ECE120: Introduction to Computer Engineering

Notes Set 3.2 Finite State Machine Design Examples, Part I

This set of notes uses a series of examples to illustrate design principles for the implementation of finite state machines (FSMs) using digital logic. We begin with an overview of the design process for a digital FSM, from the development of an abstract model through the implementation of functions for the next-state variables and output signals. Our first few examples cover only the concrete aspects: we implement several counters, which illustrate the basic process of translating a concrete and complete state transition diagram into an implementation based on flip-flops and logic gates. We next consider a counter with a number of states that is not a power of two, with which we illustrate the need for FSM initialization.

We then consider the design process as a whole through a more general example of a counter with multiple inputs to control its behavior. We work from an abstract model down to an implementation, illustrating how semantic knowledge from the abstract model can be used to simplify the implementation. Finally, we illustrate how the choice of representation for the FSM's internal state affects the complexity of the implementation. Fortunately, designs that are more intuitive and easier for humans to understand also typically make the best designs in terms of other metrics, such as logic complexity.

3.2.1 Steps in the Design Process

Before we begin exploring designs, let's talk briefly about the general approach that we take when designing an FSM. We follow a six-step process:

1. develop an abstract model
2. specify I/O behavior
3. complete the specification
4. choose a state representation
5. calculate logic expressions
6. implement with flip-flops and gates

In Step 1, we translate our description in human language into a model with states and desired behavior. At this stage, we simply try to capture the intent of the description and are not particularly thorough nor exact.

Step 2 begins to formalize the model, starting with its input and output behavior. If we eventually plan to develop an implementation of our FSM as a digital system (which is not the only choice, of course!), all input and output must consist of bits. Often, input and/or output specifications may need to match other digital systems to which we plan to connect our FSM. In fact, *most problems in developing large digital systems today arise because of incompatibilities when composing two or more separately designed pieces* (or **modules**) into an integrated system.

Once we know the I/O behavior for our FSM, in Step 3 we start to make any implicit assumptions clear and to make any other decisions necessary to the design. Occasionally, we may choose to leave something undecided in the hope of simplifying the design with "don't care" entries in the logic formulation.

In Step 4, we select an internal representation for the bits necessary to encode the state of our FSM. In practice, for small designs, this representation can be selected by a computer in such a way as to optimize the implementation. However, for large designs, such as the LC-3 instruction set architecture that we study later in this class, humans do most of the work by hand. In the later examples in this set of notes, we show how even a small design can leverage meaningful information from the design when selecting the representation, leading to an implementation that is simpler and is easier to build correctly. We also show how one can use abstraction to simplify an implementation.

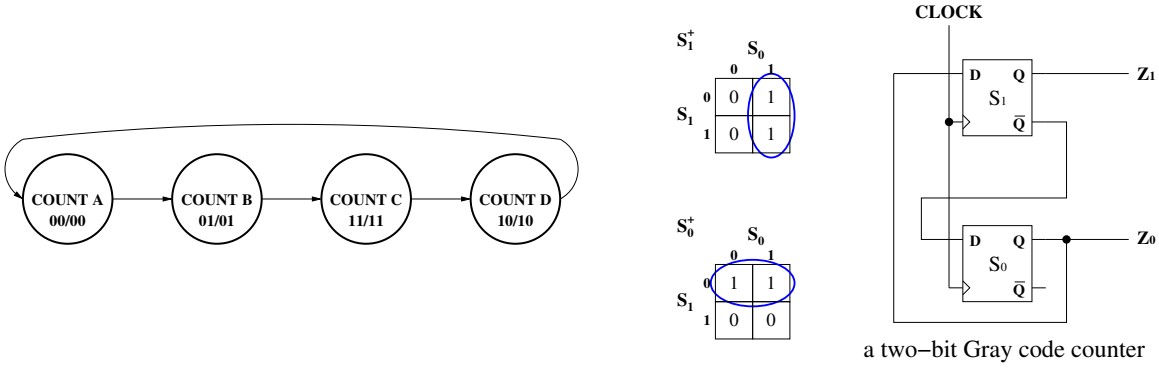
By Step 5, our design is a complete specification in terms of bits, and we need merely derive logic expressions for the next-state variables and the output signals. This process is no different than for combinational logic, and should already be fairly familiar to you.

Finally, in Step 6, we translate our logic expressions into gates and use flip-flops (or registers) to hold the internal state bits of the FSM. In later notes, we use more complex building blocks when implementing an FSM, building up abstractions in order to simplify the design process in much the same way that we have shown for combinational logic.

3.2.2 Example: A Two-Bit Gray Code Counter

Let's begin with a two-bit Gray code counter with no inputs. As we mentioned in Notes Set 2.1, a Gray code is a cycle over all bit patterns of a certain length in which consecutive patterns differ in exactly one bit. For simplicity, our first few examples are based on counters and use the internal state of the FSM as the output values. You should already know how to design combinational logic for the outputs if it were necessary. The inputs to a counter, if any, are typically limited to functions such as starting and stopping the counter, controlling the counting direction, and resetting the counter to a particular state.

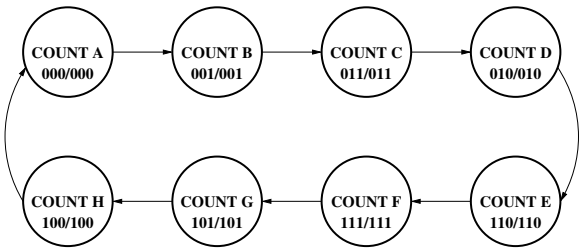
A fully-specified transition diagram for a two-bit Gray code counter appears below. With no inputs, the states simply form a loop, with the counter moving from one state to the next each cycle. Each state in the diagram is marked with the internal state value S_1S_0 (before the "/") and the output Z_1Z_0 (after the "/"), which are always equal for this counter. Based on the transition diagram, we can fill in the K-maps for the next-state values S_1^+ and S_0^+ as shown to the right of the transition diagram, then derive algebraic expressions in the usual way to obtain $S_1^+ = S_0$ and $S_0^+ = \overline{S_1}$. We then use the next-state logic to develop the implementation shown on the far right, completing our first counter design.



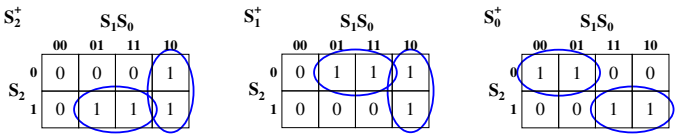
a two–bit Gray code counter

3.2.3 Example: A Three-Bit Gray Code Counter

Now we'll add a third bit to our counter, but again use a Gray code as the basis for the state sequence. A fully-specified transition diagram for such a counter appears to the right. As before, with no inputs, the states simply form a loop, with the counter moving from one state to the next each cycle. Each state in the diagram is marked with the internal state value $S_2S_1S_0$ (before "/") and the output $Z_2Z_1Z_0$ (after "/").



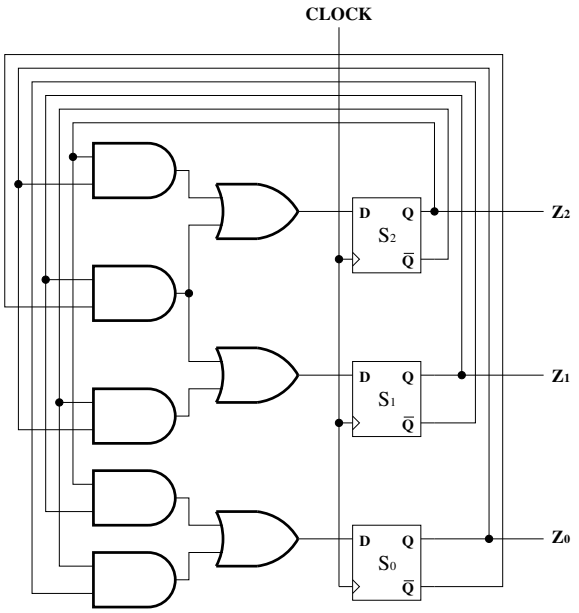
Based on the transition diagram, we can fill in the K-maps for the next-state values S_2^+, S_1^+, and S_0^+ as shown to the right, then derive algebraic expressions. The results are more complex this time.



For our next-state logic, we obtain:

$$\begin{aligned} S_2^+ &= S_2\ S_0 + S_1\ \overline{S_0} \\ S_1^+ &= \overline{S_2}\ S_0 + S_1\ \overline{S_0} \\ S_0^+ &= \overline{S_2}\ \overline{S_1} + S_2\ S_1 \end{aligned}$$

Notice that the equations for S_2^+ and S_1^+ share a common term, $S_1\overline{S_0}$. This design does not allow much choice in developing good equations for the next-state logic, but some designs may enable you to reduce the design complexity by explicitly identifying and making use of common algebraic terms and sub-expressions for different outputs. In modern design processes, identifying such opportunities is generally performed by a computer program, but it's important to understand how they arise. Note that the common term becomes a single AND gate in the implementation of our counter, as shown to the right.



a three–bit Gray code counter

Looking at the counter's implementation diagram, notice that the vertical lines carrying the current state values and their inverses back to the next state logic inputs have been carefully ordered to simplify understanding the diagram. In particular, they are ordered from left to right (on the left side of the figure) as $\overline{S_0}S_0\overline{S_1}S_1\overline{S_2}S_2$. When designing any logic diagram, be sure to make use of a reasonable order so as to make it easy for someone (including yourself!) to read and check the correctness of the logic.

3.2.4 Example: A Color Sequencer

Early graphics systems used a three-bit red-green-blue (RGB) encoding for colors. The color mapping for such a system is shown to the right.

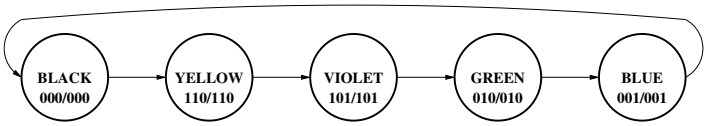
RGB	color
000	black
001	blue
010	green
011	cyan
100	red
101	violet
110	yellow
111	white

Imagine that you are charged with creating a counter to drive a light through a sequence of colors. The light takes an RGB input as just described, and the desired pattern is

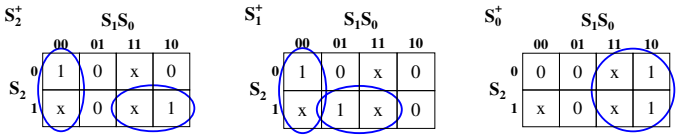
off (black) yellow violet green blue

You immediately recognize that you merely need a counter with five states. How many flip-flops will we need? At least three, since $\lceil \log_2 (5) \rceil = 3$. Given that we need three flip-flops, and that the colors we need to produce as outputs are all unique bit patterns, we can again choose to use the counter's internal state directly as our output values.

A fully-specified transition diagram for our color sequencer appears to the right. The states again form a loop, and are marked with the internal state value $S_2S_1S_0$ and the output RGB.



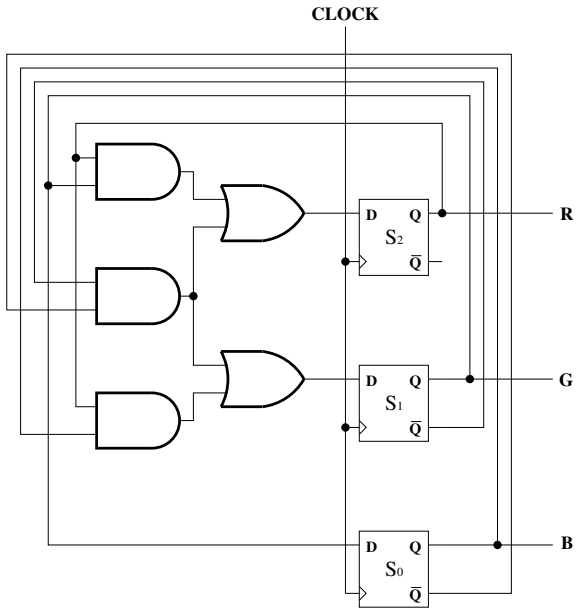
As before, we can use the transition diagram to fill in K-maps for the next-state values S_2^+, S_1^+, and S_0^+, as shown to the right. For each of the three states not included in our transition diagram, we have inserted x's into the K-maps to indicate "don't care." As you know, we can treat each x as either a 0 or a 1, whichever produces better results (where "better" usually means simpler equations). The terms that we have chosen for our algebraic equations are illustrated in the K-maps. The x's within the ellipses become 1s in the implementation, and the x's outside of the ellipses become 0s.



For our next-state logic, we obtain:

$$\begin{aligned} S_2^+ &= S_2\ S_1 + \overline{S_1}\ \overline{S_0} \\ S_1^+ &= S_2\ S_0 + \overline{S_1}\ \overline{S_0} \\ S_0^+ &= S_1 \end{aligned}$$

Again our equations for S_2^+ and S_1^+ share a common term, which becomes a single AND gate in the implementation shown to the right.



an RGB color sequencer

3.2.5 Identifying an Initial State

Let's say that you go the lab and build the implementation above, hook it up to the light, and turn it on. Does it work? Sometimes. Sometimes it works perfectly, but sometimes the light glows cyan or red briefly first. At other times, the light is an unchanging white.

What could be going wrong?

Let's try to understand. We begin by deriving K-maps for the implementation, as shown to the right. In these K-maps, each of the x's in our design has been replaced by either a 0 or a 1. These entries are highlighted with green italics.

S_2^+

S_2 \ S_1S_0	00	01	11	10
0	1	0	*0*	0
1	*1*	0	*1*	1

S_1^+

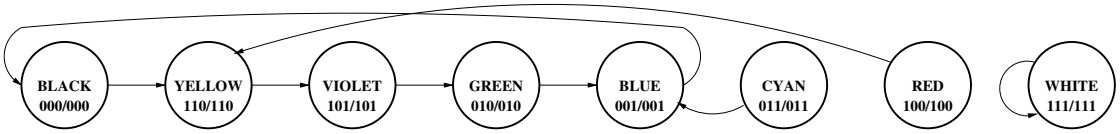
S_2 \ S_1S_0	00	01	11	10
0	1	0	*0*	0
1	*1*	1	*1*	0

S_0^+

S_2 \ S_1S_0	00	01	11	10
0	0	0	*1*	1
1	*0*	0	*1*	1

Now let's imagine what might happen if somehow our FSM got into the $S_2S_1S_0 = 111$ state. In such a state, the light would appear white, since $RGB = S_2S_1S_0 = 111$. What happens in the next cycle? Plugging into the equations or looking into the K-maps gives (of course) the same answer: the next state is the $S_2^+ S_1^+ S_0^+ = 111$ state. In other words, the light stays white indefinitely! As an exercise, you should check what happens if the light is red or cyan.

We can extend the transition diagram that we developed for our design with the extra states possible in the implementation, as shown below. As with the five states in the design, the extra states are named with the color of light that they produce.

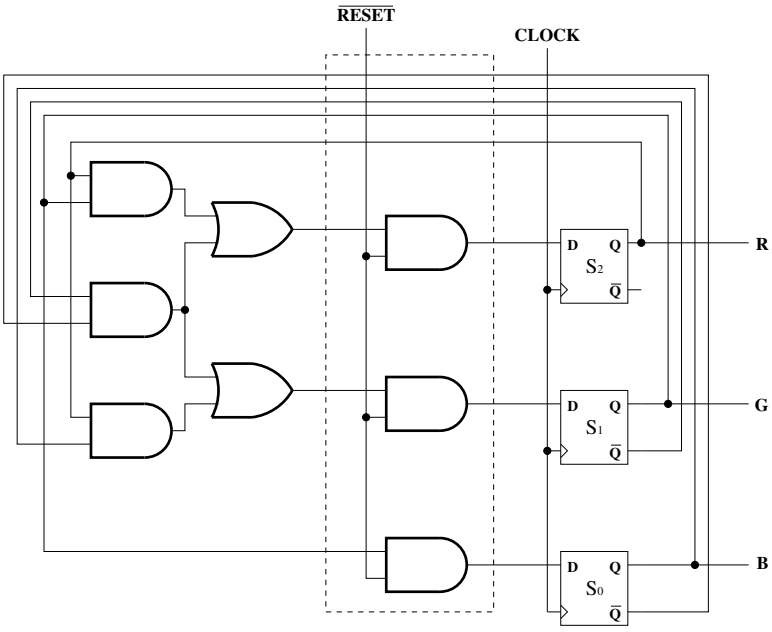


Notice that the FSM does not move out of the WHITE state (ever). You may at this point wonder whether more careful decisions in selecting our next-state expressions might address this issue. To some extent, yes. For example, if we replace the S_2S_1 term in the equation for S_2^+ with $S_2\overline{S_0}$, a decision allowed by the "don't care" boxes in the K-map for our design, the resulting transition diagram does not suffer from the problem that we've found. However, even if we do change our implementation slightly, we need to address another aspect of the problem: how can the FSM ever get into the unexpected states?

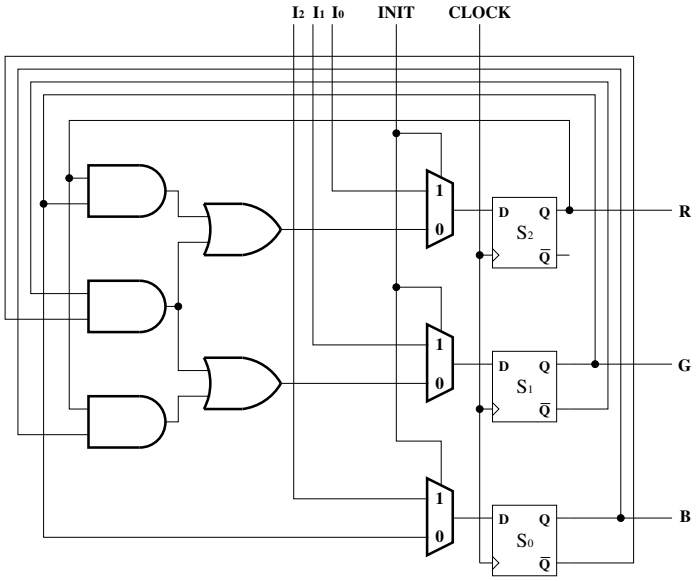
What is the initial state of the three flip-flops in our implementation? *The initial state may not even be 0s and 1s unless we have an explicit mechanism for initialization.* Initialization can work in two ways. The first approach makes use of the flip-flop design. As you know, a flip-flop is built from a pair of latches, and we can make use of the internal reset lines on these latches to force each flip-flop into the 0 state (or the 1 state) using an additional input.

Alternatively, we can add some extra logic to our design. Consider adding a few AND gates and a $\overline{RESET}$ input (active low), as shown in the dashed box in the figure to the right. In this case, when we assert $\overline{RESET}$ by setting it to 0, the FSM moves to state 000 in the next cycle, putting it into the BLACK state. The approach taken here is for clarity; one can optimize the design, if desired. For example, we could simply connect $\overline{RESET}$ as an extra input into the three AND gates on the left rather than adding new ones, with the same effect.

We may sometimes want a more powerful initialization mechanism—one that allows us to force the FSM into any specific state in the next cycle. In such a case, we can add multiplexers to each of our flip-flop inputs, allowing us to use the $INIT$ input to choose between normal operation ($INIT = 0$) of the FSM and forcing the FSM into the next state given by $I_2I_1I_0$ (when $INIT = 1$).



an RGB color sequencer with reset



an RGB color sequencer with arbitrary initialization

3.2.6 Developing an Abstract Model

We are now ready to discuss the design process for an FSM from start to finish. For this first abstract FSM example, we build upon something that we have already seen: a two-bit Gray code counter. We now want a counter that allows us to start and stop the count. What is the mechanism for stopping and starting? To begin our design, we could sketch out an abstract next-state table such as the one shown to the right above. In this form of the table, the first column lists the states, while each of the other columns lists states to which the FSM transitions after a clock cycle for a particular input combination. The table contains two states, counting and halted, and specifies that the design uses two distinct buttons to move between the states.

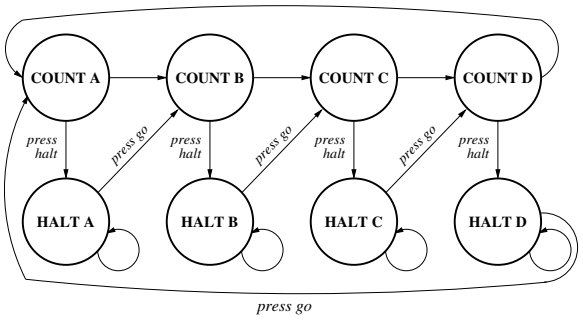
state	no input	halt button	go button
counting	counting	halted	
halted	halted		counting

A counter with a single counting state, of course, does not provide much value. We extend the table with four counting states and four halted states, as shown to the right. This version of the table also introduces more formal state names, for which these notes use all capital letters.

state	no input	halt button	go button
COUNT A	COUNT B	HALT A	
COUNT B	COUNT C	HALT B	
COUNT C	COUNT D	HALT C	
COUNT D	COUNT A	HALT D	
HALT A	HALT A		COUNT B
HALT B	HALT B		COUNT C
HALT C	HALT C		COUNT D
HALT D	HALT D		COUNT A

The upper four states represent uninterrupted counting, in which the counter cycles through these states indefinitely. A user can stop the counter in any state by pressing the "halt" button, causing the counter to retain its current value until the user presses the "go" button.

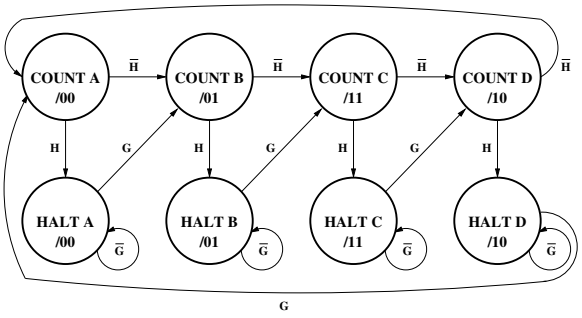
Below the state table is an abstract transition diagram, which provides exactly the same information in graphical form. Here circles represent states (as labeled) and arcs represent transitions from one state to another based on an input combination (which is used to label the arc).



We have already implicitly made a few choices about our counter design. First, the counter shown retains the current state of the system when "halt" is pressed. We could instead reset the counter state whenever it is restarted, in which case we need only five states: four for counting and one more for a halted counter. Second, we've designed the counter to stop when the user presses "halt" and to resume counting when the user presses "go." We could instead choose to delay these effects by a cycle. For example, pressing "halt" in state COUNT B could take the counter to state HALT C, and pressing "go" in state HALT C could take the system to state COUNT C. In these notes, we implement only the diagrams shown.

3.2.7 Specifying I/O Behavior

We next start to formalize our design by specifying its input and output behavior digitally. Each of the two control buttons provides a single bit of input. The "halt" button we call H, and the "go" button we call G. For the output, we use a two-bit Gray code. With these choices, we can redraw the transition diagram as show to the right.



In this figure, the states are marked with output values Z_1Z_0 and transition arcs are labeled in terms of our two input buttons, G and H. The uninterrupted counting cycle is labeled with $\overline{H}$ to indicate that it continues until we press H.

3.2.8 Completing the Specification

Now we need to think about how the system should behave if something outside of our initial expectations occurs. Having drawn out a partial transition diagram can help with this process, since we can use the diagram to systematically consider all possible input conditions from all possible states. The state table form can make the missing parts of the specification even more obvious.

For our counter, the symmetry between counting states makes the problem substantially simpler. Let's write out part of a list of states and part of a state table with one counting state and one halt state, as shown to the right. Four values of the inputs HG are possible (recall that N bits allow 2^N possible patterns). We list the columns in Gray code order, since we may want to transcribe this table into K-maps later.

state		description
first counting state	COUNT A	counting, output $Z_1Z_0 = 00$
first halted state	HALT A	halted, output $Z_1Z_0 = 00$

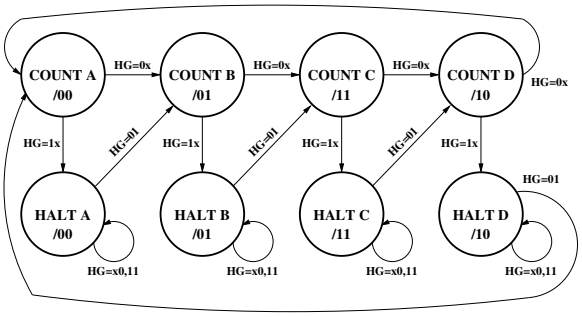
	HG			
state	00	01	11	10
COUNT A	COUNT B	unspecified	unspecified	HALT A
HALT A	HALT A	COUNT B	unspecified	unspecified

Let's start with the COUNT A state. We know that if neither button is pressed ($HG = 00$), we want the counter to move to the COUNT B state. And, if we press the "halt" button ($HG = 10$), we want the counter to move to the HALT A state. What should happen if a user presses the "go" button ($HG = 01$)? Or if the user presses both buttons ($HG = 11$)? Answering these questions is part of fully specifying our design. We can choose to leave some parts unspecified, but *any implementation of our system will imply answers*, and thus we must be careful. We choose to ignore the "go" button while counting, and to have the "halt" button override the "go" button. Thus, if $HG = 01$ when the counter is in state COUNT A, the counter moves to state COUNT B. And, if $HG = 11$, the counter moves to state HALT A.

Use of explicit bit patterns for the inputs HG may help you to check that all four possible input values are covered from each state. If you choose to use a transition diagram instead of a state table, you might even want to add four arcs from each state, each labeled with a specific value of HG. When two arcs connect the same two states, we can either use multiple labels or can indicate bits that do not matter using a **don't-care** symbol, x. For example, the arc from state COUNT A to state COUNT B could be labeled $HG = 00, 01$ or $HG = 0x$. The arc from state COUNT A to state HALT A could be labeled $HG = 10, 11$ or $HG = 1x$. We can also use logical expressions as labels, but such notation can obscure unspecified transitions.

Now consider the state HALT A. The transitions specified so far are that when we press "go" ($HG = 01$), the counter moves to the COUNT B state, and that the counter remains halted in state HALT A if no buttons are pressed ($HG = 00$). What if the "halt" button is pressed ($HG = 10$), or both buttons are pressed ($HG = 11$)? For consistency, we decide that "halt" overrides "go," but does nothing special if it alone is pressed while the counter is halted. Thus, input patterns $HG = 10$ and $HG = 11$ also take state HALT A back to itself. Here the arc could be labeled $HG = 00, 10, 11$ or, equivalently, $HG = 00, 1x$ or $HG = x0, 11$.

To complete our design, we apply the same decisions that we made for the COUNT A state to all of the other counting states, and the decisions that we made for the HALT A state to all of the other halted states. If we had chosen not to specify an answer, an implementation could produce different behavior from the different counting and/or halted states, which might confuse a user.



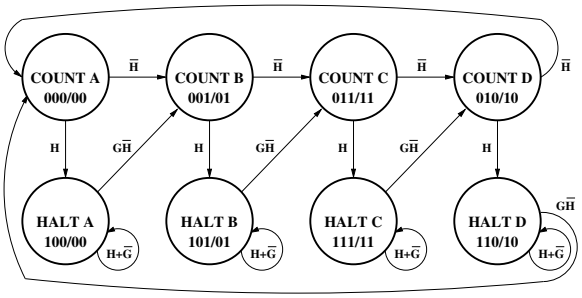
The resulting design appears to the right.

3.2.9 Choosing a State Representation

Now we need to select a representation for the states. Since our counter has eight states, we need at least three ($\lceil \log_2 (8) \rceil = 3$) state bits $S_2S_1S_0$ to keep track of the current state. As we show later, *the choice of representation for an FSM's states can dramatically affect the design complexity.* For a design as simple as our counter, you could just let a computer implement all possible representations (there aren't more than 840, if we consider simple symmetries) and select one according to whatever metrics are interesting. For bigger designs, however, the number of possibilities quickly becomes impossible to explore completely.

Fortunately, *use of abstraction in selecting a representation also tends to produce better designs* for a wide variety of metrics (such as design complexity, area, power consumption, and performance). The right strategy is thus often to start by selecting a representation that makes sense to a human, even if it requires more bits than are strictly necessary. The resulting implementation will be easier to design and to debug than an implementation in which only the global behavior has any meaning.

Let's return to our specific example, the counter. We can use one bit, S_2, to record whether or not our counter is counting ($S_2 = 0$) or halted ($S_2 = 1$). The other two bits can then record the counter state in terms of the desired output. Choosing this representation implies that only wires will be necessary to compute outputs Z_1 and Z_0 from the internal state: $Z_1 = S_1$ and $Z_0 = S_0$. The resulting design, in which states are now labeled with both internal state and outputs ($S_2S_1S_0/Z_1Z_0$) appears to the right. In this version, we have changed the arc labeling to use logical expressions, which can sometimes help us to think about the implementation.

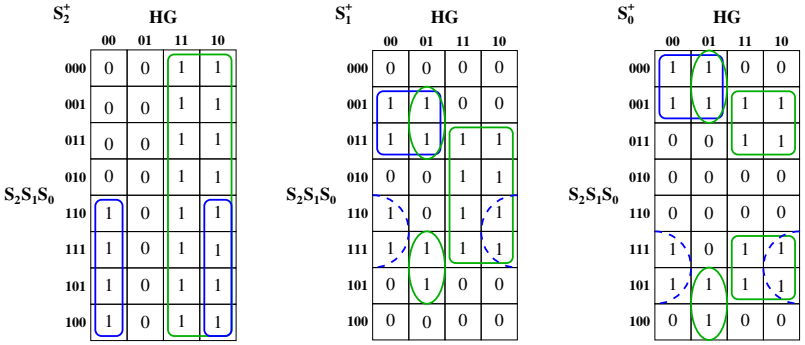


The equivalent state listing and state table appear below. We have ordered the rows of the state table in Gray code order to simplify transcription of K-maps.

state	$S_2S_1S_0$	description
COUNT A	000	counting, output $Z_1Z_0 = 00$
COUNT B	001	counting, output $Z_1Z_0 = 01$
COUNT C	011	counting, output $Z_1Z_0 = 11$
COUNT D	010	counting, output $Z_1Z_0 = 10$
HALT A	100	halted, output $Z_1Z_0 = 00$
HALT B	101	halted, output $Z_1Z_0 = 01$
HALT C	111	halted, output $Z_1Z_0 = 11$
HALT D	110	halted, output $Z_1Z_0 = 10$

		HG			
state	$S_2S_1S_0$	00	01	11	10
COUNT A	000	001	001	100	100
COUNT B	001	011	011	101	101
COUNT C	011	010	010	111	111
COUNT D	010	000	000	110	110
HALT D	110	110	000	110	110
HALT C	111	111	010	111	111
HALT B	101	101	011	101	101
HALT A	100	100	001	100	100

Having chosen a representation, we can go ahead and implement our design in the usual way. As shown to the right, K-maps for the next-state logic are complicated, since we have five variables and must consider implicants that are not contiguous in the K-maps. The S_2^+ logic is easy enough: we only need two terms, as shown.



Notice that we have used color and line style to distinguish different implicants in the K-maps. Furthermore, the symmetry of the design produces symmetry in the S_1^+ and S_0^+ formula, so we have used the same color and line style for analogous terms in these two K-maps. For S_1^+, we need four terms. The green ellipses in the $HG = 01$ column are part of the same term, as are the two halves of the dashed blue circle. In S_0^+, we still need four terms, but three of them are split into two pieces in the K-map. As you can see, the utility of the K-map is starting to break down with five variables.

3.2.10 Abstracting Design Symmetries

Rather than implementing the design as two-level logic, let's try to take advantage of our design's symmetry to further simplify the logic (we reduce gate count at the expense of longer, slower paths).

Looking back to the last transition diagram, in which the arcs were labeled with logical expressions, let's calculate an expression for when the counter should retain its current value in the next cycle. We call this variable $HOLD$. In the counting states, when $S_2 = 0$, the counter stops (moves into a halted state without changing value) when H is true. In the halted states, when $S_2 = 1$, the counter stops (stays in a halted state) when $H + \overline{G}$ is true. We can thus write

$$\begin{aligned} HOLD &= \overline{S_2} \cdot H + S_2 \cdot (H + \overline{G}) \\ HOLD &= \overline{S_2}H + S_2H + S_2\overline{G} \\ HOLD &= H + S_2\overline{G} \end{aligned}$$

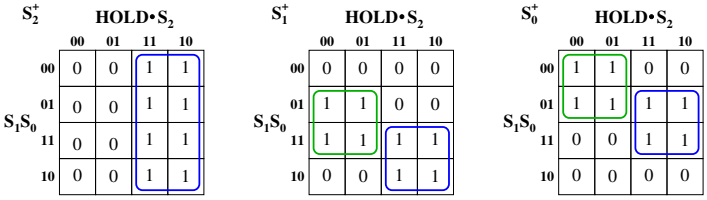
In other words, the counter should hold its current value (stop counting) if we press the "halt" button or if the counter was already halted and we didn't press the "go" button. As desired, the current value of the counter (S_1S_0) has no impact on this decision. You may have noticed that the expression we derived for $HOLD$ also matches S_2^+, the next-state value of S_2 in the K-map on the previous page.

Now let's re-write our state transition table in terms of $HOLD$. The left version uses state names for clarity; the right uses state values to help us transcribe K-maps.

		$HOLD$	
state	$S_2S_1S_0$	0	1
COUNT A	000	COUNT B	HALT A
COUNT B	001	COUNT C	HALT B
COUNT C	011	COUNT D	HALT C
COUNT D	010	COUNT A	HALT D
HALT A	100	COUNT B	HALT A
HALT B	101	COUNT C	HALT B
HALT C	111	COUNT D	HALT C
HALT D	110	COUNT A	HALT D

		$HOLD$	
state	$S_2S_1S_0$	0	1
COUNT A	000	001	100
COUNT B	001	011	101
COUNT C	011	010	111
COUNT D	010	000	110
HALT A	100	001	100
HALT B	101	011	101
HALT C	111	010	111
HALT D	110	000	110

The K-maps based on the $HOLD$ abstraction are shown to the right. As you can see, the necessary logic has been simplified substantially, requiring only two terms each for both S_1^+ and S_0^+. Writing the next-state logic algebraically, we obtain

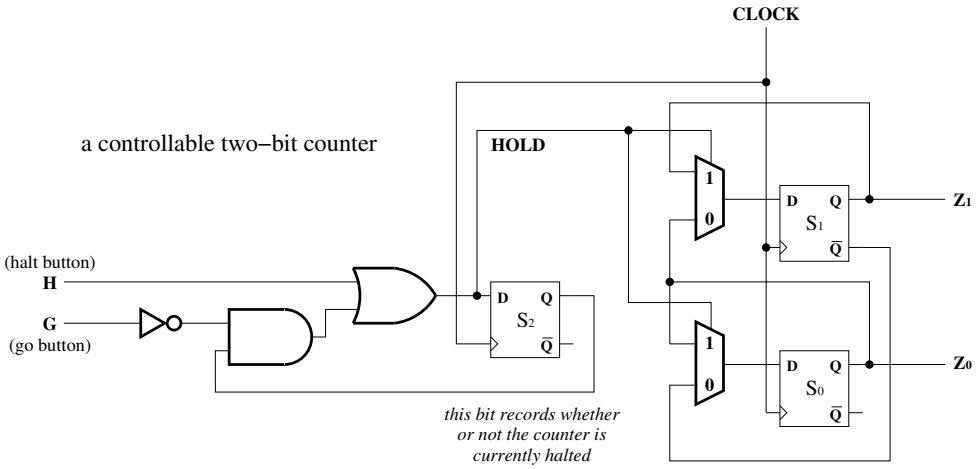


$$\begin{aligned} S_2^+ &= HOLD \\ S_1^+ &= \overline{HOLD} \cdot S_0 + HOLD \cdot S_1 \\ S_0^+ &= \overline{HOLD} \cdot \overline{S_1} + HOLD \cdot S_0 \end{aligned}$$

Notice the similarity between the equations for $S_1^+S_0^+$ and the equations for a 2-to-1 mux: when $HOLD = 1$, the counter retains its state, and when $HOLD = 0$, it counts.

An implementation appears below. By using semantic meaning in our choice of representation—in particular the use of S_2 to record whether the counter is currently halted ($S_2 = 1$) or counting ($S_2 = 0$)—we have enabled ourselves to separate out the logic for deciding whether to advance the counter fairly cleanly from the logic for advancing the counter itself. Only the $HOLD$ bit in the diagram is used to determine whether or not the counter should advance in the current cycle.

Let's check that the implementation matches our original design. Start by verifying that the $HOLD$ variable is calculated correctly, $HOLD = H + S_2\overline{G}$, then look back at the K-map for S_2^+ in the low-level design to verify that the expression we used does indeed match.



Next, check the mux abstraction. When $HOLD = 1$, the next-state logic for S_1^+ and S_0^+ reduces to $S_1^+ = S_1$ and $S_0^+ = S_0$; in other words, the counter stops counting and simply stays in its current state. When $HOLD = 0$, these equations become $S_1^+ = S_0$ and $S_0^+ = \overline{S_1}$, which produces the repeating sequence for S_1S_0 of 00, 01, 11, 10, as desired. You may want to look back at our two-bit Gray code counter design to compare the next-state equations.

We can now verify that the implementation produces the correct transition behavior. In the counting states, $S_2 = 0$, and the $HOLD$ value simplifies to $HOLD = H$. Until we push the "halt" button, S_2 remains 0, and and the counter continues to count in the correct sequence. When $H = 1$, $HOLD = 1$, and the counter stops at its current value ($S_2^+S_1^+S_0^+ = 1S_1S_0$, which is shorthand for $S_2^+ = 1$, $S_1^+ = S_1$, and $S_0^+ = S_0$).

In any of the halted states, $S_2 = 1$, and we can reduce $HOLD$ to $HOLD = H + \overline{G}$. Here, so long as we press the "halt" button or do not press the "go" button, the counter stays in its current state, because $HOLD = 1$. If we release "halt" and press "go," we have $HOLD = 0$, and the counter resumes counting ($S_2^+S_1^+S_0^+ = 0S_0\overline{S_1}$, which is shorthand for $S_2^+ = 0$, $S_1^+ = S_0$, and $S_0^+ = \overline{S_1}$). We have now verified the implementation.

What if you wanted to build a three-bit Gray code counter with the same controls for starting and stopping? You could go back to basics and struggle with six-variable K-maps. Or you could simply copy the $HOLD$ mechanism from the two-bit design above, insert muxes between the next state logic and the flip-flops of the three-bit Gray code counter that we designed earlier, and control the muxes with the $HOLD$ bit. Abstraction is a powerful tool.

3.2.11 Impact of the State Representation

What happens if we choose a bad representation? For the same FSM—the two-bit Gray code counter with start and stop inputs—the table below shows a poorly chosen mapping from states to internal state representation. Below the table is a diagram of an implementation using that representation. Verifying that the implementation's behavior is correct is left as an exercise for the determined reader.

state	$S_2S_1S_0$
COUNT A	000
COUNT B	101
COUNT C	011
COUNT D	010

state	$S_2S_1S_0$
HALT A	111
HALT B	110
HALT C	100
HALT D	001