A systematic approach to verifying FSMs

Shaker Sarwary - October 27, 2005

Large SOCs (systems on chips) comprise many FSMs (finite-state machines), which combine with datapaths, memories, and other components. Although FSMs are among the most abundant components on chips, many designers lack an understanding of their role and their impact on the quality and validation of the final chip implementation. FSMs are sources of functional bugs in SOCs. Designers often attribute poor timing, power, and performance to poor FSM design; therefore, FSM verifiability and FSM-design-style considerations are important aspects of SOC design. Although verification tools can perform checks on FSMs, such as deadlock and unreachable states, these tools may not consider the aspects of FSM-design styles (Figure 1). This article takes a predictive look at FSMs and discusses functional and implementation issues resulting from various FSM-design styles. It also provides guidelines for designing and implementing FSMs to meet design goals (references 1 and 2).

The most common functional problems in FSMs include unreachable or deadlocked states, dead transitions, constant registers, and asynchronous inputs. An unreachable state is one that the FSM cannot reach from any initial state, and it can indicate a functional problem in the design or result from an unintentional dead transition in the FSM. In any case, an unreachable state indicates redundant logic in the design that the designer should clean up. A dead transition is a transition that the FSM can never exercise; it also indicates redundancy in a design.

A deadlock state is a state from which the FSM cannot move to another reachable state. Similarly, an n-deadlock state is a set of n states in which all transitions from any of these states reach states only within the set. It is also possible that a transition occurs out of a state only for a finite number of times. This situation means that the FSM will eventually lock to this state. This type of deadlock state—"eventually deadlock"—may remain unidentified during simple verification and simulation but can cause serious failures on-chip. Figure 2 shows an example of an eventually deadlocked state.

Another common condition occurs when an FSM register becomes stuck at a constant value of zero or one that is caused by FSM encoding, and it may indicate a wrong or suboptimal encoding. An asynchronous input is an FSM input coming from a clock domain other than the one controlling the state registers. This condition may arise from properly built, functional handshaking circuitry. However, designers should review any asynchronous inputs to ensure that an unsynchronized input does not cause any clock-crossing issues.

FSM metrics

In addition to these problems, the FSM-design style may impact the verifiability of an FSM. Designers should analyze metrics to determine whether a given simulated or formal validation flow can verify the FSM. Designers can use FSM attributes, such as the number and depth of states, transitions, inputs, and outputs, to qualify an FSM from an implementation and verification point of
view. These attributes can have a major impact on the quality of implementation of an FSM. Encoding styles, output, and next-state-logic descriptions, as well as the presence and description of the initial state, can affect the implementation and verification of an FSM.

The number of states is the first indicator of an FSM’s complexity. A larger set of states can make the FSM code difficult to read, difficult to understand, and likely to contain functional bugs. A large number of states can complicate the FSM encoding, and it can contribute to extra combinational and sequential logic that incurs timing, area, and power penalties. Large FSMs make the validation effort more challenging and may lead to low verification coverage. Decomposing larger FSMs into smaller, communicating FSMs helps in all these areas. Keeping the number of states lower than 20 is a good FSM-coding practice.

The number of transitions in an FSM is another indicator of the FSM’s complexity. Complex transitions can lead to complex next-state combinational logic, which can directly impact the timing. Similar to FSMs with a large number of states, an FSM with a large number of transitions increases the designer’s effort to maintain, enhance, and verify that FSM. Reducing the number of transitions may require rebuilding a module with new control circuitry, but this technique results in more modular code that is easier to maintain and enhance. Typically, two transitions per state should be sufficient to express common functions in terms of an FSM. For a 20-state FSM, 40 transitions would occur.

The number of inputs to or outputs from an FSM impacts the complexity of the output and the next-state logic. As a rule of thumb, the number of inputs to an FSM should not exceed the number of states in an FSM. Typically, the number of outputs from an FSM should be small. For example, 25 outputs would help control complexity.

The depth of the FSM refers to the number of states on the longest path from an initial state without visiting the same state twice. Deep FSMs may be harder to verify, especially when they have a high number of states and transitions, which makes the FSM wide and deep. The FSM depth should be kept at less than 15 to enable a simpler verification and debugging process.

**Styles**

The encoding style for an FSM has a high impact on the quality of the FSM implementation in timing, area, power, and other attributes. Some of the most common encoding styles include one-hot encoding, Gray encoding, and minimum encoding. The one-hot-encoding style means that only one bit of the state registers can be at one for any given state of the FSM. This encoding may result in a higher number of registers in the final implementation. The combinational-logic complexity resulting from one-hot encoding may be comparable or slightly more than other encoding styles, but the combinational logic controlling the data input of each state register may be smaller, which may contribute to better timing performance. A designer may choose this encoding style for its simplicity and its timing performance.

Gray encoding is a special state-encoding style in which source and destination states for any transition in an FSM have only one different bit in their encodings. Because a single register can change value during any transition, Gray encoding is the best choice for lower power consumption. The number of registers for encoding the states may be small, but the combinational-logic complexity for next-state and output generation is unpredictable. It may be difficult to find an optimal Gray-encoding scheme for large FSMs with many transitions.

Minimum encoding results in the minimum number of state registers, but the control logic may be more complex. A designer can minimize the combinational logic with carefully assigned encoding.
Depending on the level of optimization in the encoding, this encoding style may result in good or poor timing and area performance. Unless a designer deploys an optimized encoding scheme, this encoding style may be a poor choice. Use it if the total number of registers in a design is a concern, but don't use it if power is the main concern.

Designers may want to consider custom encoding as a variation of the minimum-encoding style, which adds extra bits or registers to accomplish power, area, or timing goals. It does not have to follow the one-hot- or Gray-encoding style, and designers should not use it unless the project's performance requirements justify its use.

Moore and Mealey are two FSM styles operating from an output perspective. The output logic of a Moore FSM is only a function of the state registers, whereas the output logic of a Mealey machine is a function of FSM inputs and state registers. It is safer to design a Moore machine. If a design requires a Mealey machine, the designer must thoroughly validate the output logic and the higher level module using the FSM. It is important to watch for any failures resulting from asynchronous behavior of the FSM outputs.

**Next-state logic**

Another way to look at FSMs is from a next-state-logic perspective. In simple FSMs, each state is a macro that represents the encoding of the FSM, and each assignment to a next-state variable is a direct assignment to these macros. When a designer directly assigns the next-state variable to a state macro, the logic involved in the next-state calculation comes from conditional statements preceding the assignment and the state's encoding. For code clarity, use simple assignments for next-state variables in an FSM.

However, in some cases, the next state may not use a simple assignment, such as when an FSM requires a complex next-state scheme requiring heavy logic computation. In these cases, the FSM may compute the next state in a separate function or task to simplify or modularize the code. In some cases, a designer may use the function to compute FSM outputs. These extensions indicate FSM complexity, and designers should avoid them whenever possible by decomposing and simplifying the FSM.

The FSM may also compute the next state by using arithmetic or logic operators. The most common arithmetic operation is assigning the next state as the current state incremented by a constant value. This next-state-computation scheme hides complexity. There may be more states than items in the case statement defining the FSM because each next-state calculation may engender a new state that the items lack. This next-state-computation scheme becomes even more complex when it appears in the default item of the case statement.

An FSM may have one or more initial states. As a general rule, an FSM should have an initial state, and the designer should integrate a reset signal into the FSM description to initialize the FSM. An FSM without a standard initial state is susceptible to functional problems and can introduce extra difficulty in analysis, verification, and maintenance.

**Validation**

Given how FSM features can impact the verification and implementation of a design, it is important to automate the validation methodology so that systematic controls enforce an FSM-design style for a project. The validation tools must automatically detect functional issues in FSMs. The tool should check not only coding styles and syntax, but also potential functional problems, such as deadlock.
states. Once the tool identifies a bug or problem area, a comprehensive debugging environment needs to perform a root-cause analysis. This debugging environment should support a clear messaging mechanism, RTL-code back annotation, schematic highlighting, an FSM-bubble-diagram viewer, and a waveform viewer. All of these components must tightly integrate to facilitate debugging and the analysis for FSM-related bugs. The tool should present implementation issues in a succinct chart of easy analysis and an FSM-quality audit. This report should link to the debugging environment for easy access and modification of the associated RTL code, as well as FSM exploration.

FSMs are among the biggest sources of bugs and problems in a design. Designers should carefully analyze the metrics to determine the quality of an FSM with regard to its impact on final implementation and verifiability. For a given project, a designer may choose one or multiple FSM styles to fit the requirements of a project. Additionally, for various FSM attributes, such as the number of states and the number of transitions, the designer should implement an upper limit that supports the project requirement and enables easier verifiability of each FSM.

These requirements define an FSM-design methodology that tools should automatically enforce to remove human errors and reduce the design time, building time, and time to market. By thoroughly understanding the types of FSMs and when to use different styles, designers can develop better FSMs. To avoid costly re-spins, it’s important to follow the creation guidelines and look for a good FSM-validation approach.

References