How to Implement State Machines in Your FPGA

by Adam Taylor
Principal Engineer
EADS Astrium
aptaylor@theiet.org
State machines are often the backbone of FPGA development. Choosing the right architecture and implementation methods will ensure that you obtain an optimal solution.

FPGAs are often called upon to perform sequence- and control-based actions such as implementing a simple communication protocol. For a designer, the best way to address these actions and sequences is by using a state machine. State machines are logical constructs that transition among a finite number of states. A state machine will be in only one state at a particular point in time. It will, however, move between states depending upon a number of triggers.

Theoretically, state machines are divided into two basic classes—Moore and Mealy—that differ only in how they generate the state machine outputs. In a Moore type, the state machine outputs are a function of the present state only. A classic example is a counter. In a Mealy state machine, the outputs are a function of the present state and inputs. A classic example is the Richards controller (see http://en.wikipedia.org/wiki/Richards_controller).

DEFINING A STATE MACHINE
When faced with the task of defining a state machine, the first step is to develop a state diagram. A state diagram shows the states, the transitions between states and the outputs from the state machine. Figure 1 shows two state diagrams, one for a Moore state machine (left) and the other for a Mealy state machine.

If you had to implement these diagrams in physical components (as engineers did before FPGAs were available), you would start by generating the present-state and next-state tables, and producing the logic required to implement the state machine. However, as we will be using an FPGA for implementation, we can work directly from the state transition diagram.

ALGORITHMIC STATE DIAGRAMS
While many state machines are designed using the state diagram approach shown in Figure 1, another method of describing a state machine’s behavior is the algorithmic state chart. This ASM chart (Figure 2) is much closer in appearance to a software-engineering flow chart. It consists of three basic constructs:

1. **State box**, which is associated with the state name and contains a list of the state outputs (Moore)
2. **Decision box**, which tests for a condition being true and allows the next state to be determined
3. **Conditional output box**, which allows the state machine to describe Mealy outputs dependent upon the current state and inputs

Some engineers feel that a state machine described in ASM format is easier to map to implementation in a hardware description language such as VHDL.

XPLANATION: FPGA 101
State machines are often the backbone of FPGA development. Choosing the right architecture and implementation methods will ensure that you obtain an optimal solution.
MOORE VS. MEALY: WHICH SHOULD I CHOOSE?

The decision to implement a Moore or a Mealy machine will depend upon the function the state machine is required to perform along with any specified reaction times. The major difference between the two is in how the state machines react to inputs. A Moore machine will always have a delay of one clock cycle between an input and the appropriate output being set. This means a Moore machine is incapable of reacting immediately to a change of input, as can be seen clearly in Figure 3. This ability of the Mealy machine to react immediately to the inputs often means that Mealy machines require fewer states to implement the same function as a Moore implementation would need. However, the drawback of a Mealy machine is that when communicating with another state machine, there is the danger of race conditions, which occur when the output is unexpectedly and critically dependent on the sequence or timing of other events.

Of course, a state machine does not have to be just Moore or Mealy. It is possible to create a hybrid state machine that uses both styles to deliver a more efficient implementation of the function required. For example, a state machine that is used to receive RS232 serial data could be a hybrid.

IMPLEMENTING THE STATE MACHINE

Using a high-level language like VHDL, it is easy to implement a state machine directly from the state diagram. VHDL supports enumerated types, allowing you to define the actual state names. Here's an example:

```vhdl
TYPE state IS (idle, led_on, led_off);
```
The above type declaration corresponds to the state diagrams shown in Figure 1, for a machine that toggles a light-emitting diode on and off when a button is pushed.

There are many methods of implementing a state machine. However, the choices basically fall into two categories. The first basic technique uses a one-shot approach that contains everything within a single process. The alternative is a two-process approach, which separates the combinational and sequential logic.

As a general rule, most engineers prefer to implement a single-process state machine. These have several benefits over the traditionally taught two-process approach:

- They remove the risk of incomplete coverage of signals in the combinational process, creating latches.
- State machine outputs are aligned to the clock.
- They are generally easier to debug than a two-process implementation.

Regardless of which approach you decide upon to implement your state machine, you will evaluate the next-state determination and any outputs using a CASE statement, as seen in Figure 4. The figure shows a side-by-side comparison between a Moore (left) and Mealy machine using a single-process approach.

**STATE MACHINE ENCODING**

The state variable is stored within flip-flops, which are updated with the next state on the next clock edge (even if there is no state change). How you use these flip-flops to represent the state value depends upon the number of states and on whether you choose to direct your synthesis tool in one particular method. The three most common types of state encoding are:

- **Sequential**—State encoding follows a traditional binary sequence for the states.
- **Gray**—Similar to the sequential encoding scheme except that the state encoding uses Gray code, where only one bit changes between state encodings.
- **One-Hot**—This technique allots one flip-flop for each state within the state machine. Only one flip-flop is currently set high and the rest are low; hence the name “one-hot.”

Both sequential and Gray encoding schemes will require a number of flip-flops, which you can determine by the equation below.

\[
\text{FlipFlops} = \text{Ceil} \left( \frac{\text{Log10}(\text{States})}{\text{Log10}(2)} \right)
\]

By contrast, one-hot encoding schemes require the same number of states as there are flip-flops.

The automatic assignment of state encoding depends upon the number of states the state machine contains. While this will vary depending upon the synthesis tool you have selected, you can use this general rule of thumb for encoding:

- Sequential, less than five states
- One-hot, five to 50 states
- Gray, greater than 50 states

Figure 3 – Screens show simulation results for Moore (top) and Mealy outputs.
In addition, you can also use the syn_encoding attribute to define the values of the state encoding directly. For example, suppose you desired to encode a three-state machine using the following state encoding: Idle = "11," led_on = "10," led_off = "01," as opposed to the more traditional sequence of "00," "01" and "10."

```vhdl
TYPE state IS (idle, led_on, led_off) ;
SIGNAL current_state : state := idle;
ATTRIBUTE syn_encoding STRING;
ATTRIBUTE syn_encoding OF current_state :
SIGNAL IS "sequential";
```

Often you will not necessarily think about what state encoding to use, instead allowing the synthesis engine to determine the correct implementation—getting involved only if the chosen style causes an issue. However, should you need to take things into your own hands and define the state encoding, there is no need to do so long-hand, defining constants for each state with the state encoding. Instead, you can use an attribute within the code to drive the synthesis tool to choose a particular encoding style, as demonstrated below.

```vhdl
TYPE state IS (idle, led_on, led_off) ;
SIGNAL current_state : state := idle;
ATTRIBUTE syn_encoding STRING;
ATTRIBUTE syn_encoding OF current_state :
SIGNAL IS "sequential";

```

where “sequential” can be also “gray” and “onehot.” You can also combine these three choices—sequential, Gray and one-hot—with the “safe” attribute to ensure the state machine can recover to a valid state should it enter an illegal state.

```
```

In addition, you can also use the syn_encoding attribute to define the values of the state encoding directly. For example, suppose you desired to encode a three-state machine using the following state encoding: Idle = “11,” led_on = “10,” led_off = “01,” as opposed to the more traditional sequence of “00,” “01” and “10.”

```vhdl
TYPE state IS (idle, led_on, led_off) ;
SIGNAL current_state : state := idle;
ATTRIBUTE syn_encoding STRING;
ATTRIBUTE syn_encoding OF current_state :
SIGNAL IS “sequential”;
```

As the engineer, you are responsible for using the correct settings in the synthesis tool to ensure the tool does not ignore any attributes. For example, the Xilinx® XST tool requires that you set the FSM Option to USER while Synopsys’ Synplify requires that the FSM Compiler be turned off.

The equation given earlier determines the number of flip-flops needed for a state machine implementation. Since not all state machines are to a power of two, this means that
some states will not be used within the design. As the engineer implementing the state machine, you must be responsible for ensuring these unused states are correctly handled within the design. There are several basic techniques to implement to accomplish this goal that will serve a broad range of designs, and other, more advanced techniques to employ if you are working in high-reliability, safety-critical fields. (See Xcell Journal issue 73 for an in-depth article titled “Using FPGA in Mission-Critical Systems,” which looks at state machine protection.)

However, for most applications you will simply need to ensure your state machine correctly addresses unused states and recovers properly should it enter an illegal state. There are two main options for doing so. The first method is to implement a safe state machine using the synthesis tool. The tool will typically insert additional logic to detect an illegal state and return the state machine to a valid state. The second option, which gives you more control over the logic being implemented, is to declare all $2^n$ states and use another attribute to ensure they are not optimized out despite having no entry condition. This means the state is never entered by any condition within the state machine except an error (single-event upset, etc.). The code below shows the use of attributes to prevent removal of these unused states.

```vhdl
TYPE state IS (idle, led_on, led_off) ;
SIGNAL current_state : state := idle;
ATTRIBUTE syn_keep BOOLEAN;
ATTRIBUTE syn_keep OF current_state : SIGNAL IS TRUE ;
```

In short, safe, efficient state machine design is a key skill for every engineer working with FPGAs. The choice among Moore, Mealy or even mixed machines depends upon the needs of your overall system. Whichever type of state machine you select, understanding the tools and techniques available for implementation will ensure you achieve the optimal solution.

---

**Techway**

*The way of innovation*

**Versatile FPGA Platform**

**KINTEX-7**

- PCI-e 4x Gen2
- Kintex-7 Series
- SDK for Windows and Linux
- Ready to go 10 GbE on FMC slot!

A cost-effective solution for intensive calculations and high speed communications.

www.techway.eu

---