

## LEARNING THE ROPES

Ingo Cyliax

### State Machines

In this installment, Ingo illustrates state machines using algorithmic state machine charts, implementing them through one-hot state encoding, the implementation of choice for FPGAs. Ingo takes us through the basic design methodology, using a simple design with two states. Here, the hardware has to depend on internal state, instead of just computing an output value.



This month, I'd like to look at state machine design in FPGAs. Up to now, I've been writing about design elements that can be used in what is called the data path of a design. Typically, a design is broken up into a data path (elements like register banks, adders, multipliers, etc.) and a control section. Some designs don't have much of a control part. This might be something similar to a DSP element like a FIR filter, where a result on each clock cycle is computed and output.

However, more interesting designs usually have a more complex control part. Some examples are processors, algorithms that are implemented directly in hardware, or protocol controllers. Here the hardware may have to react to inputs and depend on internal state, rather than just compute an output value.

Some of you have probably studied state machines and their designs. You may have drawn state charts and computed next state equations. Writing down specifications, like in the form of a state chart, is good design practice,

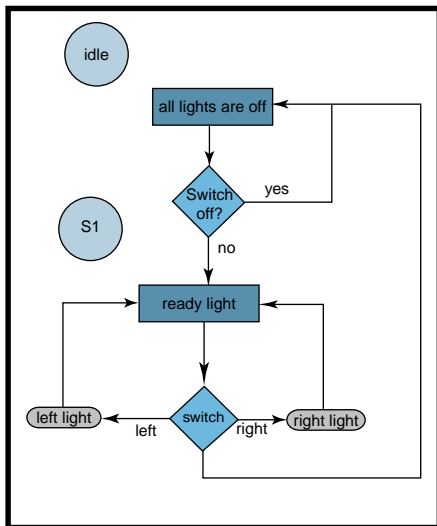
especially for state machine designs. They tend to be too complex for designers who start drawing schematics too early, usually designing themselves into a corner.

This month, I'm going to show you how to spec a state machine using algorithmic state machine (ASM) charts, and implement them efficiently using one-hot state encoding, which is the state machine implementation of choice for FPGAs.

ASMs were developed by T.E. Osborne and first described in C.R. Clare's book, *Designing Logic Systems Using State Machines*. [1] ASMs are good to use because you can express both Mealy and Moore state machines, and they can be abstract. The latter is important in the beginning of the design process when you're not yet sure what the implementation will look like. Although I'm going to describe synchronous state machines, ASMs can be used to describe asynchronous systems and software as well.

ASMs use graphical blocks to describe the design. There are three types of blocks I will use. One is the state block, which is a rectangle and denotes a particular state in the system. Each state block has a label next to it that describes the name of the state. This can be an abstract name at first, and later, encoding can be associated with this state label. The contents of the state box describe what happens when the machine enters that state. These are known as the unconditional outputs and describe the behavior of a Moore machine.

Another block is a diamond-shaped box. This box represents a conditional branch or test. It is used to indicate what input variables and conditions are consulted to compute the next state or a conditional output. A branch block can have multiple output graphs, each of which has a label or expression



**Figure 1**—Here you see an Algorithmic State Machine (ASM) chart that shows a simple machine. All of the outputs and inputs are symbolic at this point.

that has to be true for that branch to be taken.

Finally, there are conditional outputs. These are ovals that follow a branch to indicate one of several possible output conditions that can happen in the same state. This is used to describe what Mealy machines do.

So, you can see that ASMs can be used to describe Mealy and Moore state machines or a mix of both. Initially, the descriptions in the boxes can be abstract. In early phases of the design, these may be just phrases, like “turn on the motor” or “check operator input.” As the design is refined into an implementation, these labels become more concrete and represent signals and variables.

## AN ASM EXAMPLE

Let’s look at an ASM example so you can get a feel for it. Figure 1 shows a simple state machine. It has an input that is either left, right, or off. The first state of the machine is an idle state. The machine will cycle in this state while the input is off. After the input goes to left or right, the ready indicator will come on and either the left or right indicator will turn on (depending on the input) and stay on until turned off.

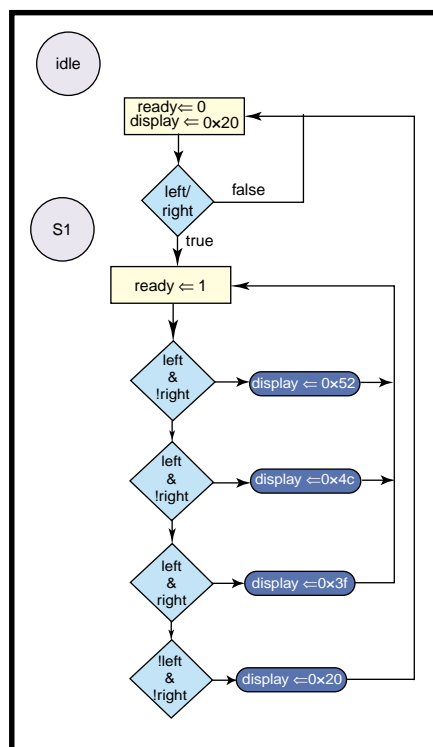
This example illustrates that you can write down an ASM chart from a verbal description. Notice that I have not specified any encoding because I

don’t really know yet what the architecture is like. At this point, you could write a small program from this ASM to model or simulate your machine before you have any ideas about the architecture.

## INPUT AND OUTPUT SIGNALS

Let’s look at implementing this machine. For this you need to know how the display and input signals are implemented. Let’s assume that the inputs come from two input signals, left and right. If left is high, the left button is activated; if right is high, the right button is activated. The display, however, is more complicated. There is an alphanumeric display that takes an ASCII input. On this, you want to display “R” for right and “L” for left. If neither is selected, you want the display to be blank, which is a space character. The display is driven with a 7-bit bus signal called *disp[6:0]*. There is also an LED, which has a “ready” signal attached.

I have changed the ASM to indicate the encoding for all signals used in



**Figure 2**—In this ASM, I have assigned specific encodings for the inputs and outputs. The inputs are two signals (right and left), and there are several outputs signals. Ready is a single output, and display is a 7-bit signal that carries ASCII representation of “R”, “L”, and “?”.

Figure 2. While changing the ASM to include decoding the input signals and encoding the output signals, I noticed that there could be a condition in which both the left and right buttons were on. I decided to handle this by displaying a “?” character in the display.

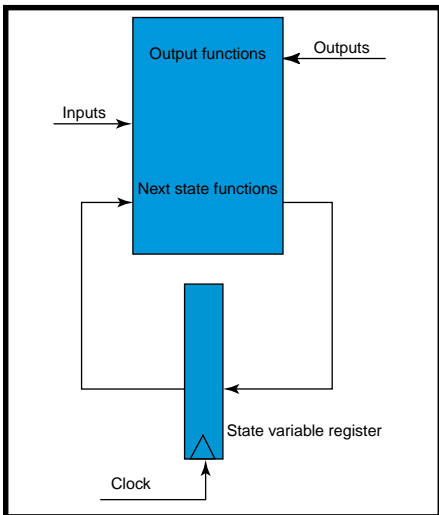
This discrepancy only came up because I used the particular signal encoding. If I had used discrete LED lights for left and right, rather than a ASCII display, it might have been all right to turn both lights on under those conditions. However, with the ASCII display, I would have ended up with a jumbled display because it would have been the logic OR of ASCII “R” and “L.”

## IMPLEMENTATION

At this point I have tied down the input and output signals, but haven’t decided how to implement the actual state machine. Traditionally, there are several ways you can implement state machines. State machines are usually like the one shown in Figure 3. There is a state variable, which is implemented as registers. There is a next function generator, which will look at the current state of the machine, evaluate some inputs, and compute what the next state should be. And, there is an output function generator, which computes the output based on the current state and input signals.

The function generators can be look-up tables, logic gates to generate the output function and next state function. The state variable contains some kind of encoding of the states in the state machine. One type of encoding would be to assign a sequential binary number to all of the states in the machine and use the binary number as the representation in the state variable register. The advantage of doing this is that you can represent a large number of states with  $\log_2(n)$  register bits. The disadvantage is that, for some state transitions, you may have a large number of bits changing in the state variable register.

Generally, you would encode the states in such a way that only a few bits change in the register. This reduces power consumption and makes



**Figure 3**—A typical state machine has a state variable register and logic or look-up tables that compute the next state as well as outputs based on the current state and inputs from the outside.

the next state equations smaller because, on average, only a few bits are affected by a state change. Encoding methods such as gray codes are a favorite for these types of state machines.

### ONE-HOT ENCODING

Taken to the extreme, you can devise an encoding scheme that has one bit per state. This type of encoding is called one-hot because only one state variable bit is set, or “hot,” for each state. The benefit is that the next state generation function is simple. Also, because only two bits change per transition, power consumption is small. The drawback is that it takes a lot of register bits, one per state. However, all of these are a real advantage in FPGAs, because FPGAs are register rich.

Typically, there is a register after each LUT. For many state machines, the next state generator for a state bit will fit in a single LUT. Also, because next state functions tend to be local in scope, the routing between state variables is minimal when expressed as one-hot. For most state machines, one-hot is the way to go in FPGAs.

Also, one-hots are easy to implement in schematics. You can almost see the state machine representation jump out at you because each state has its own flip-flop. So, let’s check out how to implement the machine in a one-hot encoding.

Figure 4 shows the finished ASM. It is much like the ASM in Figure 3, but I have added the state encoding in its proper place, above and to the right of the state box. There are two state variables, but only one can be on at a time. However, a binary encoded state machine could get away with a single register because it’s enough to hold two states.

### SYNTHESIZE

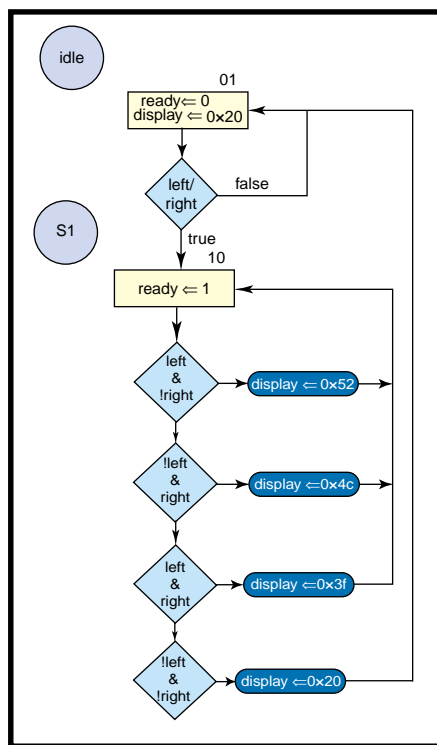
Now it’s time to synthesize the state machine, which means deriving the equations for the state variables and outputs. There are two state variables in this design, which I’ll call *idle* and *s1*. The next state equations for both of these are:

$$\text{idle} = (\text{idle} \mid \text{s1}) \ \& \ (\text{!right} \ \& \ \text{!left})$$

$$\text{s1} = (\text{idle} \mid \text{s1}) \ \& \ \text{right} \ \& \ \text{left}$$

OK, that was simple. Let’s derive the equations for the outputs of your machine. First, the unconditional ready output:

$$\text{ready} = \text{s1}$$



**Figure 4**—In this ASM, I have assigned the actual state encoding as the top right of each state box. I’m using one-hot encoding so there are two state variables, and only one of them can be on at any given time.

Left right s1 idle disp			
x	x	01	0100000
0	0	10	0100000
0	1	10	1001100
1	0	10	1010010
1	1	10	1111111

**Table 1**—This is the truth table for the ASCII display output generator. From this, you can arrive at the logic to implement it.

then the *disp* outputs, which are a bit harder. A truth table helps here (see Table 1).

From which, you get:

$$\text{disp}[0] = \text{s1} \ \& \ \text{left} \ \& \ \text{right}$$

$$\text{disp}[1] = \text{s1} \ \& \ \text{left}$$

$$\text{disp}[2] = \text{s1} \ \& \ \text{right}$$

$$\text{disp}[3] = \text{s1} \ \& \ \text{right}$$

$$\text{disp}[4] = \text{s1} \ \& \ \text{left}$$

$$\text{disp}[5] = (\text{s1} \ \& \ \text{!(left} \ \wedge \ \text{right)}) \ \& \ \text{idle}$$

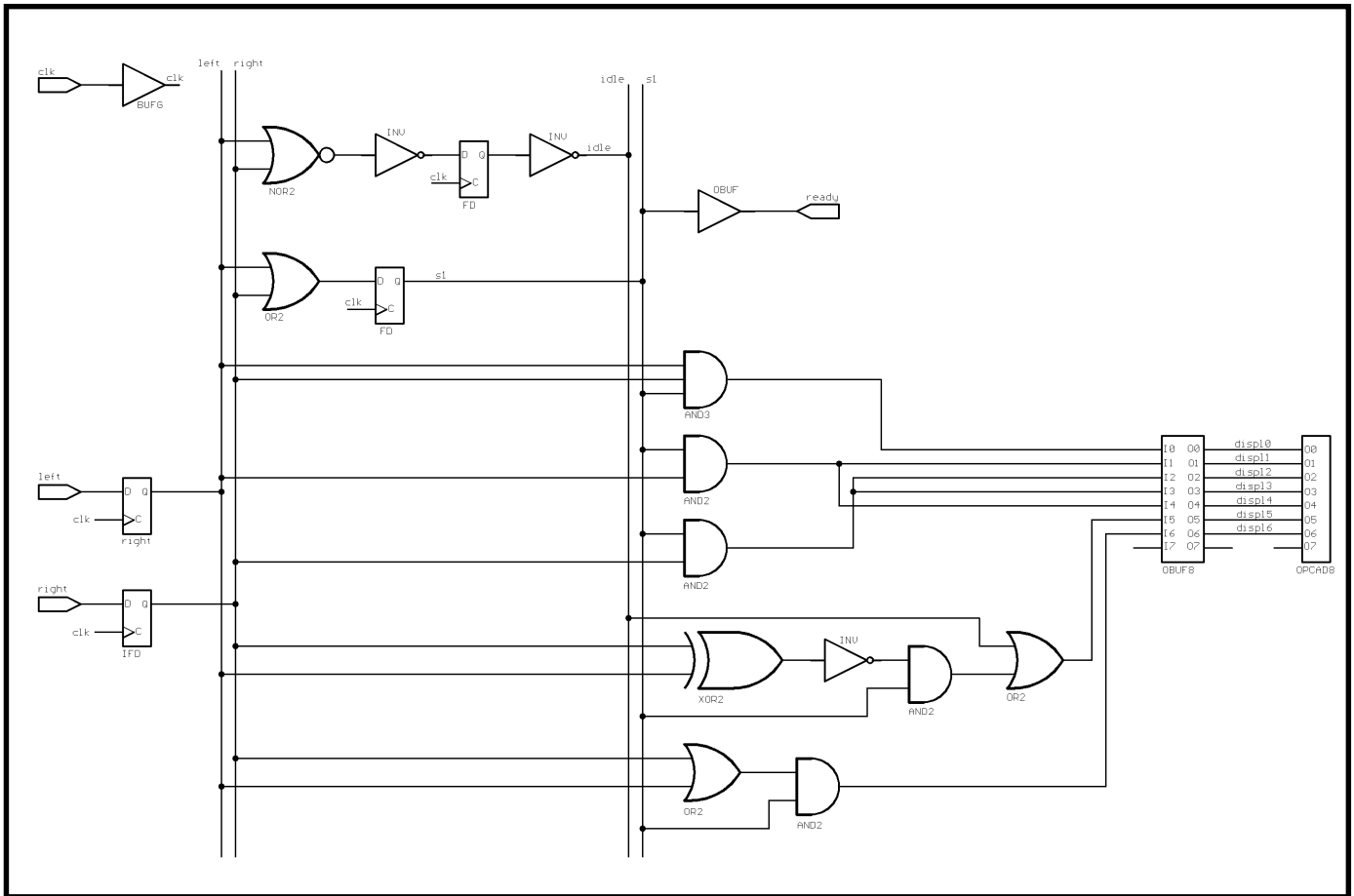
$$\text{disp}[6] = \text{s1} \ \& \ (\text{left} \ \& \ \text{right})$$

Figure 5 shows a schematic of the finished design. The inputs are synchronized using a flip-flop. There are a couple of issues with one-hot state machines that I will point out in this schematic. Consider this, if each state variable bit represents a state, what happens when all of the register bits are at zero? This happens when the machine is reset. You have to find a way to initialize a one-hot state machine.

### INITIALIZING

One way to initialize a one-hot state machine is to use a preset signal on the state that represents the starting state of your machine, while the rest of the state bits use reset signals. This works, of course, but it won’t if none of the signals reset the machine. For example, an FPGA will power up with all of the registers set to zero. You’d have to provide an external reset signal to get things started. Obviously this is not a robust design.

Another strategy is to recognize the



**Figure 5**—Here is a finished schematic of this sample state machine. Each flip-flop is a state variable. The logic in front of it computes the next state, and the logic on the right computes the outputs. The idle signal is internally inverted in the flip-flop, so when the machine is reset, the idle state will be the start state of the machine.

null state and cause a transition to the start state. This scheme will cause a glitch in the first clock tick when the state is still null. You also have to make sure that the outputs are decoded properly for the null state. It's not a clean design.

A better solution is to invert the input and output sense of the register that represents the start state. This way, if all of the registers get reset to a zero value, the encoding for the start state will be true (i.e., zero means true). This is why there are inverters on one of the registers in the schematic, representing the idle state.

By the way, inverters are free in FPGAs; they get absorbed into the equations used.

## SYNCHRONOUS DESIGN RULES

There are some rules you will need to follow in order to reduce the number of pitfalls you'll encounter when designing synchronous state machines

or other synchronous circuits.

First, use only synchronous inputs to control state transitions. By having inputs flip-flop depending on asynchronous signals, you're asking for trouble. This introduces the chance for metastability and manifests itself in unpredictable ways.

The design might work for days in the lab or even a long time in the field, but eventually the odds won't be in your favor, and it will fail. And worse, the situation is not repeatable. It's easy to synchronize an input by adding a flip-flop to it. There are input flip-flops in almost all of the FPGA technologies available today. By adding the flip-flop to an asynchronous signal, you'll make sure that whatever metastable event happens to the input register, it will have settled before reaching the state machine next state generator.

Next, use a single clock per domain. Each signal that has to traverse clock domains is an asynchronous signal. It

helps to move modules in different clock domains to different schematic pages or HDL modules. If possible, don't design with multiple clock domains.

Also, don't gate the clock. Adding a gate to the clock signal will add clock skew. Many FPGAs have flip-flops with clock enable inputs that are implemented so as not to add latency to the clock signal. Use the architectural clock enable, and make sure it is driven by a synchronous signal. Basically, the clock should go straight to all of the clocked elements in a single clock domain. Use global (low skew) clock buffers to distribute the clock in an FPGA. And, don't use the clock signal itself in any of your logic.

Finally, use top down design for designing state machines. This means you should work out the design specification, separate the architecture (data path) from the control, assign the encoding of the signals, and then opti-

mize and derive the equations for the implementation.

## UNTIL NEXT TIME

Granted, this design is simple. It only has two states and doesn't do anything interesting. It does, however, illustrate the basic design methodology that can be used to design state machines in FPGAs using one-hot state machine encoding and ASMs.

Next time, I'll look at a more complex design that uses a data path in addition to a control. ☒

*Ingo Cyliax is the Sr. Hardware Engineer at Derivation Systems Inc. (DSI) where he designs and builds embedded systems and hardware components. DSI is the leader in formally synthesized FPGA cores and specializes in embedded Java technology. Ingo has been writing on various topics ranging from real-time operating systems to nuts and bolts hardware issues for several years.*

## REFERENCE

- [1] C.R. Clare, *Designing Logic Systems Using State Machines*, McGraw-Hill, New York, NY, 1973.

## RESOURCE

- F. Prosser and D. Winkel, *Art of Digital Design*, Prentice Hall, Englewood Cliffs, NJ, 1986.

Circuit Cellar, the Magazine for Computer Applications. Reprinted by permission. For subscription information, call (860) 875-2199, [subscribe@circuitcellar.com](mailto:subscribe@circuitcellar.com) or [www.circuitcellar.com/subscribe.htm](http://www.circuitcellar.com/subscribe.htm).