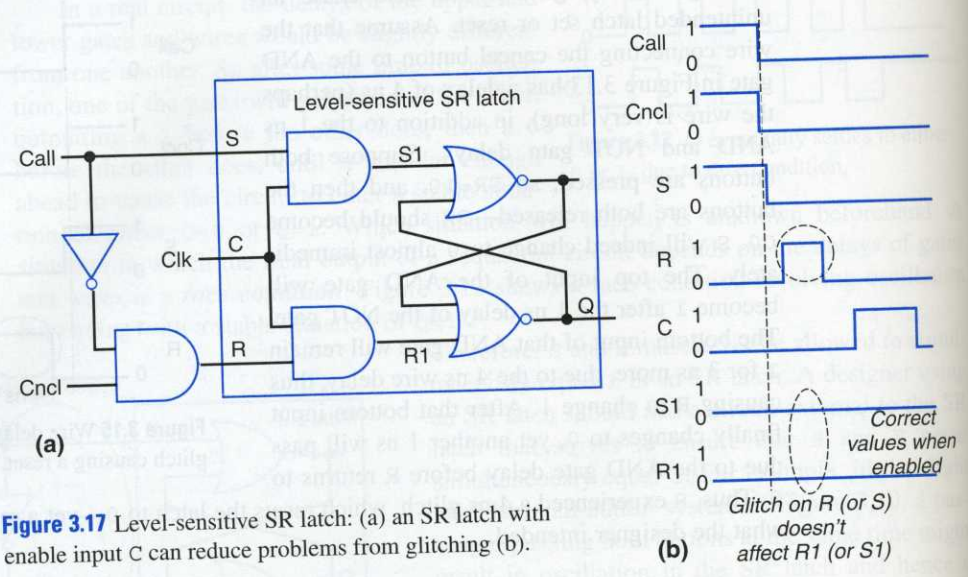


Figure 3.17 Level-sensitive SR latch: (a) an SR latch with enable input C can reduce problems from glitching (b).

Notice that the top NOR gate of an SR latch outputs the opposite value as the bottom NOR gate that outputs Q. Thus, an output Q' can be included on an SR latch almost for free, just by connecting the top gate to an output named Q'. Most latches come with both Q and Q' outputs. The symbol for a level-sensitive SR latch with such dual outputs is shown in Figure 3.18.

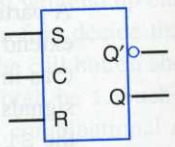


Figure 3.18 Symbol for dual-output level-sensitive SR latch.

Level-Sensitive D Latch—A Basic Bit Store

A designer using a level-sensitive SR latch has the burden of ensuring that S and R are never simultaneously 1 when the enable input is 1. One way to relieve designers of this burden is to introduce another type of latch, called a ***level-sensitive D latch*** (also known as a ***transparent D latch*** or ***gated D latch***), shown in Figure 3.19. Internally, the latch's D input connects directly to the S input of a level-sensitive SR latch, and connects through an inverter to the R input of the SR latch. The D latch is thus either setting (when D=1) or resetting (when D=0) its internal basic SR latch when the enable input C is 1.

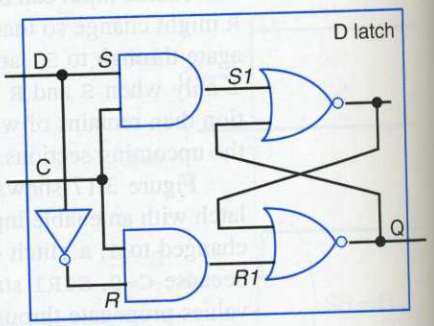
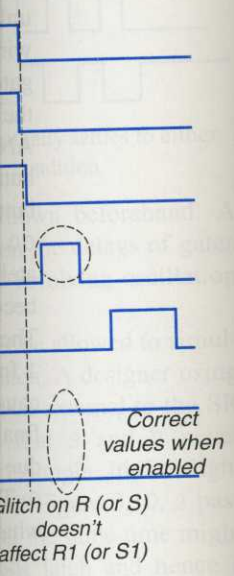
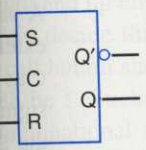
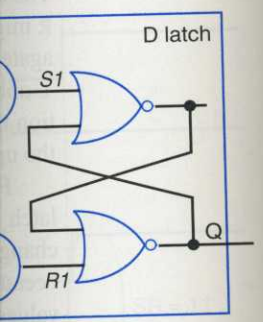


Figure 3.19 D latch internal circuit.





3.18 Symbol for dual-
level-sensitive SR latch.



ch internal circuit.

A level-sensitive D latch thus stores whatever value is present at the latch's D input when C = 1, and remembers that value when C = 0. Figure 3.20 shows a timing diagram of a D latch for sample input values on D and C; arrows indicate which signal changes cause other signals to change. When D is 1 and C is 1, the latch is set to 1, because S1 is 1 and R1 is 0. When D is 0 and C is 1, the latch is reset to 0, because R1 is 1 and S1 is 0. By making R the opposite of S, the D latch ensures that S and R won't both be 1 at the same time, as long as D is only changed when C is 0 (even if changed when C is 1, the inverter's delay could cause S and R to both be 1 briefly, but for too short of time to cause a problem).

The symbol for a D latch with dual-outputs (Q and Q') is shown in Figure 3.21.

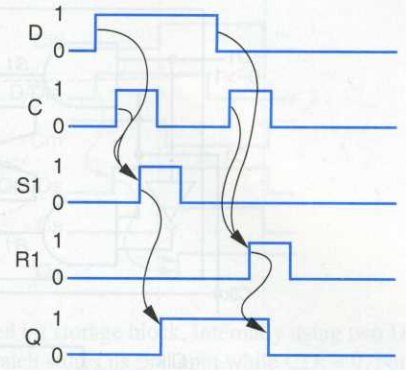


Figure 3.20 D latch timing diagram.

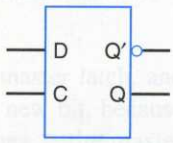


Figure 3.21 D latch symbol.

Edge-Triggered D Flip-Flop—A Robust Bit Store

The D latch still has a problem that can cause unpredictable circuit behavior—namely, signals can propagate from a latch output to another latch's input while the clock signal is 1. For example, consider the circuit in Figure 3.22 and the pulsing enable signals—a ***pulse*** is a change from 0 to 1 and back to 0, and a pulsing enable signal is called a ***clock*** signal. When Clk = 1, the value on Y will be loaded into the first latch and appear at that latch's output. If Clk still equals 1, then that value will also get loaded into the second latch. The value will keep propagating through the latches until Clk returns to 0. Through how many latches will the value propagate for a pulse on Clk? It's hard to say—we would have to know the precise timing delay information of each latch.

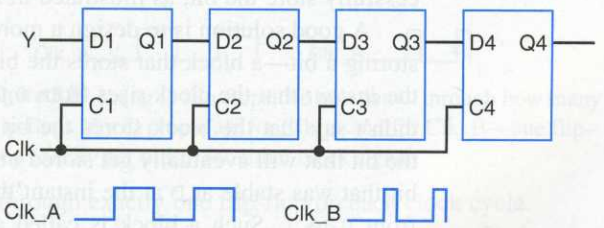


Figure 3.22 A problem with latches—through how many latches will Y propagate for each pulse of Clk_A? For Clk_B?

Figure 3.23 illustrates this propagation problem in more detail. Suppose D1 is initially 0 for a long time, changes to 1 long enough to be stable, and then Clk becomes 1. Q1 will thus change from 0 to 1 after about three gate delays, and thus D2 will also change from 0 to 1, as shown in the left timing diagram. If Clk is still 1, then that new value for D2 will propagate through the AND gates of the second latch, causing S2 to change from 0 to 1 and R2 from 1 to 0, thus changing Q2 from 0 to 1, as shown in the left timing diagram.

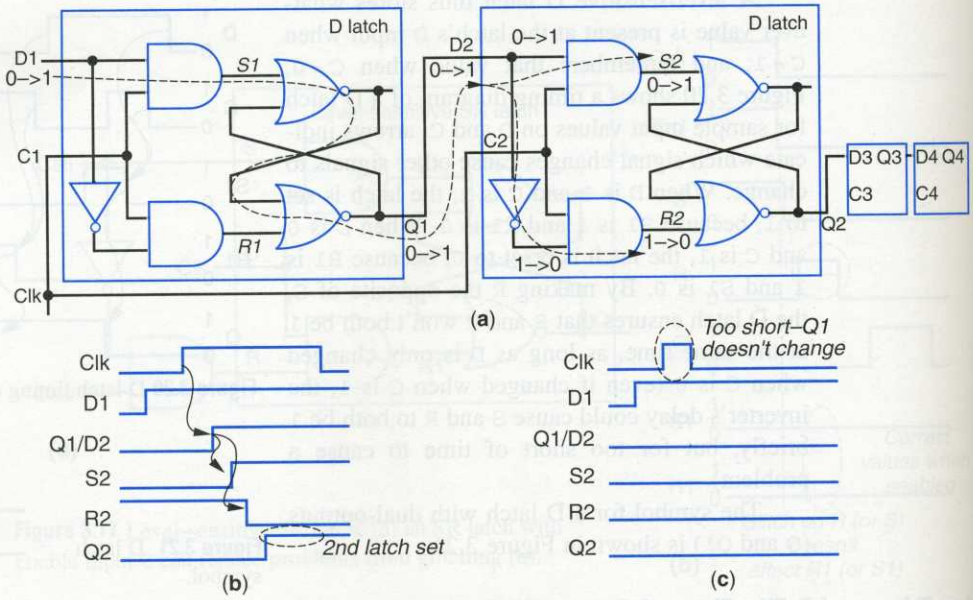


Figure 3.23 A problem with level-sensitive latches: (a) while `C = 1`, `Q1`'s new value may propagate to `D2`, (b) such propagation can cause an unknown number of latches along a chain to get updated, (c) trying to shorten the clock's time at `1` to avoid propagation to the next latch, but long enough to allow a latch to reach a stable feedback situation, is hard because making the clock's high time too short prevents proper loading of the latch.

You might suggest making the clock signal such that the clock is `1` only for a short amount of time, so there's not enough time for the new output of a latch to propagate to the next latch's inputs. But how short is short enough? 50 ns? 10 ns? 1 ns? 0.1 ns? And if we make the clock's time at `1` too short, that time may not be long enough for the bit at a latch's `D` input to stabilize in the latch's feedback circuit, and we might therefore not successfully store the bit, as illustrated in Figure 3.23(c).

A good solution is to design a more robust block for storing a bit—a block that stores the bit at the `D` input at the *instant* that the clock rises from `0` to `1`. Note that we didn't say that the block stores the bit instantly. Rather, the bit that will eventually get stored into the block is the bit that was stable at `D` at the instant that the clock rises from `0` to `1`. Such a block is called an ***edge-triggered D flip-flop***. The word "edge" refers to the vertical part of the line representing the clock signal, when the signal transitions from `0` to `1`. Figure 3.24 shows three cycles of a clock signal, and indicates the three rising clock edges of those cycles.

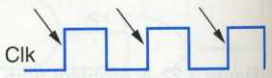


Figure 3.24 Rising clock edges.

Edge-Triggered D Flip-Flop Using a Master-Servant Design. One way to design an edge-triggered D flip-flop is to use *two* D latches, as shown in Figure 3.25.

The first D latch, known as the ***master***, is enabled (can store new values on `Dm`) when `Clk` is `0` (due to the inverter), while the second D latch, known as the ***servant***, is enabled

The common name is actually "master-slave." Some choose instead to use the term "servant," due to many people finding the term "slave" offensive. Others use the terms "primary-secondary."

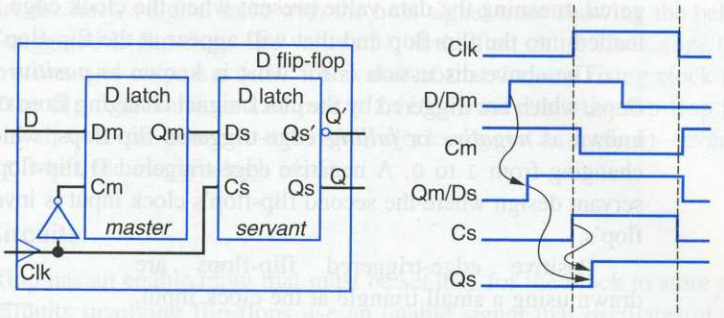


Figure 3.25 A D flip-flop implementing an edge-triggered bit storage block, internally using two D latches in a master-servant arrangement. The master D latch stores its `Dm` input while `Clk = 0`, but the new value appearing at `Qm`, and hence at `Ds`, does not get stored into the servant latch, because the servant latch is disabled when `Clk = 0`. When `Clk` becomes `1`, the servant D latch becomes enabled and thus gets loaded with whatever value was in the master latch at the instant that `Clk` changed from `0` to `1`.

when `Clk` is `1`. Thus, while `Clk` is `0`, the bit on `D` is stored into the master latch, and hence `Qm` and `Ds` are updated—but the servant latch does not store this new bit, because the servant latch is not enabled since `Clk` is not `1`. When `Clk` becomes `1`, the master latch becomes disabled, thus holding whatever bit was at the `D` input just before the clock changed from `0` to `1`. Also, when `Clk` is `1`, the servant latch becomes enabled, thus storing the bit that the master is storing, and that bit is the bit that was at the `D` input just before `Clk` changed from `0` to `1`. The two latches thus implement an edge-triggered storage block—the bit that was at the input when `Clk` changed from `0` to `1` gets stored.

The edge-triggered block using two internal latches thus prevents the stored bit from propagating through more than one flip-flop when the clock is `1`. Consider the chain of flip-flops in Figure 3.26, which is similar to the chain in Figure 3.22 but with D flip-flops in place of D latches.

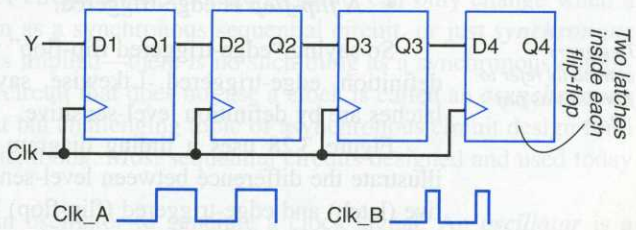


Figure 3.26 Using D flip-flops, we now know through how many flip-flops Y will propagate for Clk_A and for Clk_B—one flip-flop exactly per pulse, for either clock signal.

We know that `Y` will propagate through exactly one flip-flop on each clock cycle.

The drawback of a master-servant approach is that two D latches are needed to store one bit. Figure 3.26 shows four flip-flops, but there are two latches inside each flip-flop, for a total of eight latches.

Many alternative methods exist other than the master-servant method for designing an edge-triggered flip-flop. In fact, there are hundreds of different designs for latches and flip-flops beyond the designs shown above, with those designs differing in terms of their size, speed, power, etc. When using an edge-triggered flip-flop, a designer usually doesn't consider whether the flip-flop achieves edge-triggering using the master-servant method or using some other method. The designer need only know that the flip-flop is edge-trig-

The common name is actually "master-slave." Some choose instead to use the term "servant," due to many people finding the term "slave" offensive. Others use the terms "primary-secondary."

gered, meaning the data value present when the clock edge is rising is the value that gets loaded into the flip-flop and that will appear at the flip-flop's output some time later.

The above discussion is for what is known as ***positive*** or ***rising*** edge-triggered flip-flops, which are triggered by the clock signal changing from `0` to `1`. There are also flip-flops known as ***negative*** or ***falling*** edge-triggered flip-flops, which are triggered by the clock changing from `1` to `0`. A negative edge-triggered D flip-flop can be built using a master-servant design where the second flip-flop's clock input is inverted, rather than the first flip-flop's.

Positive edge-triggered flip-flops are drawn using a small triangle at the clock input, and negative edge-triggered flip-flops are drawn using a small triangle along with an inversion bubble, as shown in Figure 3.27. Because those symbols identify the clock input, those inputs typically are not given a name.

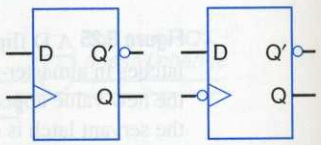


Figure 3.27 Positive (shown on the left) and negative (right) edge-triggered D flip-flops. The sideways triangle input represents an edge-triggered clock input.

Bear in mind that although the master-servant design doesn't change the output until the falling clock edge, the flip-flop is still positive edgetriggered, because the flip-flop stored the value that was at the `D` input at the instant that the clock edge was *rising*.

Latches versus Flip-Flops: Various textbooks define the terms latch and flip-flop differently. We'll use what seems to be the most common convention among designers, namely:

- A ***latch*** is level-sensitive, and
- A ***flip-flop*** is edge-triggered.

Designers commonly refer to flip-flops as just "flops."

So saying "edge-triggered flip-flop" would be redundant, since flip-flops are, by this definition, edge-triggered. Likewise, saying "level-sensitive latch" is redundant, since latches are by definition level-sensitive.

Figure 3.28 uses a timing diagram to illustrate the difference between level-sensitive (latch) and edge-triggered (flip-flop) bit storage blocks. The figure provides an example of a clock signal and a value on a signal `D`. The next signal trace is for the `Q` output of a `D` latch, which is level-sensitive. The latch ignores the first pulse on `D` (labeled as *3* in the figure) because `Clk` is low. However, when `Clk` becomes high (*1*), the latch output follows the `D` input, so when `D` changes from `0` to `1` (*4*), so does the latch output (*7*). The latch ignores the next changes on `D` when `Clk` is low (*5*), but then follows `D` again when `Clk` is high (*6, 8*).

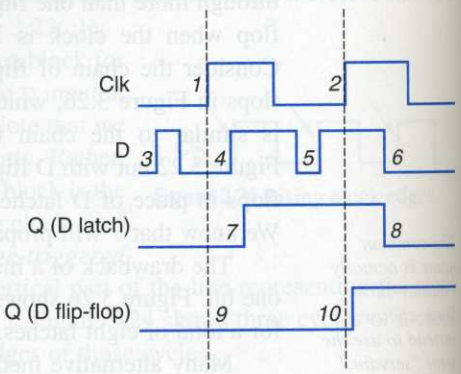


Figure 3.28 Latch versus flip-flop timing.

Compare the latch's signal trace with the next signal trace showing the behavior of a rising-edge-triggered D flip-flop. The value of `D` at the first rising clock edge (*1*) is `0`, so the flip-flop stores and outputs a `0` (*9*). The value of `D` at the next rising clock edge (*2*) is `1`, and thus the flip-flop stores and outputs a `1` (*10*). Notice that the flip-flop ignores all changes to `D` that occur between the rising clock edges (*3*, *4*, *5*, *6*)—even ignoring changes on `D` when the clock is high (*4*, *6*).

Clocks and Synchronous Circuits

The D flip-flop has an enable input that must be set to `1` for the block to store a bit. Most sequential circuits involving flip-flops use an enable signal that oscillates at a constant rate. For example, the enable signal could go high for 10 ns, then low for 10 ns, then high for 10 ns, then low for 10 ns, and so on, as in Figure 3.29. The time high and time low need not be the same, though in practice it usually is. (This book commonly shows the high time as shorter, to enhance figure readability).

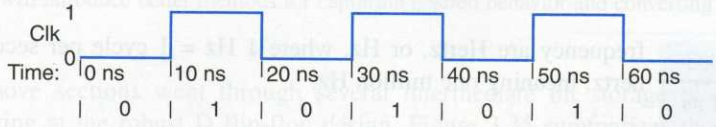


Figure 3.29 An example of a clock signal named `Clk`. Circuit inputs should only change while `Clk = 0`, such that latch inputs will be stable when `Clk` rises to `1`.

An oscillating enable signal is called a ***clock*** signal, because the signal ticks (high, low, high, low) like a clock. A circuit whose storage elements can only change when a clock signal is active is known as a synchronous sequential circuit, or just ***synchronous circuit*** (the sequential aspect is implied—there is no such thing as a synchronous combinational circuit). A sequential circuit that does not use a clock is called an ***asynchronous circuit.*** We leave the important but challenging topic of asynchronous circuit design for a more advanced digital design textbook. Most sequential circuits designed and used today are synchronous.

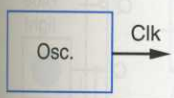


Figure 3.30 Oscillator component.

Designers typically use an oscillator to generate a clock signal. An ***oscillator*** is a digital component that outputs a signal alternating between `1` and `0` at a constant frequency, like that in Figure 3.29. An oscillator component typically has no inputs (other than power) and has an output representing the clock signal as in Figure 3.30.

A clock signal's ***period*** is the time after which the signal repeats itself—or more simply, the time between successive `1`s. The signal in Figure 3.29 has a period of 20 ns. A ***clock cycle***, or just ***cycle***, refers to one such segment of time, meaning one segment where the clock is `1` and then `0`. Figure 3.29 shows three and a half clock cycles. A clock signal's ***frequency*** is the number of cycles per second, and is computed as 1/(the clock period). The signal in Figure 3.29 has a frequency of 1/20 ns = 50 MHz. The units of

▶ HOW DOES IT WORK?—QUARTZ OSCILLATORS.

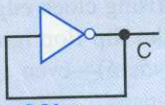


Figure 3.31 Conceptual oscillator implementation.

Conceptually, an oscillator can be thought of as an inverter feeding back to itself, as in Figure 3.31. If C is initially 1, the value will feed back through the inverter, and so C will become 0, which feeds back through the inverter, causing C to become 1 again, and so on. The oscillation frequency would depend on the delay of the inverter. Real oscillators must regulate the oscillation frequency more precisely. A common type of oscillator uses ***quartz***, a mineral consisting of silicon dioxide in crystal form. Quartz happens to vibrate if an electric current is applied to it, and that vibration is at a precise frequency determined by the quartz size and shape. Furthermore, when quartz vibrates, it generates a voltage. So by making quartz a specific size and shape and then applying a current, we get a precise electronic oscillator. The oscillator can be attached to an IC's clock signal input, as in Figure 3.32. Some ICs come with a built-in oscillator.

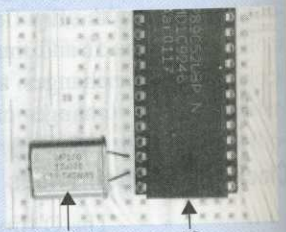


Figure 3.32 Oscillator providing a clock signal to an IC.

Freq.	Period
100 GHz	0.01 ns
10 GHz	0.1 ns
1 GHz	**1 ns**
100 MHz	10 ns
10 MHz	100 ns

Figure 3.33 Common frequency and period relationships.

frequency are Hertz, or Hz, where 1 Hz = 1 cycle per second. MHz is short for megahertz, meaning one million Hz.

A convenient way to mentally convert common clock periods to frequencies, and vice versa, is to remember that a 1 ns period equals a 1 GHz (gigahertz, meaning 1 billion Hz) frequency. Then, as shown in Figure 3.33, if one is slower (or faster) by a factor of 10, the other is slower (or faster) by a factor of 10 also—so a 10 ns period equals 100 MHz, while a 0.1 ns period equals 10 GHz.

Example 3.1 Flight attendant call-button using a D flip-flop

Let's now design the earlier-introduced flight attendant call-button system using a D flip-flop. If the call button is pressed, a 1 should be stored. If the cancel button is pressed, a 0 should be stored. If both buttons are pressed, we'll give priority to the call button, so a 1 should be stored. If neither button is pressed, the present value of Q should be stored back into the flip-flop. From this description, we see that a combinational circuit can be used to set the D flip-flop's input. The circuit's inputs will be Call, Cncl, and Q, and the output will be D, as shown in Figure 3.34(a).

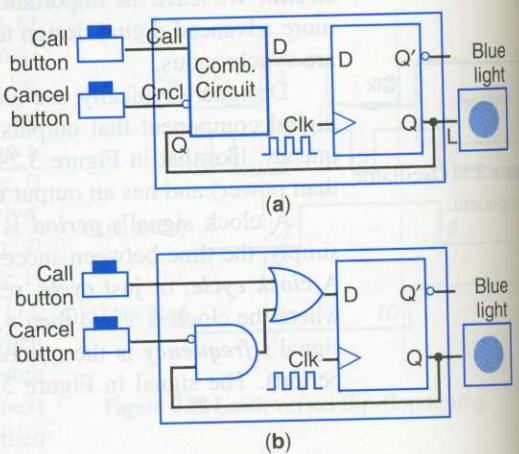


Figure 3.34 Flight attendant call-button system: (a) block diagram, and (b) implemented using a D flip-flop.

The circuit's desired behavior can be captured as the truth table in Table 3.1. If `Call=0` and `Cncl=0` (the first two rows), `D` equals `Q`'s value. If `Call=0` and `Cncl=1` (the next two rows), `D=0`. If `Call=1` and `Cncl=0` (the next two rows), `D=1`. And if both `Call=1` and `Cncl=1` (the last two rows), the `Call` button gets priority, so `D=1`.

TABLE 3.1 D truth table for call-button system.

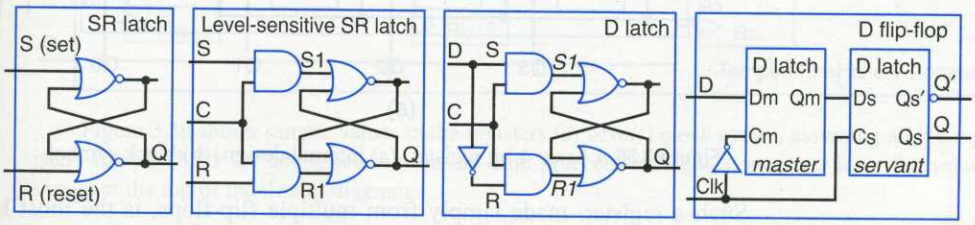
Call	Cncl	Q	D
0	0	0	0
0	0	1	1
0	1	0	0
0	1	1	0
1	0	0	1
1	0	1	1
1	1	0	1
1	1	1	1

After some algebraic simplification, we obtain the following equation for `D`:

```
D = Cncl'Q + Call
```

We can then convert the equation to the circuit shown in Figure 3.34(b). That circuit is more robust than the earlier circuit using an SR latch in Figure 3.10. But it is still not as good as it could be; Section 3.5 will explain why we might want to add additional flip-flops at the `Call` and `Cncl` inputs. Furthermore, our design process in this example was ad hoc; the following two sections will introduce better methods for capturing desired behavior and converting to a circuit.

The above sections went through several intermediate bit storage block designs before arriving at the robust D flip-flop design. Figure 3.35 summarizes those designs, including features and problems of each. Notice that the D flip-flop relies on an internal SR latch to maintain a stored bit *between* clock edges, and relies on the designer to introduce feedback outside the D flip-flop to maintain a stored bit *across* clock edges.



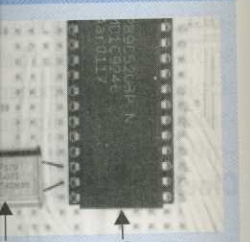
Feature: S=1 sets Q to 1, R=1 resets Q to 0. *Problem:* SR=11 yields undefined Q, other glitches may set/reset inadvertently.

Feature: S and R only have effect when C=1. An external circuit can prevent SR=11 when C=1. *Problem:* avoiding SR=11 can be a burden.

Feature: SR can't be 11. *Problem:* C=1 for too long will propagate new values through too many latches; for too short may not result in the bit being stored.

Feature: Only loads D value present at rising clock edge, so values can't propagate to other flip-flops during same clock cycle. *Tradeoff:* uses more gates internally, and requires more external gates than SR—but transistors today are more plentiful and cheaper.

Figure 3.35 Increasingly better bit storage blocks, leading to the D flip-flop.

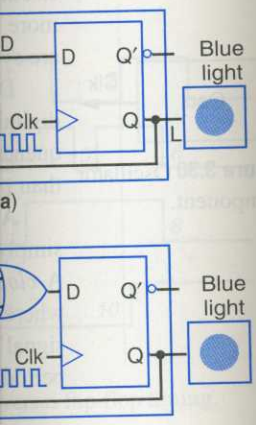


32 Oscillator providing ignal to an IC.

he oscillator can be nput, as in Figure 3.32. cillator.

MHz is short for mega-

ods to frequencies, and hertz, meaning 1 billion or faster) by a factor of 0 ns period equals 100



a)

b)

all-button system: (a) block d using a D flip-flop.

▶ A BIT OF HISTORY—RS, JK, T, AND D LATCHES AND FLIP-FLOPS.

Many textbooks, especially those with origins in the 1970s and 1980s, introduce several types of latches and flip-flops and use many pages to describe how to design sequential circuits using those different types. In the 1980s, transistors on ICs were more costly and scarcer than today. The D flip-flop-based design for the call-button system in Figure 3.34(b) uses more transistors than the SR-latch-based design in Figure 3.10—not only does a D flip-flop contain more transistors internally, but it may require more external logic to set `D` to the appropriate value. Other flip-flop types included a JK flip-flop that acts like an SR flip-flop plus the behavior that the flip-flop toggles if both inputs are `1` (toggle means to change from `0` to `1`, or from `1` to `0`), and a T flip-flop with a single input `T` that toggles the flip-flop when `1`. For a given desired behavior, using a particular flip-flop type could save transistors. Designing sequential circuits for any flip-flop type was a challenging task, involving something called "excitation tables" and comparison of different designs, and was helpful for reducing circuit transistors. But today, in the era of billion-transistor ICs, the savings of such flip-flops are trivial. Nearly all modern sequential circuits use D flip-flops and hence are created using the more straightforward design process introduced in this chapter.

Basic Register—Storing Multiple Bits

A ***register*** is a sequential component that can store multiple bits. A basic register can be built simply by using multiple D flip-flops as shown in Figure 3.36. That register can hold four bits. When the clock rises, all four flip-flops get loaded with inputs `I0`, `I1`, `I2`, and `I3` simultaneously.

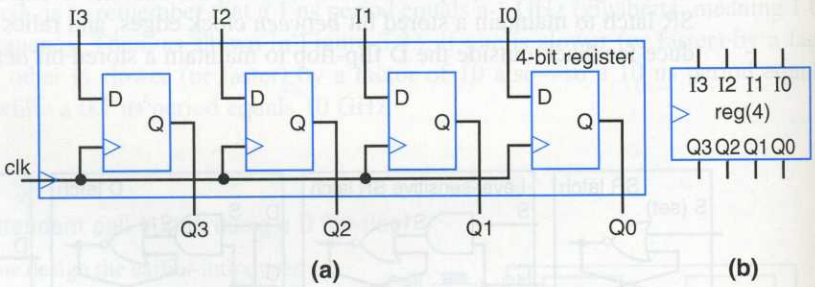


Figure 3.36 A basic 4-bit register: (a) internal design, (b) block symbol.

Such a register, made simply from multiple flip-flops, is the most basic form of a register—so basic that some companies refer to such a register simply as a "4-bit D flip-flop." Chapter 4 introduces more advanced registers having additional features and operations.

Example 3.2 Temperature history display using registers

We want to design a system that records the outside temperature every hour and displays the last three recorded temperatures, so that an observer can see the temperature trend. An architecture of the system is shown in Figure 3.37.

A timer generates a pulse on signal C every hour. A temperature sensor outputs the present temperature as a 5-bit binary number ranging from 0 to 31, corresponding to those temperatures in Celsius. Three displays convert their 5-bit binary inputs into a numerical display.

Figure 3.37 Te
history displa

Figure 3.38
Internal desig
the *Temperatu*
History Storag
component.

Figure 3.39 Exa
values in the
TemperatureHi
Storage registe
particular data i
shown moving t
registers on eac
cycle.

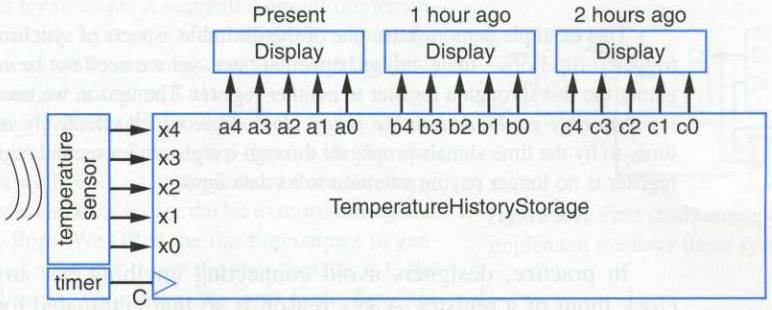


Figure 3.37 Temperature history display system.

(In practice, we would actually avoid connecting the timer output C to a clock input, instead only connecting an oscillator output to a clock input.)

We can implement the *TemperatureHistoryStorage* component using three 5-bit registers, as shown in Figure 3.38. Each pulse on signal `C` loads `Ra` with the present temperature on inputs `x4..x0` (by loading the 5 flip-flops inside `Ra` with the 5 input bits). At the same time that register `Ra` gets loaded with that present temperature, register `Rb` gets loaded with the value that was in `Ra`. Likewise, `Rc` gets loaded with `Rb`'s value. All three loads happen at the same time, namely, on the rising edge of `C`. The effect is that the values that were in `Ra` and `Rb` just before the clock edge are shifted into `Rb` and `Rc`, respectively.

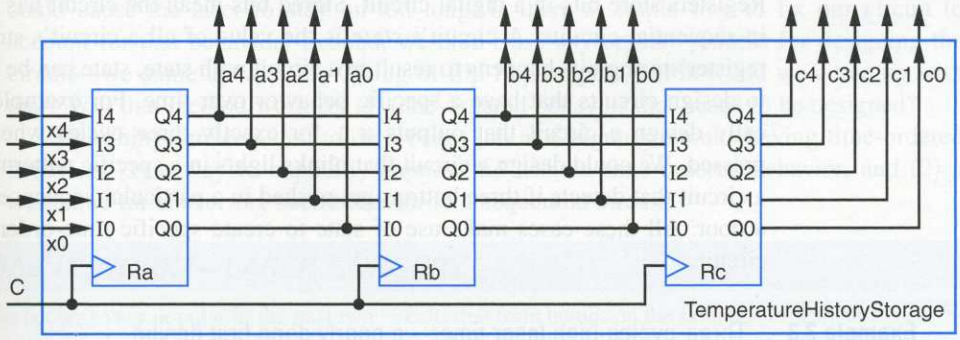


Figure 3.38 Internal design of the *Temperature History Storage* component.

Figure 3.39 shows sample values in the registers for several clock cycles, assuming all the registers initially held `0`s, and assuming that as time proceeds the inputs `x4..x0` have the values shown at the top of the timing diagram.

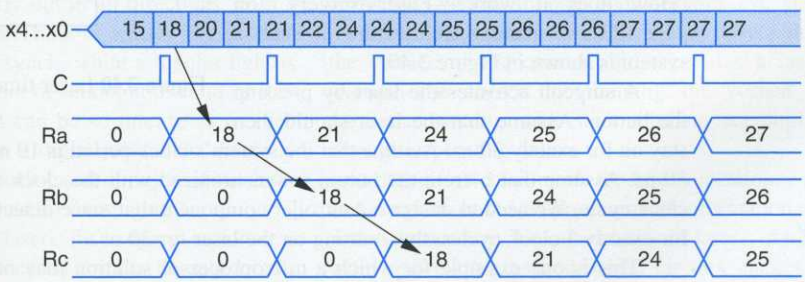


Figure 3.39 Example of values in the *TemperatureHistory Storage* registers. One particular data item, 18, is shown moving through the registers on each clock cycle.

This example demonstrates one of the desirable aspects of synchronous circuits built from edge-triggered flip-flops—many things happen at once, yet we need not be concerned about signals propagating too fast through a register to another register. The reason we need not be concerned is because registers *only get loaded on the rising clock edge*, which effectively is an infinitely small period of time, so by the time signals propagate through a register to a second register, it's too late—that second register is no longer paying attention to its data inputs.

In practice, designers avoid connecting anything but an oscillator's output to the clock input of a register. A key reason is so that automated tools that analyze a circuit's timing characteristics can work properly; such tools are beyond the scope of this book. We connected a timer's output, which pulsed once per hour, in the above example for the purpose of an intuitive introduction to registers. A better implementation would instead have an oscillator connected to the clock input, and then use the "load" input of a register when the timer output pulsed. The load input of a register will be introduced in Chapter 4.

▶ 3.3 FINITE-STATE MACHINES (FSMS)

Registers store bits in a digital circuit. Stored bits mean the circuit has ***memory*** resulting in sequential circuits. A circuit's ***state*** is the value of all a circuit's stored bits. While a register storing bits happens to result in a circuit with state, state can be intentionally used to design circuits that have a specific behavior over time. For example, we can specifically design a circuit that outputs a `1` for exactly three cycles whenever a button is pressed. We could design a circuit that blinks lights in a specific pattern. We could design a circuit that detects if three buttons get pushed in a particular sequence and then unlocks a door. All these cases make use of state to create specific time-ordered behavior for a circuit.

Example 3.3 Three-cycles-high laser timer—a poorly done first design

Consider the design of a part of a laser surgery system, such as a system for scar removal or corrective vision. Such systems work by turning on a laser for a precise amount of time (see "How does it work?—Laser surgery" on page 123). A general architecture of such a system is shown in Figure 3.40.

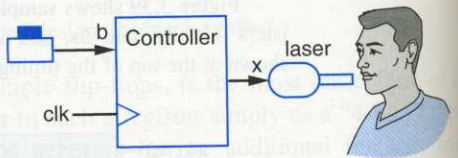


Figure 3.40 Laser timer system.

A surgeon activates the laser by pressing the button. Assume that the laser should then stay on for exactly 30 ns. Assume that the system's clock period is 10 ns, so that 3 clock cycles last 30 ns. Assume that `b` from the button is synchronized with the clock and stays high for exactly 1 clock cycle. We need to design a controller component that, once detecting that `b = 1`, holds `x` high for exactly 3 clock cycles, thus turning on the laser for 30 ns.

This is one example for which a microprocessor solution may not work. Using a microprocessor's programming statements that read input ports and write output ports may not provide a way to hold an output port high for exactly 30 ns—for example, when the microprocessor clock frequency is not fast enough.

Let's try to create a sequential circuit implementation for the system. After thinking about the problem for a while, we might come up with the (bad) implementation in Figure 3.41.

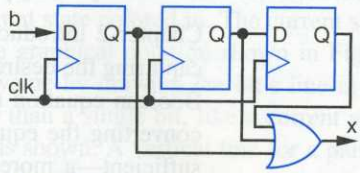


Figure 3.41 First (bad) attempt to implement the laser timer system.

Knowing the output should be held high for three clock cycles, we used three flip-flops, with the idea being that we'll shift a `1` through those three flip-flops, taking three clock cycles for the bit to move through all three flip-flops. We ORed the flip-flop outputs to generate signal `x`, so that if any flip-flop contains a `1`, the laser will be on. We made `b` the input to the first flip-flop, so when `b=1`, the first flip-flop stores a `1` on the next rising clock edge. One clock cycle later, the second flip-flop will get loaded with `1`, and assuming `b` has now returned to `0`, the first flip-flop will get loaded with `0`. One clock cycle later, the third flip-flop will get loaded with `1`, and the second flip-flop with `0`. One clock cycle later, the third flip-flop will get loaded with `0`. Thus, the circuit held the output `x` at `1` for three clock cycles after the button was pressed.

We did a poor job implementing this system. First, what happens if the surgeon presses the button a second time before the three cycles are completed? Such a situation could cause the laser to stay on too long. Is there a simple way to fix our circuit to account for that behavior? Second, we didn't use any orderly process for designing the circuit—we came up with the ORing of flip-flop outputs, but how did we come up with that? Will that method work for all time-ordered behavior that needs to be designed?

Two things are required to do a better job at designing circuits having time-ordered behavior: (1) a way to explicitly *capture* the desired time-ordered behavior, and (2) a technique for *converting* such behavior to a sequential circuit.

▶ HOW DOES IT WORK?—LASER SURGERY.

Laser surgery has become very popular in the past two decades, and has been enabled due to digital systems. Lasers, invented in the early 1960s, generate an intense narrow beam of coherent light, with photons having a single wavelength and being in phase (like being in rhythm) with one another. In contrast, a regular light's photons fly out in all directions, with a diversity of wavelengths. Think of a laser as a platoon of soldiers marching in synch, while a regular light is more like kids running out of school at the end-of-the-day bell. A laser's light can be so intense as to even cut steel. The ability of a digital circuit to carefully control the location, intensity, and duration of the laser is what makes lasers so useful for surgery.

One popular use of lasers for surgery is for scar removal. The laser is focused on the damaged cells slightly below the surface, causing those cells to be vaporized. The laser can also be used to vaporize skin cells that form bumps on the skin, due to scars or moles. Similarly, lasers can reduce wrinkles by smoothing the skin around the wrinkle to make the crevices more gradual and hence less obvious, or by stimulating tissue under the skin to stimulate new collagen growth.

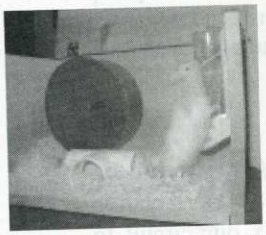
Another popular use of lasers for surgery is for correcting vision. In one popular laser eye surgery method, the surgeon uses a laser to cut open a flap on the surface of the cornea, and then uses a laser to reshape the cornea by thinning the cornea in a particular pattern, with such thinning accomplished through vaporizing cells.

A digital system controls the laser's location, energy, and duration, based on programmed information of the desired procedure. The availability of lasers, combined with low-cost high-speed digital circuits, makes such precise and useful surgery now possible.

Mathematical Formalism for Sequential Behavior—FSMs

Chapter 2 introduced a process for designing a combinational circuit that involved first capturing the desired combinational behavior using a mathematical formalism known as a Boolean equation (or a truth table, which could be converted to an equation), and then converting the equation to a circuit. For sequential behavior, a Boolean equation is not sufficient—a more powerful mathematical formalism is needed that can describe time-ordered behavior.

Finite-state machines (FSMs) are one such formalism. The name is awkward, but the concept is straightforward. The main part of an FSM is a set of states representing every possible system "mode" or "situation." An FSM is "in" exactly one state at any time, that state being known as the FSM's current or present state.



The "TryingToEscape" state.

My daughter's hamster can serve as an intuitive example. After having a hamster as a family pet, I've learned that hamsters basically have four states that can be named *Sleeping*, *Eating*, *RunningOnTheWheel*, and *TryingToEscape*. Hamsters spend most of their day sleeping (being nocturnal), a bit of time eating or running on the wheel, and the rest of their time desperately trying to escape from their cage. At any given time, the hamster is in exactly one of those four states.

A digital example is a system that repeatedly sets an output `x` to `0` for one clock cycle and to `1` for one clock cycle, so the output over time will be `0 1 0 1 0 1` ... The system clearly has only two states, which might be named *Lo* and *Hi*. In state *Lo*, `x = 0`; in state *Hi*, `x = 1`. Those states and the transitions between them can be drawn as the state diagram in Figure 3.42(a). A ***state diagram*** is a graphical drawing of an FSM.

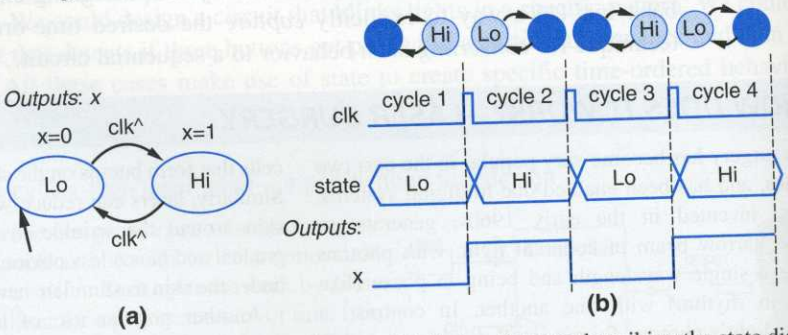


Figure 3.42 A two-state FSM: (a) state diagram, (b) timing diagram describing the state diagram's behavior. Above the timing diagram, an animation of the state diagram highlights the current state in each clock cycle. "clk^" represents the rising edge of the clock signal.

Each state in the state diagram is drawn as an oval. Assume the system starts in state *Lo*, as indicated by the arrow pointing from nothing to state *Lo*. The diagram shows that state *Lo* sets `x` to `0` by having "`x=0`" drawn near or in the state. The diagram also shows that on the next rising edge of the clock signal, *clk^*, the system transitions to state *Hi*. Such transitions are drawn as directed edges, meaning a line with an arrow at one end. The diagram also shows that state *Hi* sets `x` to `1`. On the next rising edge of the clock, the diagram shows that the system transitions back to state *Lo* again. State diagrams are a popular method for representing FSMs. FSMs can also be represented as a table or using various textual methods.

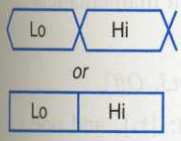


Figure 3.43 Displaying multi-bit or other values in a timing diagram.

Figure 3.42(b) provides a timing diagram showing the system's behavior. Above the timing diagram are state diagrams that show the current state colored in. The current state is also shown in the timing diagram itself using the graphical notation shown in Figure 3.43. A timing diagram easily shows a single bit's value by drawing the bit's line at the top or the bottom. But to represent something other than a single bit, like a current state or an integer value, the notation just lists the value as shown. A vertical line (or a pair of crossed lines) shows when the values change.

Note that an FSM only moves along a single transition for a single rising clock edge. In particular, when in state *Lo*, a rising clock edge causes a move along the transition to state *Hi*, but then the system must wait for *another* rising clock edge to move along the transition from *Hi* back to *Lo*.

Example 3.3 sought to build a system that held its output high for three cycles. Toward that end, the state diagram of Figure 3.42 can be extended to have four states, three of which set the output to `1`, as in Figure 3.44(a). The output `x` will be `0` for one cycle and then `1` for three cycles, as shown in the timing diagram of Figure 3.44(b). The state diagram uses state names *Off*, *On1*, *On2*, and *On3*. State names are arbitrary; the only requirement is that an FSM's state names must each be unique. The state names in Figure 3.44 could have been *S0*, *S1*, *S2*, and *S3*, but names that describe a state's purpose in the FSM are preferable.

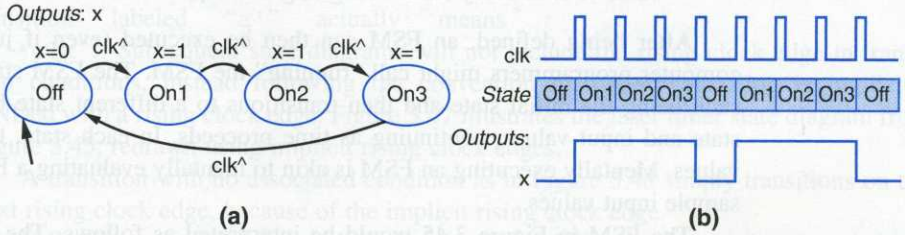


Figure 3.44 Three-cycles-high system: (a) state diagram, (b) timing diagram.

Boolean expressions can be associated with the transitions to extend the behavior. Figure 3.45(a) extends the state diagram by associating an expression with the transition from state *Off* to state *On1* such that the expression requires not just a rising clock, but also that `b=1` (written just as `b`) in order for the transition to be taken. Another transition can be added from *Off* back to *Off*, with the expression of a rising clock and `b=0` (written as `b'`). The timing diagram in Figure 3.45(b) shows the state and output behavior for the given input values on `b`. The initial state is *Off*. While `b` is `0`, the FSM stays in *Off* (it keeps transitioning back to *Off* at each rising clock). When `b` becomes `1`, the FSM transitions to *On1* at the next rising clock, then to *On2*, then *On3*, then back to *Off*.

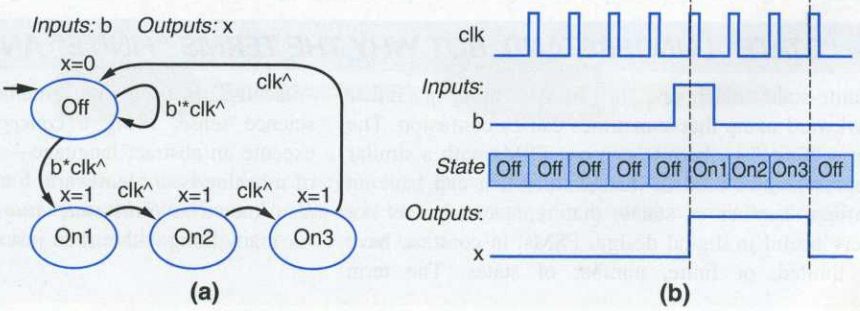


Figure 3.45 Three-cycles-high system with button input: (a) state diagram, (b) timing diagram.

The above examples illustrate that a ***finite-state machine*** or ***FSM*** is a mathematical formalism consisting of several items:

- A set of states. The above example had four states: {*On1, On2, On3, Off*}.
- A set of inputs and a set of outputs. The example had one input: {`b`}, and one output: {`x`}.
- An ***initial state***: the state in which to start when the system is first powered on. An FSM's initial state can be shown graphically by a directed edge (an edge with an arrow at one end) starting from no state and pointing to the initial state. An FSM can only have one initial state. The example's initial state was the state named *Off*. Note that *Off* is just a name, and does not suggest that the system's power is off (rather, it suggests that the laser is off).
- A set of transitions: An indication of the next state based on the current state and the current values of the inputs. The example used directed edges with associated input ***conditions***, which is a Boolean expression of input variables, to indicate the next state. Those edges with conditions are called ***transitions***. The example had several transitions, such as the edge with condition `b*clk^`.
- A description of what output values to assign in each state. The example assigns a value to `x` in every state. Assigning an output in an FSM is known as an ***action***.

After being defined, an FSM can then be executed (even if just mentally)—what computer programmers might call "running" the FSM. The FSM starts with the current state being the initial state and then transitions to a different state based on the current state and input values, continuing as time proceeds. In each state, the FSM sets output values. Mentally executing an FSM is akin to mentally evaluating a Boolean equation for sample input values.

The FSM in Figure 3.45 would be interpreted as follows. The system starts in the initial state *Off*. The system stays in state *Off* until one of the state's two outgoing transitions has a true condition. One of those transitions has the condition of `b'*clk^`—in that case, the system transitions right back to state *Off*. The other transition has the condition of `b*clk^`—in that case, the system transitions to state *On1*. The system stays in state *On1* until its only outgoing transition's condition `clk^` becomes true—in which case the system transitions to state *On2*. Likewise, the system stays in *On2* until the next rising clock edge, transitioning to *On3*. The system stays in *On3* until the next rising clock edge, transitioning back to state *Off*. State *Off* has associated the action of setting `x=0`, while the states *On1*, *On2*, and *On3* each set `x=1`.

▶ "STATE" I UNDERSTAND, BUT WHY THE TERMS "FINITE" AND "MACHINE?"

Finite-state machines, or FSMs, have a rather awkward name that sometimes causes confusion. The term "finite" is there to contrast FSMs with a similar representation used in mathematics that can have an infinite number of states; that representation is not very useful in digital design. FSMs, in contrast, have a limited, or finite, number of states. The term "machine" is used in its mathematical or computer science sense, being a *conceptual* object that can execute an abstract language—specifically, that sense of machine is *not* hardware. Finite-state machines are also known as ***finite-state automata***. FSMs are used for many things other than just digital design.

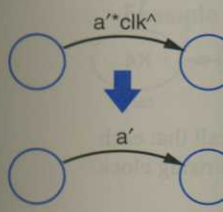


Figure 3.46 Simplifyin notation: implicit risin clock edge on every transition.

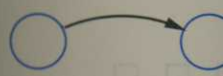


Figure 3.48 Transition is taken on next rising clock edge.

Example 3

The FSM in Figure 3.45 precisely describes the desired time-ordered behavior of the laser timer system from Example 3.3.

It is interesting to examine the behavior of this FSM if the button is pressed a second time while the laser is on. Notice that the transitions among the *On* states are independent of the value of *b*. So this system will always turn the laser on for exactly three cycles and then return to the *Off* state to await another press of the button.

Simplifying FSM Notation: Making the Rising Clock Implicit

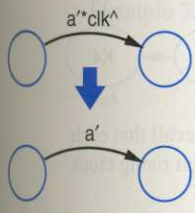


Figure 3.46 Simplifying notation: implicit rising clock edge on every transition.

Thus far the rising clock edge (`clk^`) has appeared in the condition of every FSM transition, because this book only considers the design of sequential circuits that are synchronous and that use rising edge-triggered flip-flops to store bits. Synchronous circuits with edge-triggered flip-flops make up the majority of sequential circuits in modern practice. As such, to make state diagrams more readable, most textbooks and designers follow the convention shown in Figure 3.46 wherein every FSM transition is *implicitly ANDed* with a rising clock edge. For example, a transition labeled "`a'`" actually means "`a'*clk^`." Subsequent state diagrams will not include the rising clock edge in transition conditions, instead following the convention that *every* transition is implicitly ANDed with a rising clock edge. Figure 3.47 illustrates the laser timer state diagram from Figure 3.45, redrawn using implicit rising clock edges.

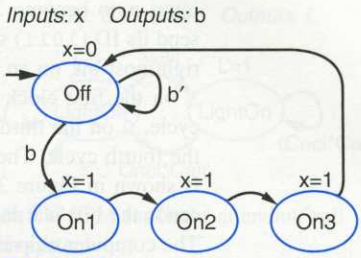


Figure 3.47 Laser timer state diagram assuming every transition is ANDed with a rising clock.

Figure 3.48 Transition is taken on next rising clock edge.

A transition with no associated condition as in Figure 3.48 simply transitions on the next rising clock edge, because of the implicit rising clock edge.

Following are more examples showing how FSMs can describe time-ordered behavior.

Example 3.4 Secure car key

Have you noticed that the keys for many new automobiles have a thicker plastic head than in the past (see Figure 3.49)? The reason is that, believe it or not, there is a computer chip inside the head of the key, implementing a secure car key. In a basic version of such a secure car key, when the driver turns the key in the ignition, the car's computer (which is under the hood and communicates using what's called the *basestation*) sends out a radio signal asking the car key's chip to respond by sending an identifier via a radio signal. The chip in the key then responds by sending the identifier

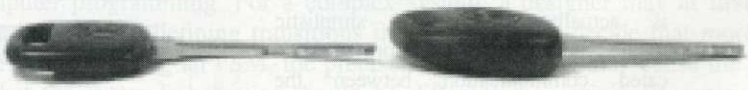


Figure 3.49 Why are the heads of car keys getting thicker? Note that the key on the right is thicker than the key on the left. The key on the right has a computer chip inside that sends an identifier to the car's computer, thus helping to reduce car thefts.

(ID), using what's known as a *transponder* (a transponder "transmits" in "response" to a request). If the basestation does not receive a response or the key's response has an ID different than the ID programmed into the car's computer, the computer shuts down and the car won't start.

Let's design the controller for such a key, having an ID of `1011` (real IDs are typically 32 bits long or more, not just 4 bits). Assume the controller has an input `a` that is `1` when the car's computer requests the key's ID. Thus the controller initially waits for the input `a` to become `1`. The key should then send its ID (`1011`) serially, starting with the rightmost bit, on an output `r`; the key sends `1` on the first clock cycle, `1` on the second cycle, `0` on the third cycle, and finally `1` on the fourth cycle. The FSM for the controller is shown in Figure 3.50. Note that the FSM sends the bits starting from the bit on the right, which is known as the *least significant bit* (LSB). The computer chip in the car key has circuitry that converts radio signals to bits and vice versa.

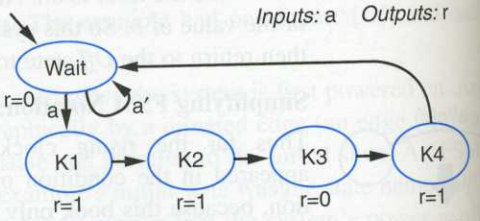


Figure 3.50 Secure car key FSM. Recall that each edge's condition includes an implicit rising clock edge.

Figure 3.51 provides a timing diagram for the FSM for a particular situation. When we set `a = 1`, the FSM enters state *K1* and outputs `r = 1`. The FSM then proceeds through *K2*, *K3*, and *K4*, outputting `r = 1`, `0`, and `1`, respectively, even though we returned input `a` to `0`.

Timing diagrams represent a particular situation defined by how we set the inputs. What would have happened if we had held `a = 1` for many more clock cycles? The timing diagram in Figure 3.52 illustrates that situation. Notice how in that case the FSM, after returning to state *Wait*, proceeds to state *K1* again on the next cycle.

"So my car key may someday need its batteries replaced?" you might ask. Actually, no—those chips in keys draw their power, as well as their clock, from the magnetic component of the radio-frequency field generated from the computer basestation, as in RFID chips. The extremely low power requirement makes custom digital circuitry, rather than instructions on a microprocessor, a preferred implementation.

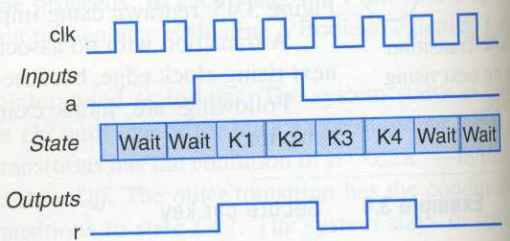


Figure 3.51 Secure car key timing diagram.

Computer chip keys make stealing cars a lot harder—no more "hot-wiring" to start a car, since the car's computer won't work unless it also receives the correct identifier. And the method above is actually an overly simplistic method—many cars have more sophisticated communication between the computer and the key, involving several communications in both directions, even using encrypted communication—making fooling the car's computer even harder. A drawback of secure car keys is that you

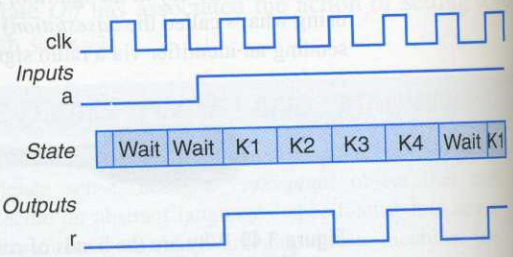


Figure 3.52 Secure car key timing diagram for a different sequence of values on input `a`.

can't just run down to the local hardware store and copy those keys for $5 any longer—copying keys requires special tools that today can run $50-$100. A common problem while computer chip keys were becoming popular was that low-cost locksmiths didn't realize the keys had chips in them, so copies were made and the car owners went home and later couldn't figure out why their car wouldn't start, even though the key fit in the ignition slot and turned.

Example 3.5 Flight-attendant call button

This example uses an FSM to describe the desired behavior of the flight-attendant call button system from Figure 3.1. The FSM has inputs `Call` and `Cncl` for the call and cancel buttons, and output `L` to control the light. `Call` will be given priority if both buttons are pressed. The FSM has two states, *LightOff*, which sets `L` to `0`, and *LightOn*, which sets `L` to `1`, as shown in Figure 3.53. *LightOff* is the initial state. The FSM stays in that state until `Call` is `1`, which causes a transition to *LightOn*. If `Call` is `0`, the FSM stays in *LightOff*. In state *LightOn*, the only way to transition back to *LightOff* is if `Cncl` is `1` and `Call` is `0` (because the call button has priority), meaning `Cncl*Call'`. If that condition is false, i.e., `(Cncl*Call')'` is true, the FSM stays in *LightOn*.

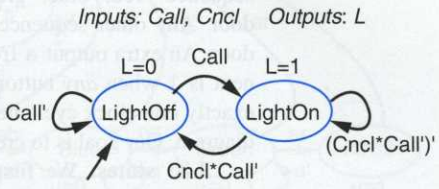


Figure 3.53 FSM for flight-attendant call button system.

Notice how clearly the FSM captures the behavior of the flight-attendant call button system. Once you understand FSMs, an FSM description is likely to be more concise and precise than an English description.

How to Capture Desired System Behavior as an FSM

The previous section showed FSM examples, but how were those FSMs originally created? Creating an FSM that captures desired system behavior can be a challenging task for a designer. Using the following method can help:

- List states: First list all possible states of the system, giving each a meaningful name, and denoting the initial state. Optionally add some transitions if they help indicate the purpose of each state.
- Create transitions: For each state, define all possible transitions leaving that state.
- Refine the FSM: Execute the FSM mentally and make any needed improvements.

The method described above is just a guide. Capturing behavior as an FSM may require some creativity and trial-and-error, as is the case in some other engineering tasks, like computer programming. For a complex system, a designer may at first list a few states, and then upon defining transitions the designer may decide that more states are required. While creating an FSM, the preciseness of the FSM may cause the designer to realize that the system's behavior should be different than originally anticipated. Note also that many different FSMs could be created that describe the same desired behavior; one FSM may be easier to understand while another FSM may have fewer states, for example. Experience can help greatly in creating correct and easy-to-understand FSMs that capture desired system behavior.

Example 3.6 Code detector

You've probably seen keypad-protected doors in airports or hospitals that require a person to press a sequence of buttons (i.e., a code) to unlock the door. A door may have three buttons, colored red, green, and blue, and a fourth button for starting the code. Pressing the start button followed by the sequence—red, blue, green, red—unlocks the door. Any other sequence would not unlock the door. An extra output `a` from the buttons component is `1` when *any* button is pressed; `a` is `1` for exactly one clock cycle no matter how long a button is pressed. Figure 3.54 shows a system block diagram. Our goal is to create an FSM that describes the *CodeDetector* block.

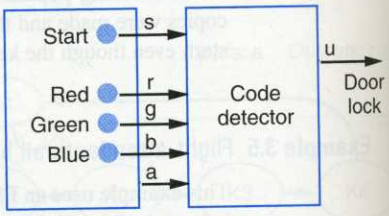


Figure 3.54 Code detector architecture.

List states: We first list the possible states of the code detector FSM, shown in Figure 3.55. A state is needed to wait for the start button to be pressed; we name that state *Wait*, and add transitions showing that the FSM stays in that state while `s` is `0`. After the start button is pressed, a state is needed to wait for the first button to be pressed; we name that state *Start*. While in *Start*, if a button is pressed (`a=1`) and it is the red button (`r=1`), then the FSM should enter a state that indicates that the first colored button pressed was the correct one; we name that state *Red1*, and add a transition from *Start* to *Red1* with condition `ar` (for `a=1` AND `r=1`) to make clear that *Red1* is entered if the red button is pressed. While in *Red1*, if the blue button is pressed, the FSM should enter another state indicating that the second colored button pressed was correct; we name that state *Blue*, and add a transition from *Red1* to *Blue* also. Likewise, we add states *Green* and *Red2* for the last two of the four required button presses. If the FSM reaches the *Red2* state, then all four pressed buttons were the correct ones, and thus the door should be unlocked. Hence, state *Red2* has associated the action of `u=1`, while all the other states set `u=0`. At this point, the FSM is incomplete, but lists all the states and a few key transitions to capture the main behavior of detecting the correct sequence of pressed colored buttons.

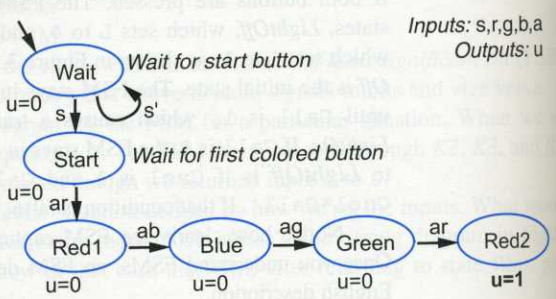


Figure 3.55 Code detector's possible states.

Create transitions: The next step is to create transitions for each state. State *Wait* already has a complete set of transitions: when `s` is `0`, the FSM stays in *Wait*; when `s` is `1`, the FSM goes to *Start*. State *Start*'s transitions should include waiting for a colored button to be pressed, so we add a transition with condition `a'` pointing back to *Start*, shown in Figure 3.56. If a button is pressed and that button is the red button, then

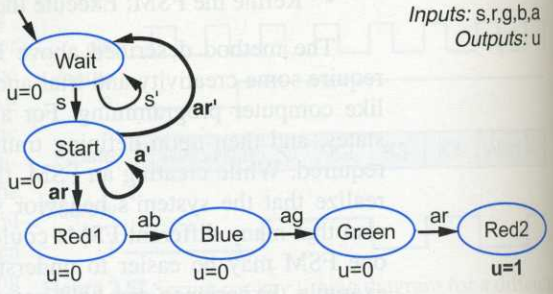


Figure 3.56 Code detector FSM with more transitions.

the FSM should go to state *Red1*; we'd already added that transition (`ar`). If a button is pressed and that button is not the red button (`ar'`), then the FSM should somehow enter a "fail" mode and not unlock the door. At this point, we might consider adding another state called *Fail*. Instead, we decide that the FSM should go back to the *Wait* state and just wait for the start button to be pressed again, so we add such a transition with condition `ar'` as shown.

The pattern of three transitions for state *Start* can be replicated for states *Red1*, *Blue*, and *Green*, modified to detect the correct colored button press as shown in Figure 3.57. Finally, we must decide what the FSM should do after the FSM reaches state *Red2* and unlocks the door. For simplicity of this example, we decide to have the FSM just return to state *Wait*, which locks the door again; a real system would keep the door unlocked for a fixed period of time before locking it again.

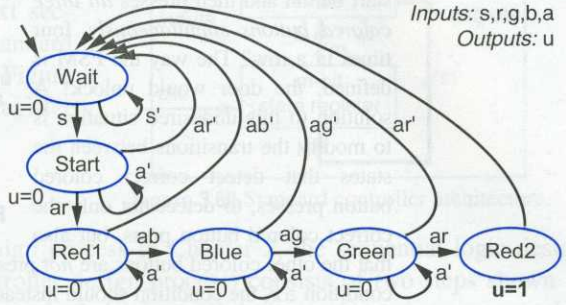


Figure 3.57 Code detector FSM with complete transitions.

Refine the FSM: We can now mentally execute the FSM to see if it behaves as desired:

- The FSM begins in the *Wait* state. As long as the start button is not pressed (`s'`), the FSM stays in *Wait*; when the start button `s` is pressed (and a rising clock edge arrives, of course), the FSM goes to the *Start* state.
- Being in the *Start* state means the FSM is now ready to detect the sequence red, blue, green, red. If no button is pressed (`a'`), the FSM stays in *Start*. If a button is pressed AND that button is the red button (`ar`), the FSM goes to state *Red1*. Instead, if a button is pressed AND that button is not the red button (`ar'`), the FSM returns to the `Wait` state—note that when in the *Wait* state, further presses of the colored buttons would be ignored, until the start button is pressed again.
- The FSM stays in state *Red1* as long as no button is pressed (`a'`). If a button is pressed AND that button is blue (`ab`), the FSM goes to state *Blue*; if that button is not blue (`ab'`), the FSM returns to state *Wait*. At this point, we detect a potential problem—what if the red button is still being pressed as part of the first button press when the next rising clock edge arrives? The FSM would go to state *Wait*, which is not what we want. One solution is to add another state, *Red1_Release*, that the FSM transitions to after *Red1*, and in which the FSM stays until `a=0`. For simplicity, we'll instead assume that each button has a special circuit that synchronizes the button with the clock signal. That circuit sets its output to `1` for exactly one clock cycle for each unique press of the button. This is necessary to ensure that the current state doesn't inadvertently change to another state if a button press lasts longer than a single clock cycle. We'll design such a synchronization circuit in Example 3.9.
- Likewise, the FSM stays in state *Blue* as long as no button is pressed (`a'`), and goes to state *Green* on condition `ag`, and state *Wait* on condition `ag'`.
- Finally, the FSM stays in *Green* if no button is pressed, and goes to state *Red2* on condition `ar`, and to state *Wait* on condition `ar'`.
- If the FSM makes it to state *Red2*, that means that the user pressed the buttons in the correct sequence—*Red2* will set `u=1`, thus unlocking the door. Note that all other states set `u=0`. The FSM then returns to state *Wait*.

The FSM works well for normal button presses, but let's mentally execute the FSM for unusual cases. What happens if the user presses the start button and then presses *all three colored buttons simultaneously*, four times in a row? The way the FSM is defined, the door would unlock! A solution to this undesired situation is to modify the transitions between the states that detect correct colored button presses, to detect not only the correct colored button press, but also that the other colored buttons are *not* pressed. For example, for the transition leaving state *Start* with condition `ar`, the condition should instead be `a(rb'g')`. That change also means that the transition going back to state *Wait* should have the condition `a(rb'g')'`. The intuitive meaning of that condition is that a button was pressed, but it was not just the red button. Similar changes can be made to the other transition conditions too, resulting in the improved FSM of Figure 3.58.

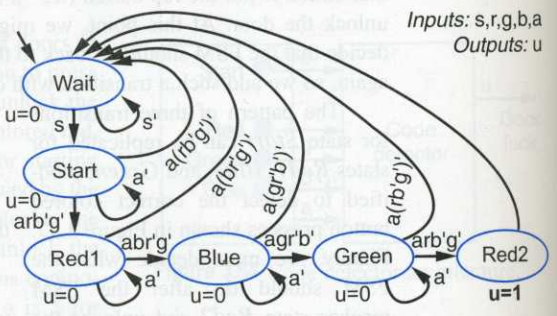


Figure 3.58 Improved code detector FSM.

▶ 3.4 CONTROLLER DESIGN

Standard Controller Architecture for Implementing an FSM as a Sequential Circuit

The previous section provided examples of capturing sequential behavior using FSMs. This section defines a process to convert an FSM to a sequential circuit. The sequential circuit that implements an FSM is commonly called a ***controller***. Converting an FSM to a controller is quite straightforward when a standard pattern, commonly called a standard architecture, is used for the controller. Other ways exist for implementing an FSM, but using the standard architecture results in a straightforward design process.

A standard controller architecture for an FSM consists of a register and combinational logic. For example, the standard controller architecture for the laser timer FSM of Figure 3.45 is shown in Figure 3.59. The controller's register stores the current FSM state and is thus called a ***state register***. Each state is represented as a unique bit encoding. For example, the laser timer's *Off* state could be encoded as `00`, *On1* as `01`, *On2* as `10`, and *On3* as `11`, the four states thus requiring a 2-bit state register.

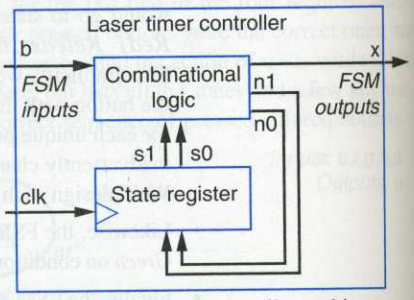


Figure 3.59 Standard controller architecture for the laser timer.

The combinational logic computes the output values for the present state, and also computes the next state based on the current state and current input values. Its inputs are thus the state register bits (`s1` and `s0` in the example of Figure 3.59) and the FSM's external inputs (`b` for the example). The combinational logic's outputs are the outputs of

the FSM (x for the example), as well as the next state bits to be loaded into the state register (n1 and n0).

The details of the combinational logic determine the behavior of the circuit. The process for creating those details will be covered in the next section. A more general view of the standard controller architecture appears in Figure 3.60. That figure shows a state register that is *m* bits wide.

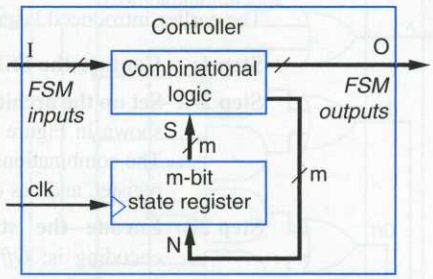


Figure 3.60 Standard controller architecture.

Controller (Sequential Logic) Design Process

As in the combinational logic design process in Chapter 2, the sequential logic design process (which we'll call the controller design process) consists of two steps shown in Table 3.2. The first step is to ***capture*** the behavior, and the second step is to ***convert*** that captured behavior into a circuit. Combinational logic was captured as equations or truth tables, but those formalisms are insufficient for the time-ordered behavior of a controller, so capturing a controller's behavior is done with an FSM. The second step of converting the captured behavior to a circuit requires several substeps as shown in Table 3.2. Each substep is a straightforward task—while the design process' first step of capturing the behavior as an FSM may require some trial-and-error, the second step of converting the FSM to a circuit is a straightforward "mechanical" activity even though it consists of several substeps. Examples will introduce and illustrate the controller design process.

TABLE 3.2 Controller design process.

Step		Description
Step 1: Capture behavior	***Capture*** *the FSM*	Create an FSM that describes the desired behavior of the controller.
Step 2: Convert to circuit	2A ***Set up*** *the architecture*	Set up the standard architecture by using a state register of appropriate width and combinational logic. The logic's inputs are the state register bits and the FSM inputs; the logic's outputs are the next state bits and the FSM outputs.
	2B ***Encode*** *the states*	Assign a unique binary number, known as an ***encoding,*** to each state. Any encoding is sufficient as long as each state has a unique encoding. Usually a minimum number of bits is used and an encoding is assigned to each state by counting up in binary.
	2C ***Fill in*** *the truth table*	Translate the FSM into a truth table for the combinational logic such that the logic will generate the outputs and next state signals for the given FSM. Ordering the inputs with state bits first makes the correspondence between the table and the FSM clear.
	2D ***Implement*** *the combinational logic*	Implement the combinational logic using any method.

Example 3.7 Three-cycles-high laser timer controller (continued)

The earlier-introduced laser timer can be implemented using the controller design process.

Step 1: Capture the FSM. The FSM was already created in Figure 3.47.

Step 2A: Set up the architecture. The standard controller architecture for the laser timer FSM was shown in Figure 3.59. The state register has two bits to represent each of the four states. The combinational logic has external input `b` and inputs `s1` and `s0` coming from the state register, and has external output `x` and outputs `n1` and `n0` going to the state register.

Step 2B: Encode the states. A valid state encoding is: *Off*: `00`, *On1*: `01`, *On2*: `10`, *On3*: `11`. Any non-redundant encoding is fine. The state diagram with encoded states is in Figure 3.61.

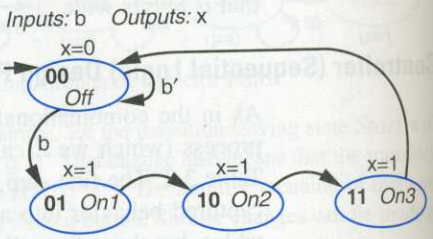


Figure 3.61 Laser timer state diagram with encoded states.

Step 2C: Fill in the truth table. Given the implementation architecture and the binary encoding of each state, the FSM can be translated into the truth table for the combinational logic, as shown in Table 3.3. Placing the inputs coming from the state register in the table's leftmost input columns allows us to easily see which rows correspond to which states. We fill all combinations of inputs on the left, as usual for a truth table. For each row, we look at the state diagram in Figure 3.61 to determine the appropriate outputs. For the two rows starting with `s1s0 = 00` (state *Off*), `x` should be `0`. If `b = 0`, the controller should stay in state *Off*, so `n1n0` should be `00`. If `b = 1`, the controller should go to state *On1*, so `n1n0` should be `01`.

Likewise, for the two rows starting with `s1s0 = 01` (state *On1*), `x` should be `1` and the next state should be *On2* (regardless of `b`'s value), so `n1n0` should be `10`. We complete the last four rows similarly.

Note the difference between the FSM inputs and outputs of Figure 3.61, and the combinational logic inputs and outputs of Figure 3.62—the latter include the bits from and to the state register.

TABLE 3.3 Truth table for laser timer controller's combinational logic.

	Inputs			Outputs		
	s1	s0	b	x	n1	n0
Off	0	0	0	0	0	0
	0	0	1	0	0	1
On1	0	1	0	1	1	0
	0	1	1	1	1	0
On2	1	0	0	1	1	1
	1	0	1	1	1	1
On3	1	1	0	1	0	0
	1	1	1	1	0	0

Step 2D: Implement the combinational logic. The design can be completed using the combinational logic design process from Chapter 2. The following equations for the three combinational outputs come from the truth table:

```
x = s1 + s0  (note from the table
  that x = 1 if s1 = 1 or s0 = 1)
n1 = s1's0b' + s1's0b +
  s1s0'b' + s1s0'b
n1 = s1's0 + s1s0'
n0 = s1's0'b + s1s0'b' + s1s0'b
n0 = s1's0'b + s1s0'
```

We then obtain the sequential circuit in Figure 3.62, implementing the FSM.

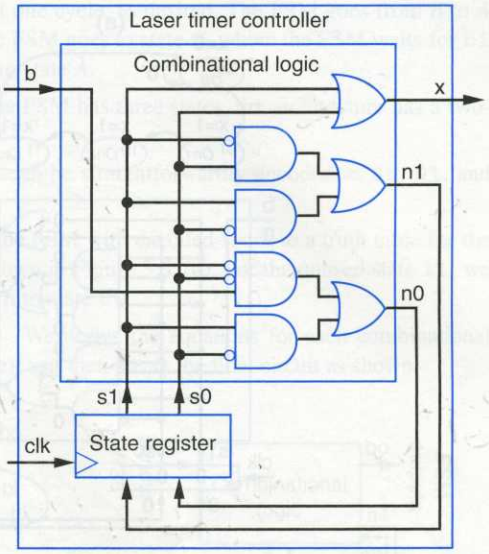


Figure 3.62 Final implementation of the three-cycles-high laser timer controller.

Many textbooks use different table organizations from that in Table 3.3. However, we intentionally organized the table so that it serves both as a *state table*, which is a tabular representation of an FSM, and as a truth table that can be used to design the combinational logic of the controller.

Example 3.8 Understanding the laser timer controller's behavior

To aid in understanding how a controller implements an FSM, this example traces through the behavior of the three-cycles-high laser timer controller. Assume the system is initially in state `00` (`s1s0=00`), `b` is `0`, and the clock is currently low. As shown in Figure 3.63(a), based on the combinational logic, `x` will be `0` (the desired output in state `00`), `n1` will be `0`, and `n0` will be `0`, meaning the value `00` will be waiting at the state register's inputs. Thus, on the *next* clock edge, `00` will be loaded into the state register, meaning the system stays in state `00`—which is correct.

Now suppose `b` becomes `1`. As shown in Figure 3.63(b), `x` will still be `0`, as desired. `n1` will be `0`, but `n0` will be `1`, meaning the value `01` will be waiting at the state register's inputs. Thus, on the *next* clock edge, `01` will be loaded into the state register, as desired.

As in Figure 3.63(c), soon after `01` is loaded into the state register, `x` will become `1` (after the register is loaded, there's a slight delay as the new values for `s1` and `s0` propagate through the combinational logic gates). That output is correct—the system should output `x=1` when in state `01`. Also, `n1` will become `v` and `n0` will equal `0`, meaning the value `10` will be waiting at the state register inputs. Thus, on the next clock edge, `10` will be loaded into the state register, as desired.

After `10` is loaded into the state register, `x` will stay `1`, and `n1n0` becomes `11`. When another clock edge comes, `11` will be loaded into the register, `x` will stay `1`, and `n1n0` becomes `00`.

When another clock edge comes, `00` will be loaded into the register. Soon after, `x` will become `0`, and if `b` is `0`, `n1n0` will stay `00`; if `b` is `1`, `n1n0` will become `01`. Notice that the system is back in the state where it started.

Understanding how a state register and combinational logic implement a state machine can take a while, since in a particular state (indicated by the value presently in the state register), we generate

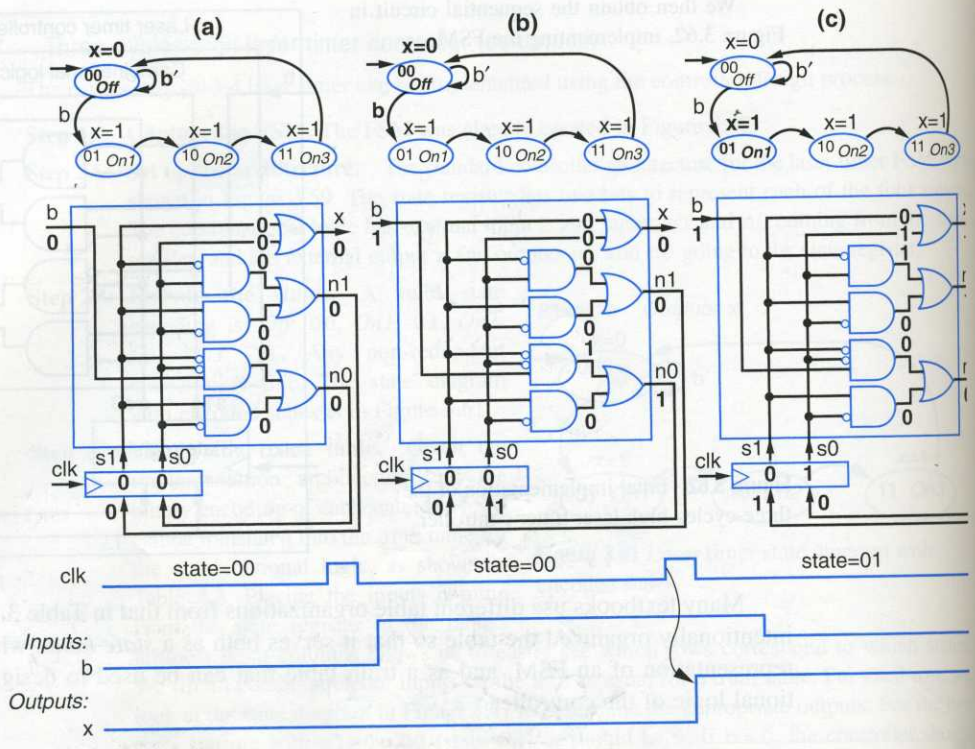


Figure 3.63 Tracing the behavior of the three-cycles-high laser timer controller.

the external output for that state, and we generate the signals for the *next* state—but we don't transition to that next state (i.e., we don't load the state register) until the next clock edge.

Example 3.9 Button press synchronizer

This example develops a circuit that synchronizes a button press to a clock signal, such that when the button is pressed, the result is a signal that is `1` for exactly one clock cycle. Such a synchronized signal is useful to prevent a single button press that lasts multiple cycles from being interpreted as multiple button presses. Figure 3.64 uses a timing diagram to illustrate the desired behavior.

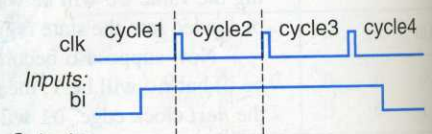


Figure 3.64 Desired timing diagram of the button press synchronizer.

The circuit's input will be a signal `bi`, and the output a signal `bo`. When `bi` becomes `1`, representing the button being pressed, the system should set `bo` to `1` for exactly one cycle. The system waits for `bi` to return to `0` again, and then waits for `bi` to become `1` again, which would represent the next pressing of the button.

Step 1: Capture the FSM. Figure 3.65(a) shows an FSM describing the circuit's behavior. The FSM waits in state *A*, outputting `bo=0`, until `bi` is `1`. The FSM then transitions to state *B*, outputting bo=1. The FSM will then transition to either state *A* or *C*, which both set

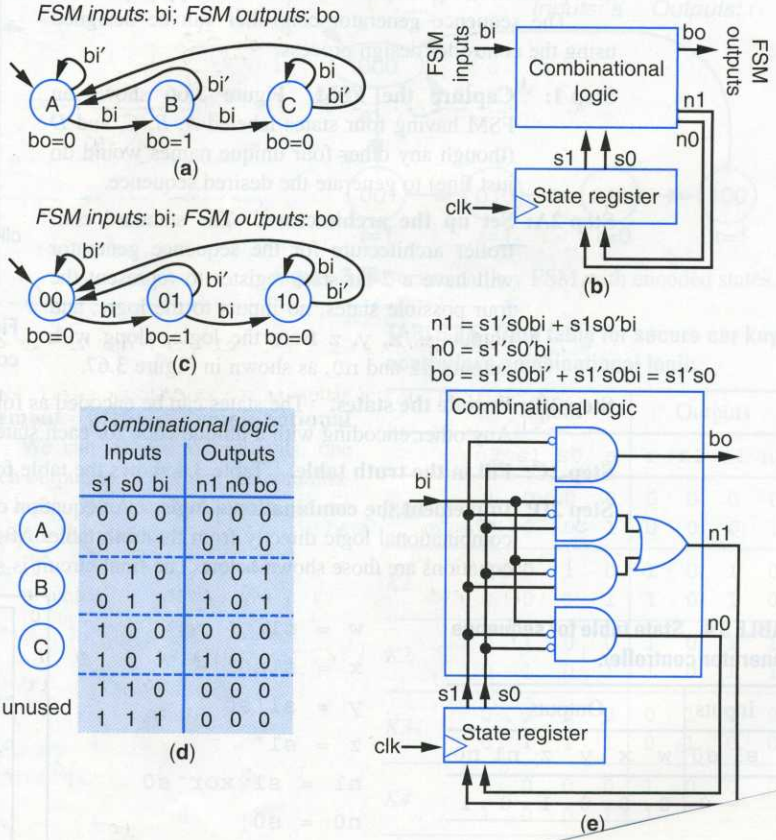
bo=0 again, so that bo was 1 for just one cycle, as desired. The FSM goes from *B* to *A* if bi returned to 0. If bi is still 1, the FSM goes to state *C*, where the FSM waits for bi to return 0, causing a transition back to state *A*.

Step 2A: Set up the architecture. Because the FSM has three states, the architecture has a two-bit state register, as in Figure 3.65(b).

Step 2B: Encode the states. The three states can be straightforwardly encoded as 00, 01, and 10, as in Figure 3.65(c).

Step 2C: Fill in the truth table. We convert the FSM with encoded states to a truth table for the controller's combinational logic, as shown in Figure 3.65(d). For the unused state 11, we have chosen to output bo=0 and return to state 00.

Step 2D: Implement the combinational logic. We derive the equations for each combinational logic output, as shown in Figure 3.65(e), and then create the final circuit as shown.



Combinational logic

	Inputs			Outputs		
	s1	s0	bi	n1	n0	bo
A	0	0	0	0	0	0
	0	0	1	0	1	0
B	0	1	0	0	0	1
	0	1	1	1	0	1
C	1	0	0	0	0	0
	1	0	1	1	0	0
unused	1	1	0	0	0	0
	1	1	1	0	0	0

(d)

Figure 3.65 Button press synchronizer design steps: (a) initial FSM, (b) controller architecture, (c) FSM with encoded states, (d) truth table for the combinational logic, (e) final controller with implemented combinational logic.

Example 3.10 Sequence generator

This example designs a sequential circuit with four outputs: w, x, y, and z. The circuit should generate the following sequence of output patterns: 0001, 0011, 1100, and 1000, one per clock cycle. After 1000, the circuit should repeat the sequence. Sequence generators are common in a variety of systems, such as a system that blinks a set of four lights in a particular pattern for a festive lights display. Another example is a system that rotates an electric motor a fixed number of degrees each clock cycle by powering magnets around the motor in a specific sequence to attract the magnetized motor to the next position in the rotation—known as a ***stepper motor***, because the motor rotates in steps.

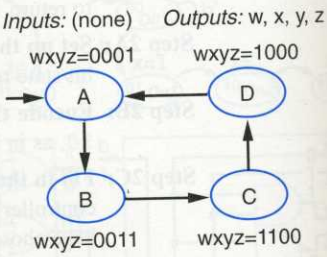


Figure 3.66 Sequence generator FSM.

The sequence generator controller can be designed using the controller design process:

Step 1: Capture the FSM. Figure 3.66 shows an FSM having four states labeled *A*, *B*, *C*, and *D* (though any other four unique names would do just fine) to generate the desired sequence.

Step 2A: Set up the architecture. The standard controller architecture for the sequence generator will have a 2-bit state register to represent the four possible states, no inputs to the logic, and outputs w, x, y, z from the logic, along with outputs n1 and n0, as shown in Figure 3.67.

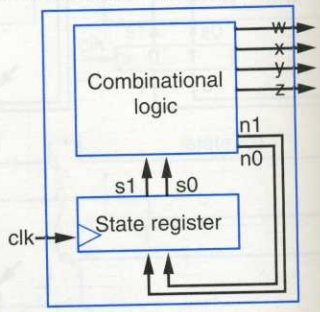


Figure 3.67 Sequence generator controller architecture.

Step 2B: Encode the states. The states can be encoded as follows—*A*: 00, *B*: 01, *C*: 10, *D*: 11. Any other encoding with a unique code for each state would also be fine.

Step 2C: Fill in the truth table. Table 3.4 shows the table for the FSM with encoded states.

Step 2D: Implement the combinational logic. An equation can be derived for each output of the combinational logic directly from the truth table. After some algebraic simplification, the equations are those shown below. The final circuit is shown in Figure 3.68.

TABLE 3.4 State table for sequence generator controller.

	Inputs		Outputs					
	s1	s0	w	x	y	z	n1	n0
A	0	0	0	0	0	1	0	1
B	0	1	0	0	1	1	1	0
C	1	0	1	1	0	0	1	1
D	1	1	1	0	0	0	0	0

```
w  = s1
x  = s1s0'
y  = s1's0
z  = s1'
n1 = s1 xor s0
n0 = s0'
```

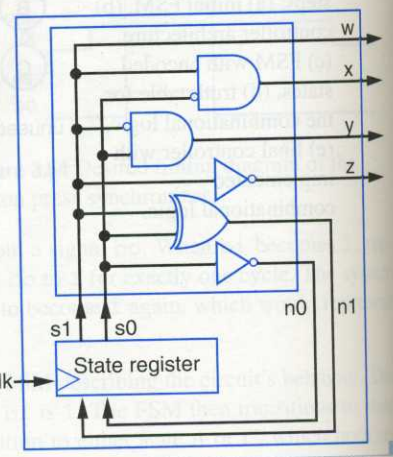


Figure 3.68 Sequence generator controller with implemented combinational logic.

Example 3.11 Secure car key controller (continued)

Let's complete the design for the secure car key controller from Example 3.4. We already carried out **Step 1: Capture the FSM**, shown in Figure 3.50. The remaining steps are as follows.

Step 2A: Set up the architecture. The FSM has five states, and thus requires a 3-bit state register, which can represent eight states; three states will be unused. The inputs to the combinational logic are `a` and the three state bits `s2`, `s1`, and `s0`, while the outputs are signal `r` and next state outputs `n2`, `n1`, and `n0`. The architecture is shown in Figure 3.69.

Step 2B: Encode the states. Let's encode the states using a straightforward binary encoding of `000` through `100`. The FSM with state encodings is shown in Figure 3.70.

Step 2C: Fill in the truth table. The FSM converted to a truth table for the logic is shown in Table 3.5. For the unused states, we have chosen to set `r = 0` and the next state to `000`.

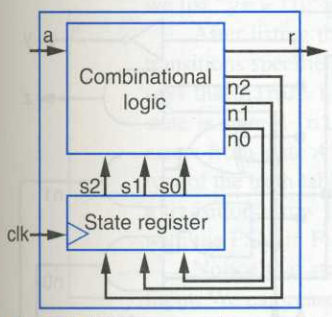


Figure 3.69 Secure car key controller architecture.

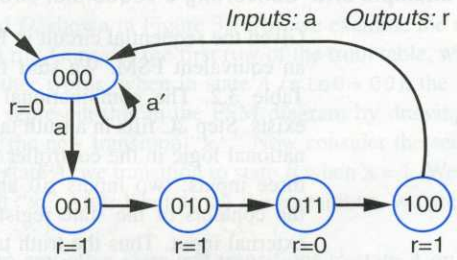


Figure 3.70 Secure car key FSM with encoded states.

Step 2D: Implement the combinational logic. We can design four circuits, one for each output, to implement the combinational logic. We leave this step as an exercise for the reader.

TABLE 3.5 Truth table for secure car key controller's combinational logic.

	Inputs				Outputs			
	s2	s1	s0	a	r	n2	n1	n0
Wait	0	0	0	0	0	0	0	0
	0	0	0	1	0	0	0	1
K1	0	0	1	0	1	0	1	0
	0	0	1	1	1	0	1	0
K2	0	1	0	0	1	0	1	1
	0	1	0	1	1	0	1	1
K3	0	1	1	0	0	1	0	0
	0	1	1	1	0	1	0	0
K4	1	0	0	0	1	0	0	0
	1	0	0	1	1	0	0	0
Unused	1	0	1	0	0	0	0	0
	1	0	1	1	0	0	0	0
	1	1	0	0	0	0	0	0
	1	1	0	1	0	0	0	0
	1	1	1	0	0	0	0	0
	1	1	1	1	0	0	0	0

Converting a Circuit to an FSM (Reverse Engineering)

We showed in Section 2.6 that a circuit, truth table, and equation were all forms able to represent the same combinational function. Similarly, a circuit, state table, and FSM are all forms able to represent the same sequential function.

The process in Table 3.2 for converting an FSM to a circuit can be applied in reverse to convert a circuit to an FSM. In general, converting a circuit to an equation or FSM is known as ***reverse engineering*** the behavior of the circuit. Not only is reverse engineering useful to help develop a better understanding of sequential circuit design, but it can also be used to understand the behavior of a previously-designed circuit such as a circuit created by a designer who is no longer at a company, and also to check that a circuit we designed has the correct behavior.

Example 3.12 Converting a sequential circuit to an FSM

Given the sequential circuit in Figure 3.71, find an equivalent FSM. We start from step 2D in Table 3.2. The combinational circuit already exists. Step 2C fills in a truth table. The combinational logic in the controller architecture has three inputs: two inputs `s0` and `s1` represent the contents of the state register, and `x` is an external input. Thus the truth table will have 8 rows because there are $2^3 = 8$ possible combinations of inputs. After listing the truth table and enumerating all combinations of inputs (e.g., `s1s0x = 000`, ..., `s1s0x = 111`), the techniques in Section 2.6 can be used to fill in the values of the outputs. Consider the output `y`. The combinational circuit shows that `y = s1'`. Knowing this, we place a `1` in the `y` column of the truth table in every row where `s1 = 0`, and place a `0` in the remaining spaces in the `y` column. Consider `n0`, which the circuit shows as having the Boolean equation `n0 = s1's0'x`. Accordingly, we set `n0` to `1` when `s1 = 0` and `s0 = 0` and `x = 1`. We fill in the columns for `z` and `n1` using a similar analysis and move on to the next step.

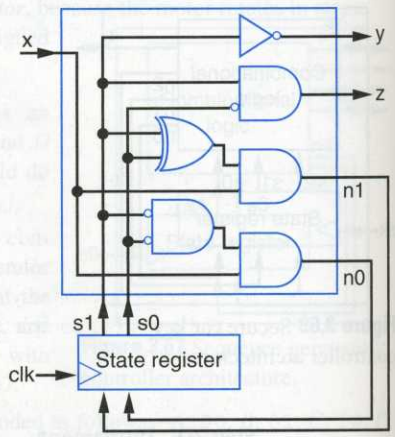


Figure 3.71 Circuit with unknown behavior.

Step 2B encodes the states. The states have already been encoded, so this step in reverse assigns a name to encoded state. We arbitrarily choose the names *A*, *B*, *C*, and *D*, seen in Table 3.6.

Step 2A sets up the standard controller architecture. This step requires no work since the controller architecture was already defined.

Finally, step 1 captures the FSM. Initially, we can set up an FSM diagram with the four states whose names were given in step 2A, shown in Figure 3.72(a). Next, we list the values of the FSM outputs `y` and `z` next to each state as defined by the truth table

TABLE 3.6 Truth table for circuit.

	Inputs			Outputs			
	s1	s0	x	n1	n0	y	z
A	0	0	0	0	0	1	0
	0	0	1	0	1	1	0
B	0	1	0	0	0	1	0
	0	1	1	1	0	1	0
C	1	0	0	0	0	0	1
	1	0	1	1	0	0	1
D	1	1	0	0	0	0	0
	1	1	1	0	0	0	0

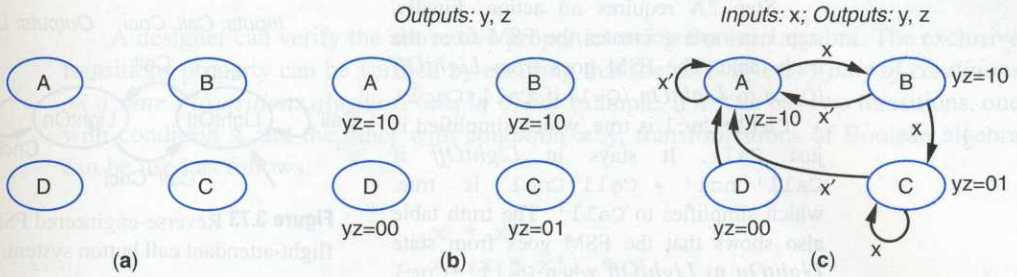


Figure 3.72 Converting a truth table to an FSM diagram: (a) initial FSM, (b) FSM with outputs specified, and (c) FSM with outputs and transitions specified.

in Table 3.6. For example, in state *A* (`s1s0 = 00`), the outputs y and z are `1` and `0`, respectively, so we list "`yz = 10`" with state *A* in the FSM.

After listing the outputs for states *B*, *C*, and *D*, shown in Figure 3.72(b), we examine the state transitions specified in the truth table by `n1` and `n0`. Consider the first row of the truth table, which says that `n1n0 = 00` when `s1s0x = 000`. In other words, when in state *A* (`s1s0 = 00`), the next state is state *A* (`n1n0 = 00`) if `x` is `0`. We can represent this in the FSM diagram by drawing an arrow from state *A* back to state *A* and labeling the new transition "`x'`." Now consider the second row of the truth table, which indicates that from state *A*, we transition to state *B* when `x = 1`. We add a transition arrow from state *A* to *B* and label it "x." After labeling all the transitions, we are left with the FSM in Figure 3.72(c).

Notice that state *D* cannot be reached from any other state and transitions to state *A* on any input. We can reasonably infer that the original FSM had only three states and state *D* is an extra, unused state. For completeness, it is preferable to leave state *D* in the final diagram, however.

Given any synchronous circuit consisting of logic gates and flip-flops, we can always redraw the circuit as consisting of a state register and logic—the standard controller architecture—just by grouping all the flip-flops together. Thus, the approach described above works for any synchronous circuit, not just a circuit already drawn in the form of the standard controller architecture.

Example 3.13 Reverse engineering the D-flip-flop-based flight-attendant call button system

Figure 3.34 showed a sequential circuit designed in an ad hoc manner rather than using this chapter's controller design process. Reverse engineering that circuit proceeds as follows. Treating the D flip-flop as a one-bit state register with input D and output Q, step 2D obtains the equation for the controller's output to the light as `L = Q`, and for the controller's next state as `D = Cncl'Q + Call`. Step 2C creates a truth table. The inputs to the combinational logic are `Q`, `Call`, and `Cncl`, while the outputs are `D` and `L`. The table is shown in Table 3.7, filling the output values based on the above equations for `L` and `D`. For step 2B in reverse, we give the name *LightOff* to the state `Q=0`, and *LightOn* to `Q=1`.

TABLE 3.7 Truth table for circuit.

	Inputs			Outputs	
	Q	Call	Cncl	D	L
Light Off	0	0	0	0	0
	0	0	1	0	0
	0	1	0	1	0
	0	1	1	1	0
Light On	1	0	0	1	1
	1	0	1	0	1
	1	1	0	1	1
	1	1	1	1	1

Step 2A requires no action. Finally, step 1 in reverse creates the FSM from the truth table. The FSM goes from *LightOff* (`Q=0`) to *LightOn* (`Q=1`) if `Call*Cncl' + Call*Cncl` is true, which simplified is just `Call`. It stays in *LightOff* if `Call'Cncl' + Call'Cncl` is true, which simplifies to `Call'`. The truth table also shows that the FSM goes from state *LightOn* to *LightOff* when `Call'*Cncl` is true. It stays in *LightOn* if the condition `Call'Cncl' + Call*Cncl' + Call*Cncl` is true, which simplifies to `Call'Cncl' + Call`, which further simplifies to `Cncl' + Call`.

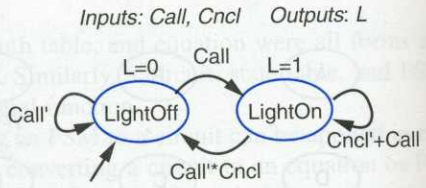


Figure 3.73 Reverse-engineered FSM for flight-attendant call button system.

Note that this FSM is equivalent to the FSM in Figure 3.53 created directly to describe the flight-attendant call button system's desired behavior; the conditions that look different can be manipulated to be the same. Thus, the circuit built using the ad hoc approach seems to be correct in this case.

Common Mistakes when Capturing FSMs

Some mistakes are commonly made when capturing an FSM, relating to properties regarding the transitions leaving a state. In short, *one and only one* transition condition should ever evaluate to true during any rising clock edge. The common mistakes involve:

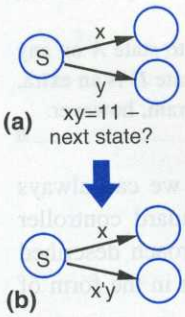


Figure 3.74 A state's transitions should be exclusive.

1. ***Non-exclusive transitions***—For a given state, when a rising clock comes, all the state's transitions should be ***exclusive***, meaning no more than one transition condition should be true. Consider an FSM with inputs `x` and `y`, and a state *S* with the two outgoing transitions shown in Figure 3.74(a). What happens when `x = 1` and `y = 1`—which transition should the FSM take? An FSM should be ***deterministic***, meaning the transition to take can always be uniquely determined. The FSM creator might label the transitions "`x`" and "`x'y`" as shown in Figure 3.74(b) to solve the problem. Actually, a particular type of FSM known as a ***nondeterministic FSM*** does allow more than one condition to be true and chooses among them randomly. But we want deterministic FSMs when designing circuits, so we won't consider nondeterministic FSMs further.

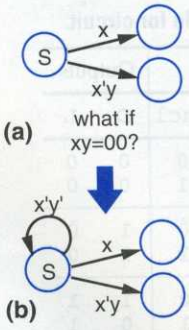


Figure 3.75 A state's transitions must be complete.

2. ***Incomplete transitions***—For a given state, when a rising clock edge comes, the state's transitions must be ***complete***, meaning *one* of the transitions from that state must have a true condition. In other words, every input combination should be accounted for in every state. Designers sometimes forget to ensure this. For example, consider an FSM with inputs `x` and `y`, and a state *S* with the outgoing transitions shown in Figure 3.75(a). What happens if the FSM is in *S*, and `x = 0` and `y = 0`? Neither of the two transitions from the state has a true condition. The FSM is incompletely specified. An FSM creator can add a third transition, indicating what state to go to if `x'y'` is true, as in Figure 3.75(b). The three transitions now cover all possible values of `x` and `y`. A commonly forgotten transition is a transition pointing from a state back to itself. Sometimes making a transition the complement of another transition is a simple way to ensure completeness; e.g., if one of two transitions has the condition `xy`, then the other transition can be given the condition `(xy)'` (avoid trying to write that other condition as `x'+y'` as that commonly leads to mistakes).

A designer can verify the above two properties using Boolean algebra. The exclusive transitions property can be verified by ensuring that the *AND of every pair of conditions on a state's transitions always results in* `0`. For example, if a state has two transitions, one with condition `x` and the other with condition `x'y`, transformations of Boolean algebra can be used as follows:

```
x * x'y
= (x * x') * y
= 0 * y
= 0
```

If a state has three transitions with conditions *C1*, *C2*, and *C3*, the designer can verify that *C1*C2*=`0`, *C1*C3*=`0`, and finally that *C2*C3*=`0`, thus verifying that every pair yields `0`. Note that verifying that *C1*C2*C3*=`0` does *not* verify that the transitions are exclusive; for example, if *C1* and *C2* were exclusive but *C2* and *C3* were not, *C1*C2*C3* would still equal `0` because `0`**C3*=`0`.

The second property of complete transitions can be verified by checking that the *OR of all the conditions on a state's transitions results in* `1`. Considering the same example of a state that has two transitions, one with condition `x` and the other with condition `x'y`, transformations of Boolean algebra can be applied as follows:

```
x + x'y
= x*(1+y) + x'y
= x + xy + x'y
= x + (x+x')y
= x + y
```

The OR of those two conditions is not `1` but rather `x+y`. If `x` and `y` were both `0`, neither condition would be true, and so the next state would not be specified in the FSM. Figure 3.75(b) fixed this problem by adding another transition, `x'y'`. Checking these transitions yields:

```
x + x'y + x'y'
= x + x'(y+y')
= x + x'*1
= x + x'
= 1
```

If a state has three transitions with conditions *C1*, *C2*, and *C3*, the designer can verify that *C1*+*C2*+*C3*=`1`.

Proving the properties for the transitions of every state can be time-consuming. A good FSM capture tool will verify the above two properties automatically and inform the designer of any problems.

Example 3.14 Verifying transition properties for the code detector FSM

Recall the code detector from Example 3.6. Suppose a designer instead captured the behavior as shown in Figure 3.76, using different conditions for the transitions leaving the states *Start*, *Red1*, *Blue*, and *Green*. We want to verify the exclusive transition property for the transitions leaving state *Start*. There are three conditions: `ar`, `a'`, and `a(r'+b+g)`. We thus have three pairs of conditions. We AND each pair and prove that each equals 0 as follows:

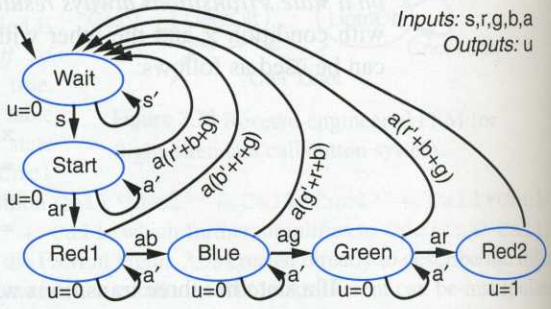


Figure 3.76 Problematic code detector FSM.

```
ar * a'
= (a*a')r
= 0*r
= 0
```

```
a' * a(r'+b+g)
= (a'*a)*(r'+b+g)
= 0*(r'+b+g)
= 0
```

```
ar * a(r'+b+g)
= (a*a)*r*(r'+b+g)
= a*r*(r'+b+g)
= arr'+arb+arg
= 0 + arb+arg
= arb + arg
= ar(b+g)
```

It appears the FSM has non-exclusive transitions, because the AND of the third pair of conditions does not result in `0`, which in turn means both conditions could be true at the same time—resulting in a nondeterministic FSM (if both conditions are true, what is the next state?). Recall from the code detector problem description that we want to transition from the *Start* state to the *Red1* state when a button is pressed (`a=1`) and that button is the red button and no other colored button is pressed. The FSM in Figure 3.76 has the condition `ar`. The mistake is under-specifying this condition; it should instead be `arb'g'`—in other words, a button has been pressed (`a`) and it is the red button (`r`) and the blue button has not been pressed (`b'`) and the green button has not been pressed (`g'`). The transition from *Start1* back to the *Wait* state could then be written as `a(rb'g')'` (which is the same as in Figure 3.76 after applying DeMorgan's Law). After this change, we can again try to verify the "only one condition is true" property for all pairs of the three conditions `arb'g'`, `a'`, and `a(rb'g')'`:

As evidence that this "pitfall" is indeed common, we admit that the hypothetical mistake in this example was in fact a mistake made in an early edition of this book. A reviewer of the book caught it. We added this example and this note to stress the point that the mistake is common.

```
arb'g' * a'
= aa'*rb'g'
= 0*rb'g'
= 0
```

```
a'*a(rb'g')'
= 0*(rb'g')'
= 0
```

```
arb'g' * a(rb'g')'
= a*a*(rb'g')*(rb'g')'
write rb'g' as Y for clarity...
= a*a*Y*Y'
= a*a*0
= 0
```

We would need to change the transition conditions of the other states similarly (as was done in Figure 3.58), and then check the pairs of conditions for those states' transitions too.

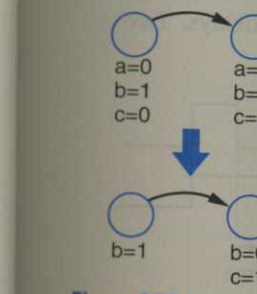


Figure 3.77
Unassigned outputs implicitly set to 0.

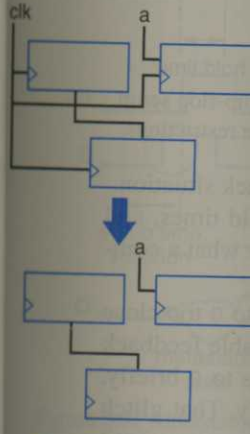


Figure 3.78 Implicit clock connections.

To verify the completeness property for state *Start*, we OR the three conditions and prove they equal `1`:

```
arb'g' + a'' + a(rb'g')'
= a' + arb'g' + a(rb'g')' (write rb'g' as Y for clarity)
= a' + aY + aY'
= a' + a(Y+Y') = a' + a(1)
= a' + a
= 1
```

We would need to check the property for all other states too.

FSM and Controller Conventions

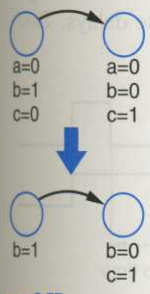


Figure 3.77 Unassigned outputs implicitly set to 0.

Simplifying FSM Notations: Unassigned Outputs

We already introduced the simplified FSM notation wherein every transition is implicitly ANDed with a rising clock edge. Another commonly used simplification involves assigning outputs. If an FSM has many outputs, listing the assignment of every output in every state can become cumbersome, and make the relevant behavior of the FSM hard to discern. A common simplifying notation is shown in Figure 3.77—if an output is not explicitly assigned in a state, the output is *implicitly* assigned a `0`. If the assignment of an output to `0` in a state is fundamental to understanding that state's behavior, then the output should still be explicitly assigned to `0` in order to aid someone trying to understand the behavior of the FSM.

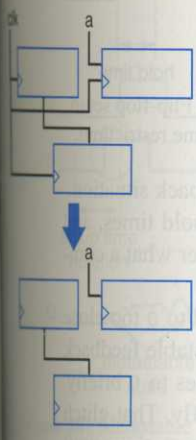


Figure 3.78 Implicit clock connections.

Simplifying Circuit Drawings: Implicit Clock Connections

Most sequential circuits have a single clock signal connected to all sequential components. A component is known to be sequential because of the small triangle input drawn on the component's block symbol. Many circuit drawings therefore use a simplification wherein the clock signal is assumed to be connected to all sequential components, as in Figure 3.78. This simplification leads to less cluttered wiring in the drawing.

Mathematical Formalisms in Combinational and Sequential Circuit Design

This book has thus far described two mathematical formalisms, Boolean functions and FSMs, for designing combinational and sequential circuits, respectively. Note that those formalisms aren't *necessary* to design circuits. Recall that the first attempt at building a three-cycles-high laser timer in Figure 3.41 just involved connecting components together in the hopes of creating a correctly working circuit. However, using those formalisms provides for a structured method of designing circuits. Those formalisms also provide the basis for powerful automated tools to assist with design, such as a tool that would automatically check for the common pitfalls described earlier in this section, tools that automatically convert Boolean equations or FSMs into circuits, tools that verify that two circuits are equivalent, or tools that simulate systems. The chapter scarcely touched on all the benefits of those mathematical formalisms relating to automating the various aspects of designing circuits and of verifying that the circuits behave properly. The importance of using sound mathematical formalisms to guide design cannot be overstated.

▶ 3.5 MORE ON FLIP-FLOPS AND CONTROLLERS

Non-Ideal Flip-Flop Behavior

When first learning digital design we assume ideal behavior for logic gates and flip-flops, just like when first learning physics of motion we assume there's no friction or wind resistance. However, there is a non-ideal behavior of flip-flops—metastability—that is such a common problem in the practice of real digital design, we feel obliged to discuss the issue briefly here. Digital designers in practice should study metastability and possible solutions quite thoroughly before doing serious designs. Metastability comes from failing to meet flip-flop setup or hold times, which are now introduced.

Setup Times and Hold Times

Flip-flops are built from wires and logic gates, and wires and logic gates have delays. Thus, a real flip-flop imposes some restrictions on when the flip-flop's inputs can change relative to the clock edge, in order to ensure correct operation despite those delays. Two important restrictions are:

- ***Setup time:*** The inputs of a flip-flop (e.g., the D input) must be stable for a minimum amount of time, known as the ***setup time***, *before* a clock edge arrives. This intuitively makes sense—the input values must have time to propagate through any flip-flop internal logic and be waiting at the internal gates' inputs before the clock pulse arrives.
- ***Hold time:*** The inputs of a flip-flop must remain stable for a minimum amount of time, known as the ***hold time***, *after* a clock edge arrives. This also makes intuitive sense—the clock signal must have time to propagate through the internal gates to create a stable feedback situation.

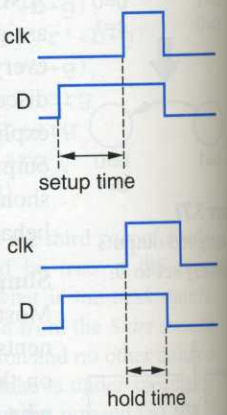


Figure 3.79 Flip-flop setup and hold time restrictions.

A related restriction is on the minimum clock pulse width—the pulse must be wide enough to ensure that the correct values propagate through the internal logic and create a stable feedback situation.

A flip-flop typically comes with a datasheet describing setup times, hold times, and minimum clock pulse widths. A ***datasheet*** is a document that tells a designer what a component does and how to properly use that component.

Figure 3.80 illustrates an example of a setup time violation. `D` changed to `0` too close to the rising clock. The result is that `R` was not `1` long enough to create a stable feedback situation in the cross-coupled NOR gates with `Q` being `0`. Instead, `Q` glitches to `0` briefly. That glitch feeds back to the top NOR gate, causing `Q'` to glitch to `1` briefly. That glitch feeds back to the bottom NOR gate, and so on. The oscillation would likely continue until a race condition caused the circuit to settle into a stable situation of `Q = 0` or `Q = 1`—or the circuit could enter a metastable state, which we now describe.

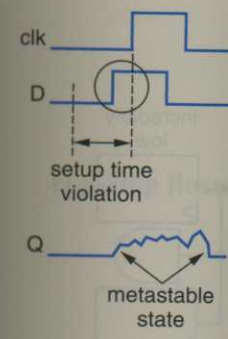


Figure 3.81 Metastable flip-flop state caused by a setup time violation.

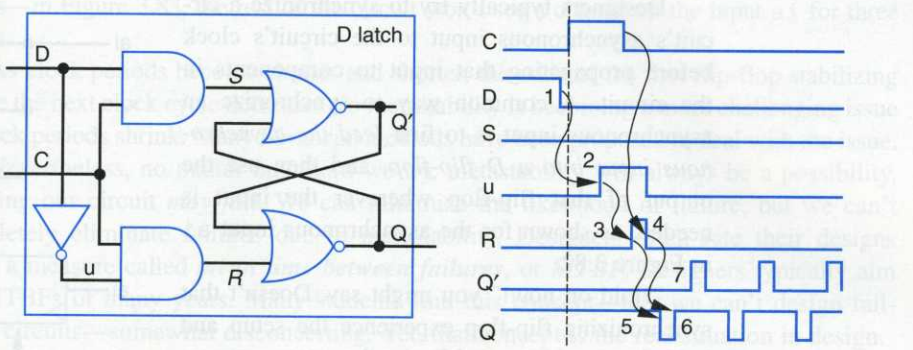


Figure 3.80 Setup time violation: D changed to 0 (1) too close to the rising clock. u changed to 1 after the inverter delay (2), and then R changed to 1 after the AND gate delay (3). But then the clock pulse was over, causing R to change back to 0 (4) before a stable feedback situation with Q=0 occurred in the cross-coupled NOR gates. R's change to 1 did cause Q to change to 0 after the NOR gate delay (5), but R's change back to 0 caused Q to change right back to 1 (6). The glitch of a 0 on Q fed back into the top NOR gate, causing Q' to glitch to 1 (7). That glitch of a 1 fed back to the bottom NOR gate, causing another glitch of a 0 on Q. That glitch runs around the cross-coupled NOR gate circuit (oscillation)—a race condition would eventually cause Q to settle to 1 or 0, or possibly enter a metastable state (to be discussed).

Metastability

If a designer fails to ensure that a circuit obeys the setup and hold times of a flip-flop, the result could be that the flip-flop enters a metastable state. A flip-flop in a ***metastable state*** is in a state other than a stable 0 or a stable 1. Metastable in general means that a system is only marginally stable—the system has other states that are far more stable. A flip-flop in a metastable state may have an output with a value that is not a 0 or a 1, instead outputting a voltage somewhere between that of a 0 and that of a 1. That voltage may also oscillate. Such an output is clearly a problem. Since a flip-flop's output is connected to other components like logic gates and other flip-flops, that strange voltage value may cause other components to output strange values, and soon the values throughout an entire circuit can be in bad shape.

Why would we ever violate setup and hold times? After all, within a circuit we design, we can measure the longest possible path from any flip-flop output to any flip-flop input. As long as we make the clock period sufficiently longer than that longest path, we can ensure the circuit obeys setup times. Likewise, we can ensure that hold times are satisfied too.

The problem is that our circuit likely has to interface to external inputs, and we can't control when those inputs change, meaning those inputs may violate setup and hold times when connected to flip-flop inputs. For example, an input may be connected from a button being pressed by a user—the user can't be told to press the button so many nanoseconds before a clock edge and to be sure to hold the button so many nanoseconds after the clock edge so that setup and hold times are satisfied. So metastability is a problem primarily when a flip-flop has inputs that are not synchronized with the circuit's

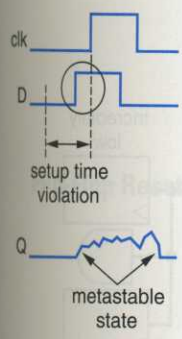


Figure 3.81 Metastable flip-flop state caused by a setup time violation.

clock—in other words, metastability is a problem when dealing with ***asynchronous inputs***.

Designers typically try to synchronize a circuit's asynchronous input to the circuit's clock before propagating that input to components in the circuit. A common way to synchronize an asynchronous input is to first *feed the asynchronous input into a D flip-flop*, and then use the output of that flip-flop wherever the input is needed, as shown for the asynchronous input `ai` in Figure 3.82.

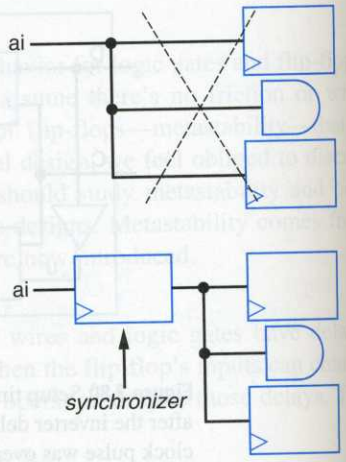


Figure 3.82 Feeding an asynchronous external input into a single flip-flop can reduce metastability problems.

"Hold on now!" you might say. Doesn't that synchronizing flip-flop experience the setup and hold time problem, and hence the same metastability issue? Yes, that's true. But at least the asynchronous input directly affects only *one* flip-flop, rather than perhaps several or dozens of flip-flops and other components. And that synchronizer flip-flop is specifically introduced for synchronization purposes and has no other purpose, whereas other flip-flops are being used to store bits for other purposes. We can therefore choose a flip-flop for the synchronizer that minimizes the metastability problem—we can choose an extremely fast flip-flop, and/or one with very small setup and hold times, and/or one with special circuitry to minimize metastability. That flip-flop may be bigger than normal or consume more power than normal, but there's only one such flip-flop per asynchronous input, so those issues aren't a problem. Bear in mind that no matter what we do, though, the synchronizer flip-flop could still become metastable, but at least we can minimize the odds of a metastable state happening by choosing a good flip-flop.

Another thing to consider is that a flip-flop will typically not stay metastable for very long. Eventually, the flip-flop will "topple" over to a stable `0` or a stable `1`, like how a coin tossed onto the ground may spin for a while (a metastable state) but will eventually topple over to a very stable head or tail. What many designers therefore do is introduce two or more flip-flops in series for synchronization purposes, as shown in Figure 3.83. So even if the first flip-flop becomes metastable, that flip-flop will likely reach a stable state before the next clock cycle, and thus the second flip-flop is even less likely to go metastable. Thus the odds of a

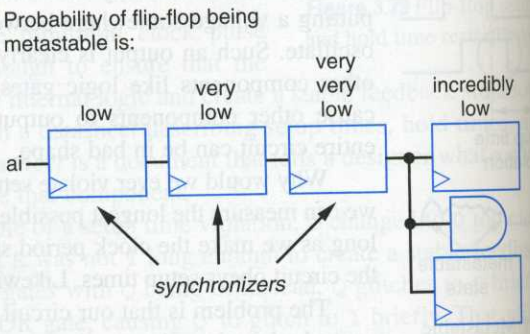


Figure 3.83 Synchronizer flip-flops reduce probability of metastability in a circuit's regular flip-flops.

metastable signal actually making it to our circuit's normal flip-flops are very low. This approach has the obvious drawback of delaying changes on the input signal by several cycles—in Figure 3.83, the rest of the circuit won't see a change on the input `ai` for three cycles.

As clock periods become shorter and shorter, the odds of the first flip-flop stabilizing before the next clock cycle decreases, so metastability is becoming a more challenging issue as clock periods shrink. Many advanced methods have been proposed to deal with the issue.

Nevertheless, no matter how hard we try, metastability will always be a possibility, meaning our circuit *may fail*. We can minimize the likelihood of failure, but we can't completely eliminate failures due to metastability. Designers often rate their designs using a measure called ***mean time between failures***, or ***MTBF***. Designers typically aim for MTBFs of many years. Many students find this concept—that we can't design fail-proof circuits—somewhat disconcerting. Yet, that concept is the real situation in design.

Designers of serious high-speed digital circuits should study the problem of metastability, and modern solutions to the problem, thoroughly.

Example 3.15 Adding a flip-flop to an asynchronous input to reduce the metastability problem

Figure 3.69 showed the controller circuit for a secure car key controller. Assuming the input `a` is asynchronous, then changes on input `a` could propagate through the controller's combinational logic to the state register's flip-flops such that flip-flop setup or hold times would be violated, resulting in metastable values. A synchronizer flip-flop could be added to the circuit's input to reduce the likelihood of metastability problems, as shown in Figure 3.84.

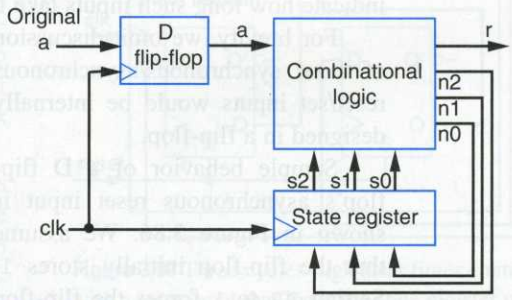


Figure 3.84 Secure car key controller extended with D flip-flop on asynchronous input to reduce chances of metastability problems.

Flip-Flop Reset and Set Inputs

Some D flip-flops (as well as other flip-flop types) come with extra inputs that can force the flip-flop to `0` or `1`, independently of the D input. One such input is a ***clear***, or ***reset***, input that forces the flip-flop to `0`. Another such input is a ***set*** input that forces the flip-flop to `1`. Reset and set inputs are very useful for initializing flip-flops to an initial value (e.g., initializing all flip-flops to `0`s) when powering up or resetting a system. These reset and set inputs should not be confused with the `R` and `S` inputs of an RS latch or flip-flop—the reset and set inputs are special control inputs to any type of flip-flop (D, RS, T, JK) that take priority over the normal data inputs of a flip-flop.

The reset and set inputs of a flip-flop may be either synchronous or asynchronous. A ***synchronous reset*** input forces the flip-flop to `0`, regardless of the value on the D input, during a rising clock edge. For the flip-flop in Figure 3.85(a), setting `R` to `1` forces the flip-flop to `0` on the next clock edge. Likewise, a ***synchronous set*** input forces the flip-flop to `1` on a rising clock edge. The reset and set inputs thus have priority over the `D` input. If a flip-flop has both a synchronous reset and a synchronous set input, the flip-flop datasheet must inform the flip-flop user which has priority if both inputs are set to `1`.

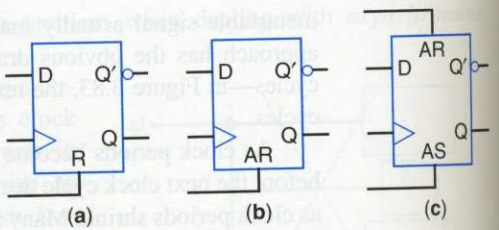


Figure 3.85 D flip-flops with: (a) synchronous reset `R`, (b) asynchronous reset `AR`, and (c) asynchronous reset and set.

An ***asynchronous reset*** clears the flip-flop to `0` independently of the clock signal—the clock does not need to be rising, or even be `1`, for the asynchronous reset to occur—hence the term "asynchronous." Likewise, an ***asynchronous set***, also known as ***preset***, can be used to asynchronously set the flip-flop to `1`. A flip-flop's datasheet must indicate how long such inputs take to have effect, usually at least 1 clock cycle.

For brevity, we omit discussion of how synchronous/asynchronous reset/set inputs would be internally designed in a flip-flop.

Sample behavior of a D flip-flop's asynchronous reset input is shown in Figure 3.86. We assume that the flip-flop initially stores `1`. Setting `AR` to `1` forces the flip-flop to `0`, independent of any clock edge. When the next clock edge appears, `AR` is still `1`, so the flip-flop stays `0` even though the input `D` is `1`. When `AR` returns to `0`, the flip-flop follows the `D` input on successive clock edges, as shown.

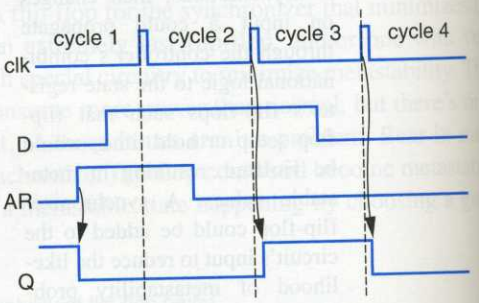
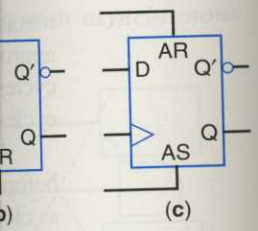


Figure 3.86 Asynchronous reset forces the flip-flop output `Q` to `0`, independent of `clk` or `D` input

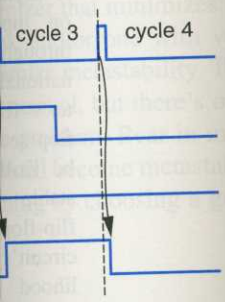
Initial State of a Controller

Particularly observant readers may have come up with a question when an earlier section implemented FSMs as controllers: what happened to the indication of the initial state of an FSM when we designed the controller implementing the FSM? The initial state of an FSM is the state that the FSM starts in when the FSM is first activated—or in controller terms, when the controller is first powered on. For example, the laser timer controller FSM in Figure 3.47 has an initial state of *Off*. When we converted graphical FSMs to truth tables, we ignored the initial state information. Thus, all of the controller circuits



th: (a) synchronous reset
AR, and (c) asynchronous

us set input, the flip-flop
h inputs are set to 1.
pendently of the clock
ne asynchronous reset to
nous set, also known as
lip-flop's datasheet must
t 1 clock cycle.



reset forces the flip-flop
t of clk or D input

on when an earlier section
ation of the initial state of
SM? The initial state of an
activated—or in controller
the laser timer controller
nverted graphical FSMs to
ll of the controller circuits

designed earlier in this chapter start in some random state based on whatever values happen to appear in the state register when the circuit is first powered on. Not knowing the initial state of a circuit could pose a problem—for example, we don't want our laser timer controller to start in a state that immediately turns on the laser.

One solution is to add an additional input, `reset`, to every controller. Setting `reset` to `1` should cause a load of the initial state into the state register. This initial state should be forced into the state register. The reset and set inputs of a flip-flop come in very handy in this situation. We can simply connect the controller's `reset` input to the reset and set inputs of the state register's flip-flops in a way that sets the flip-flops to the initial state when `reset` is `1`. For example, if the initial state of a 2-bit state register should be `01`, then we could connect the controller's reset input to reset and set inputs of the two flip-flops as shown in Figure 3.87.

Of course, for this reset functionality to work as desired, the designer must ensure that the controller's reset input is `1` when the system is first powered up. Ensuring that the reset input is `1` during power up can be handled using an appropriate electronic circuit connected to the on/off switch, the description of which is beyond the scope of this book.

Note that, if the synchronous reset or set inputs of a flip-flop are used, then the earlier-discussed setup and hold times, and associated metastability issues, apply to those reset and set inputs.

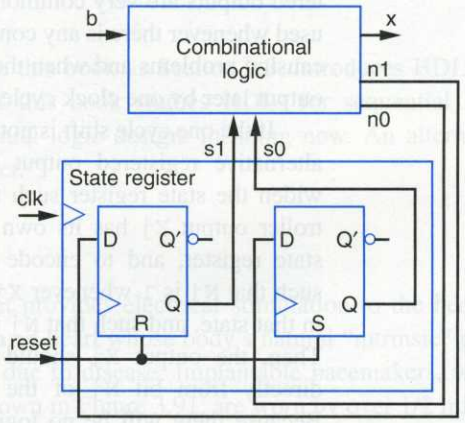


Figure 3.87 Three-cycles-high laser timer controller with a reset input that loads the state register with the initial state 01.

Non-Ideal Controller Behavior: Output Glitches

Glitching is the presence of temporary values on a wire typically caused by different delays of different logic paths leading to that wire. We saw an example of glitching in Figure 3.15. Glitching will also often occur when a controller changes states, due to different path lengths from each of the controller's state register flip-flops to the controller's outputs. Consider the three-cycles-high laser timer design in Figure 3.62. The laser should be off (output `x=0`) in state `s1s0=00` and on (`x=1`) in states `s1s0=01`, `s1s0=10`, and `s1s0=11`. However, the delay of the wire from `s1` to `x`'s OR gate in the figure could be longer than the delay of the wire from `s0` to that OR gate, perhaps due to different wire lengths. The result could be that when the state register changes state from `s1s0=01` to `s1s0=10`, the OR gate's inputs could momentarily become `00`. The OR gate would thus output `0` momentarily (a glitch). In the laser timer example, that glitch could momentarily shut off the laser—an undesired situation. Even worse would be glitches that momentarily turn *on* a laser.

A simple solution to controller output glitching is to add a flip-flop to the output. Figure 3.88 shows the laser-timer controller with a a flip-flop added to the x output. The flip-flop shifts the x output later by 1 clock cycle, which still results in three cycles high, but eliminates glitches from propagating to the x output—only the stable value appearing at the output would be loaded into the flip-flop on a rising clock edge. An output with a flip-flop added is called a ***registered output*** (think of the flip-flop as a one-bit register). Registered outputs are very common and should be used whenever there is any concern of glitches causing problems and when the shifting of the output later by one clock cycle is acceptable.

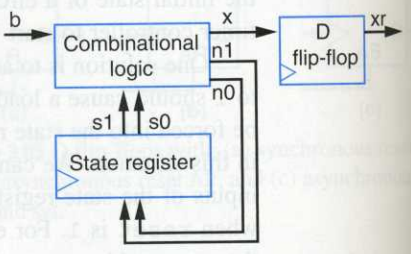


Figure 3.88 Laser timer controller with registered output to prevent glitches.

If the one cycle shift is not acceptable, an alternative registered output solution is to widen the state register such that each controller output Xj has its own bit Nj in the state register, and to encode each state *Sk* such that Nj is 1 whenever Xj is assigned 1 in that state, and such that Nj is 0 otherwise. Then, the output Xj should be connected directly from bit Nj of the state register. Because there will be no logic between the flip-flop for Nj and the output Xj, Xj will not experience glitches.

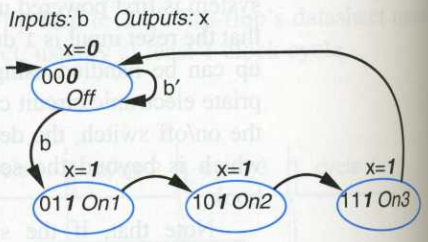


Figure 3.89 Laser timer state encoding with an extra bit corresponding to output x.

Figure 3.89 shows an encoding for the laser timer FSM where a bit has been added to the encoding corresponding to output x. The state encoding for state *Off* (which sets x to 0) has a 0 in that bit location, and has a 1 in the other three states (which set x to 1). Figure 3.90 shows how the controller would then connect x directly with its corresponding state register bit.

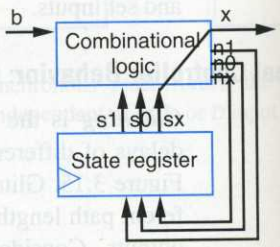


Figure 3.90 Laser timer controller with x connected to its state register bit.

Each of the above approaches has a drawback. The first approach shifts the outputs by one clock cycle and uses extra flip-flops. The second approach uses a wider state register and more combinational logic to compute the next state. Registered outputs therefore should be used when glitches should be avoided, like when the output controls a laser. In other cases, like in Chapter 5's use of controllers, the glitches don't cause problems and thus registered outputs aren't needed.

▶ 3.6 SEQUENTIAL LOGIC OPTIMIZATIONS AND TRADEOFFS (SEE SECTION 6.3)

The earlier sections described how to design basic sequential logic. This section, which physically appears in this book as Section 6.3, describes how to create *better* sequential logic (smaller, faster, etc.) using optimizations and tradeoffs. One use of this book involves studying sequential logic design optimizations and tradeoffs immediately after studying basic sequential logic design, meaning now. An alternative use studies sequential logic design optimizations and tradeoffs later, after studying the introduction of basic datapath components and RTL design (Chapters 4 and 5).

▶ 3.7 SEQUENTIAL LOGIC DESCRIPTION USING HARDWARE DESCRIPTION LANGUAGES (SEE SECTION 9.3)

This section, which physically appears in this book as Section 9.3, introduces HDLs for describing sequential logic. One use of this book studies HDLs for sequential logic immediately after studying basic sequential logic design, meaning now. An alternative use studies HDLs for sequential logic later.

▶ 3.8 PRODUCT PROFILE—PACEMAKER

A pacemaker is an electronic device that provides electrical stimulation to the heart to help regulate the heart's beating, steadying a heart whose body's natural "intrinsic" pacemaker is not working properly, perhaps due to disease. Implantable pacemakers, which are surgically placed under the skin as shown in Figure 3.91, are worn by over 1/2 million Americans. They are powered by a battery that lasts ten years or more. Pacemakers have improved the quality of life as well as lengthened the lives of many millions of people.

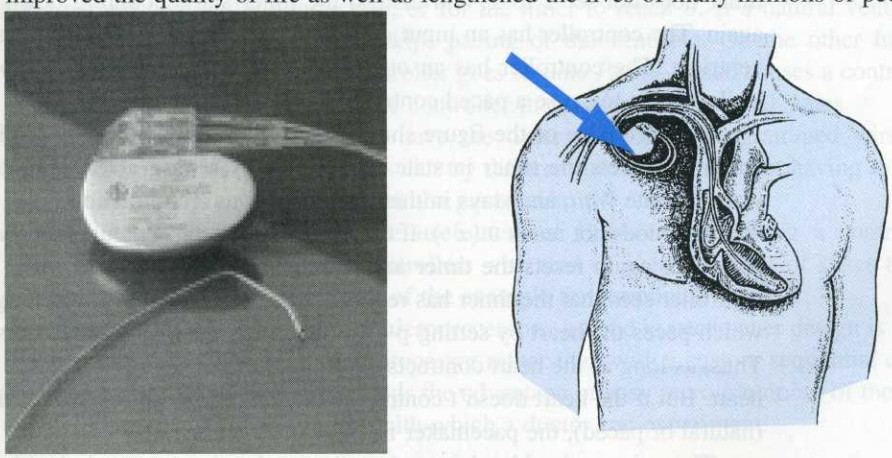


Figure 3.91 Pacemaker with leads (left), and pacemaker's location under the skin (right). Courtesy of Medtronic, Inc.

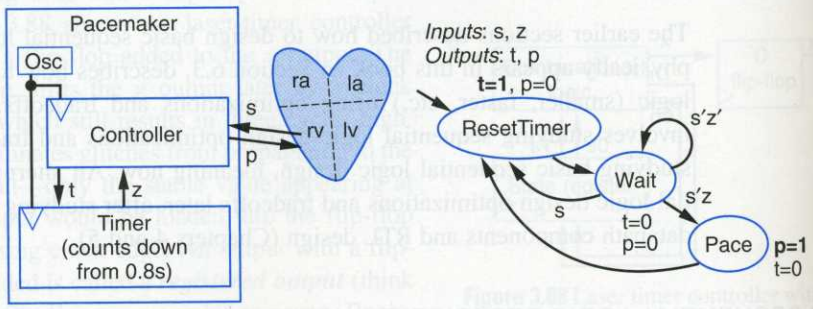


Figure 3.92 A basic pacemaker's controller FSM.

A heart has two atria (left and right) and two ventricles (left and right). The ventricles push the blood out to the arteries, while the atria receive the blood from the veins. A very simple pacemaker has one sensor to detect a natural contraction in the heart's right ventricle, and one output wire to deliver electrical stimulation to that right ventricle if the natural contraction doesn't occur within a specified time period—typically just under one second. Such electrical stimulation causes a contraction, not only in the right ventricle, but also the left ventricle.

We can describe the behavior of a simple pacemaker's controller using the FSM in Figure 3.92. The left side of the figure shows the pacemaker, consisting of a controller and a timer. The timer has an input `t`, which resets the timer when `t=1`. Upon being reset, the timer begins counting down from 0.8 seconds. If the timer counts down to `0`, the timer sets its output `z` to `1`. The timer could be reset before reaching `0`, in which case the timer does not set `z` to `1`, and instead the timer starts counting down from 0.8 seconds again. The controller has an input `s`, which is `1` when a contraction is sensed in the right ventricle. The controller has an output `p`, which the controller sets to `1` when the controller wants to cause a paced contraction.

The right side of the figure shows the controller's behavior as an FSM. Initially, the controller resets the timer in state *ResetTimer* by setting `t = 1`. Normally, the controller waits in state *Wait*, and stays in that state as long as a contraction is not detected (`s'`) and the timer does not reach `0` (`z'`). If the controller detects a natural contraction (`s`), then the controller again resets the timer and returns to waiting again. On the other hand, if the controller sees that the timer has reached `0` (`z = 1`), then the controller goes to state *Pace*, which paces the heart by setting `p=1`, after which the controller returns to waiting again. Thus, as long as the heart contracts naturally, the pacemaker applies no stimulation to the heart. But if the heart doesn't contract naturally within 0.8 seconds of the last contraction (natural or paced), the pacemaker forces a contraction.

The atria receive blood from the veins, and contract to push the blood into the ventricles. The atrial contractions occur just before the ventricular contractions. Therefore, many pacemakers, known as "atrioventricular" pacemakers, sense and pace not just the ventricular contractions, but also the atrial contractions. Such pacemakers thus have two

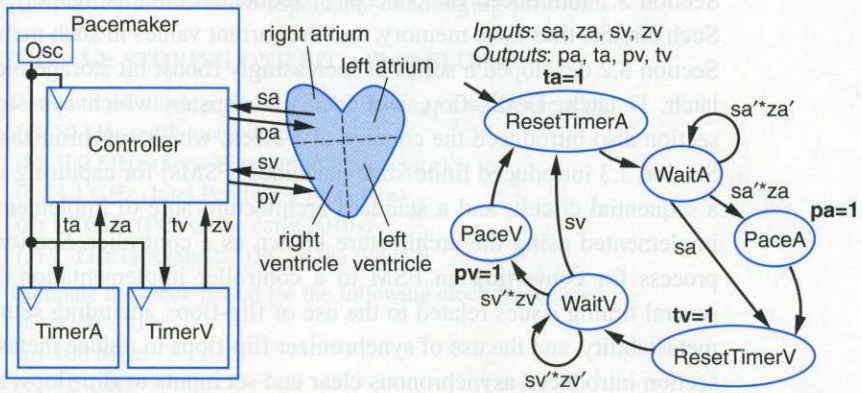


Figure 3.93 An atrioventricular pacemaker's controller FSM, using the convention that FSM outputs not explicitly set in a state are implicitly set to `0`.

sensors, and two output wires for electrical stimulation, and may provide better cardiac output, with the desirable result being higher blood pressure (Figure 3.93).

The pacemaker has two timers, one for the right atrium (*TimerA*) and one for the right ventricle (*TimerV*). The controller initially resets *TimerA* in state *ResetTimerA*, and then waits for a natural atrial contraction, or for the timer to reach `0`. If the controller detects a natural atrial contraction (`sa`), then the controller skips its pacing of the atrium. On the other hand, if *TimerA* reaches `0` first, then the controller goes to state *PaceA*, which causes a contraction in the atrium by setting `pa=1`. After an atrial contraction (either natural or paced), the controller resets *TimerV* in state *ResetTimerV*, and then waits for a natural ventricular contraction, or for the timer to reach `0`. If a natural ventricular contraction occurs, the controller skips pacing of the ventricle. On the other hand, if *TimerV* reaches `0` first, then the controller goes to state *PaceV*, which causes a contraction in the ventricle by setting `pv=1`. The controller then returns to the atrial states.

Most modern pacemakers can have the timer parameters programmed wirelessly through radio signals so that doctors can try different treatments without having to surgically remove, program, and reimplant the pacemaker.

This example demonstrates the usefulness of FSMs in describing a controller's behavior. Real pacemakers have controllers with tens or even hundreds of states to deal with various details that we left out of the example for simplicity.

With the advent of low-power microprocessors, a trend in pacemaker design is that of implementing the FSM on a microprocessor rather than with a custom sequential circuit. Microprocessor implementation yields the advantage of easy reprogramming of the FSM, expanding the range of treatments with which a doctor can experiment.

▶ 3.9 CHAPTER SUMMARY

Section 3.1 introduced the concept of sequential circuits, namely circuits that store bits. Such circuits thus have memory, and the current values in such memory is known as state. Section 3.2 developed a series of increasingly robust bit storage blocks, including the SR latch, D latch, D flip-flop, and finally a register, which can store multiple bits. The section also introduced the concept of a clock, which synchronizes the loads of registers. Section 3.3 introduced finite-state machines (FSMs) for capturing the desired behavior of a sequential circuit, and a standard architecture able to implement FSMs, with an FSM implemented using the architecture known as a controller. Section 3.4 then described a process for converting an FSM to a controller implementation. Section 3.5 described several timing issues related to the use of flip-flops, including setup time, hold time, and metastability, and the use of synchronizer flip-flops to reduce metastability problems. The section introduced asynchronous clear and set inputs to flip-flops, and described their use for initializing an FSM to its initial state. The section described the problem of output glitches and the use of registered outputs to eliminate the problem. Section 3.8 highlighted a cardiac pacemaker and illustrated the use of an FSM to describe the pacemaker's behavior.

Designing a combinational circuit begins by capturing the desired circuit behavior using either an equation or a truth table, and then following a several step process to convert the behavior to a combinational circuit. Designing a sequential circuit begins by capturing the desired circuit behavior as an FSM, and then following a several-step process to convert the behavior to a circuit consisting of a register and a combinational circuit, which together are known as a controller. Thus, conceptually, the knowledge in Chapters 2 and 3 can be used to build any digital circuit. However, many digital circuits deal with input data that are many bits wide, such as two 32-bit inputs representing two binary numbers. Imagine how complex the equations, truth tables, or FSMs would be if they involved two 32-bit inputs. Fortunately, components have been developed specifically to deal with data inputs and that therefore simplify the design process—components that will be described in the next chapter.