

Chapter 2

Post-silicon Debugging of a Single Building Block

Abstract In this chapter, we analyze the factors that complicate the post-silicon debugging of a single SOC building block. In Sect. 2.1, we first introduce a formal finite state machine (FSM) description, to capture the cycle-accurate behavior of a single building block. To debug the silicon implementation of the corresponding building block, we can subsequently use the debug process described in Sect. 1.3 to compare its behavior to the behavior that is captured by this description. In doing so, we identify however six factors in Sect. 2.2 that complicate (the application of) this process. We conclude this chapter with a summary in Sect. 2.3.

2.1 Behavior of a Single Building Block

2.1.1 Formal Definitions

Design teams commonly use an FSM description to specify the cycle-accurate behavior of an SOC building block [2, 3]. A block diagram of an FSM is shown in Fig. 2.1. The FSM in Fig. 2.1 has an input I , an internal state s , and an output O . The input I takes a value from the set \mathcal{I} with a possible input symbols. The state takes its value from the set \mathcal{S} with b possible states. The output O takes its value from the set \mathcal{O} with c possible output symbols. The current state s of the FSM is stored in a register, which is triggered by the input clock signal “clk”. The content of this register is set to the output of the combinational logic block δ , whenever a rising edge occurs on this clock signal. We indicate this *synchronous behavior* with a triangular symbol on the clock input of all registers in this book. We refer to the rising edges on the clock signal that cause this update of the FSM state as the *active edges* for these registers. A silicon implementation of this FSM stores its state in *state elements*, such as *flip-flops* and *random accessible memory (RAM)*. The combinational logic block λ determines the new value for the output O , based on the current state s and optionally the momentary value of the input I . We formally define our FSM as a 6-tuple $\mathcal{M} = \{\mathcal{S}, s^0, \mathcal{I}, \mathcal{O}, \delta, \lambda\}$, in which:

- \mathcal{S} is the *set of possible states*; $\mathcal{S} = \{s^0, \dots, s^{b-1}\}$.
- s^0 is the *initial state*; $s^0 \in \mathcal{S}$.
- \mathcal{I} is the *input alphabet*, i.e., the set of input symbols; $\mathcal{I} = \{i^0, \dots, i^{a-1}\}$.
- \mathcal{O} is the *output alphabet*, i.e., the set of output symbols; $\mathcal{O} = \{o^0, \dots, o^{c-1}\}$.

Fig. 2.1 Example reference for a building block

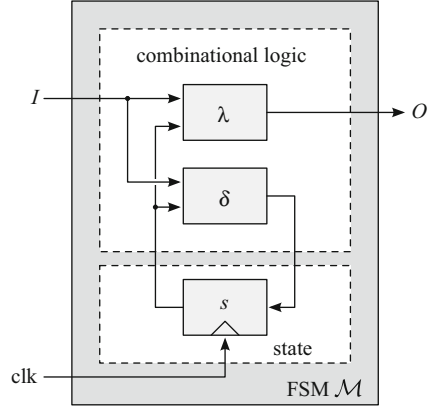


Table 2.1 Finite state machine specification

s	I	$\delta(s, I)$	$\lambda(s, I)$
s^0	0	s^0	0
s^0	1	s^1	0
s^1	0	s^0	0
s^1	1	s^2	0
s^2	0	s^0	0
s^2	1	s^3	1
s^3	0	s^0	1
s^3	1	s^0	1

- Function δ is the *FSM next-state function* that specifies the next state of the FSM, based on the current state s and the current value of the input I ; $\delta : \mathcal{S} \times \mathcal{I} \rightarrow \mathcal{S}$. Note that the definition of this function makes our FSM *deterministic*, as this function is a *total function*, i.e., defined for all its arguments, and maps every possible input combination to exactly one next state value.
- Function λ is the *FSM output function* that specifies the new value for the output O of the FSM, based on the current state s and optionally the current value of the input I ; $\lambda : \mathcal{S} \times \mathcal{I} \rightarrow \mathcal{O}$. Note that this function can represent the output behavior of both Mealy [2] and Moore FSMs [3], depending on whether the input signals respectively directly influence the outputs or only through a change in the internal state.

Table 2.1 shows the details of an example FSM. This FSM has four internal states, i.e., $\mathcal{S} = \{s^0, s^1, s^2, s^3\}$, an input alphabet $\mathcal{I} = \{0, 1\}$, and an output alphabet $\mathcal{O} = \{0, 1\}$. The FSM next-state function δ and FSM output function λ are specified in Table 2.1. While the input symbol “1” is applied to the FSM’s input, this FSM transitions through these four states on successive active clock edges. The FSM transitions from any state to its initial state on the next active clock edge when the input symbol “0” is applied.

An FSM has an associated *state transition graph (STG)* \mathcal{G} (refer to Fig. 2.2 for our example FSM). An STG is a labeled and directed graph $\mathcal{G} = (\mathcal{V}, \mathcal{E})$, where

Fig. 2.2 Corresponding STG of the FSM in Table 2.1

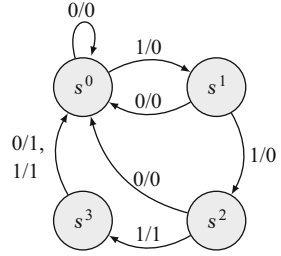
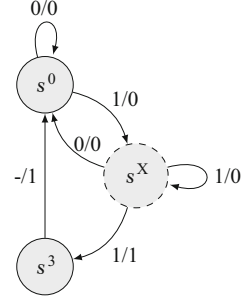


Fig. 2.3 Example FSM with a super state s^X



each vertex $v \in \mathcal{V}$ corresponds to a state $s = f_s(v)$, $f_s : \mathcal{V} \rightarrow \mathcal{S}$, and vice versa, $v = f_v(s)$, $f_v : \mathcal{S} \rightarrow \mathcal{V}$. Each edge $e_{ij} = (v, w)$, $e_{ij} \in \mathcal{E}$ and $v, w \in \mathcal{V}$, corresponds to a transition from state $s^i = f_s(v)$ to state $s^j = f_s(w)$. This edge e_{ij} has a set of labels $L_{ij} = L(e_{ij})$, $L : \mathcal{V}^2 \rightarrow 2^{\mathcal{I} \times \mathcal{O}}$. Each label $l \in L_{ij}$ is a pair of symbols (x, y) , $x \in \mathcal{I}$ and $y \in \mathcal{O}$. The input symbol x causes a transition from state s^i to state s^j , i.e., $s^j = \delta(s^i, x)$. The output symbol y corresponds to the resulting new value for the output O , i.e., $y = \lambda(s^i, x)$ [1].

In this book, we use a *super state* s^X as a compact notation for a set of FSM states. We use this notation later in this book to prevent having to draw very large and complex STGs. When we say that an FSM is in a super state s^X , we mean that its current state s is a member of the super state s^X . Our notation for a super state in an STG is a dashed circle. We only show the transitions between the super state and the states of the FSM that are not included in the super state. We do not draw the transitions between the states in the super state. As an example, the super state s^X shown in Fig. 2.3 is a compact notation for the states s^1 and s^2 that are shown in Fig. 2.2. Figure 2.3 is a behaviorally-equivalent abstraction of Fig. 2.2. Please note further that although multiple edges in the STG in Fig. 2.3 share the same input symbol, this does not indicate that the corresponding FSM is *non-deterministic*. For these transitions, both the input symbol of the FSM and the specific state inside each super-state determine which edge is taken.

We furthermore define a *stall state* using Eq. 2.1.

$$s \in \mathcal{S} \text{ is a stall state} \Leftrightarrow \exists x \in \mathcal{I}. \delta(s, x) = s \quad (2.1)$$

A stall state is a state of the FSMs, where the execution of the FSM can be temporarily suspended under external control. A stall state has a *self edge* in the associated STG. For example, s^0 in Fig. 2.2 is a stall state. *Stalling* an FSM means setting its inputs to the condition on one of its self edges, thereby forcing it to stay in the associated state once it has reached this state. We discuss stall states and their use in the communication between building blocks in Chap. 3.

We furthermore define a *reset input symbol* using Eq. 2.2.

$$x \in \mathcal{I} \text{ is a reset input symbol} \Leftrightarrow \forall s \in \mathcal{S}. \delta(s, x) = s^0 \quad (2.2)$$

A reset input symbol is an input symbol of the FSM, which, when it is applied to the input of the FSM, causes the state of the FSM to be unconditionally set to the initial state s^0 , on the next active edge of the clock. The input symbol “0” in our example FSM is a reset input symbol, as the FSM transitions from every state to the initial state s^0 on the next active edge of the clock, whenever this symbol is applied to the input of the FSM.

2.1.2 Execution Behavior

We can obtain the expected behavior for a building block from its reference FSM \mathcal{M} , when we are given an input function $x(t) : \mathbb{R}_0^+ \rightarrow \mathcal{I}$, and a clock period $T \in \mathbb{R}^+$ and a clock phase $\phi \in \mathbb{R}_0^+$ for the input clock signal. The *clock period* is defined as the amount of time between two adjacent active edges on the clock signal. The *clock phase* is defined as the amount of time between a reference moment in time $t = 0$ and the first active edge on the clock signal. The n^{th} active edge on a clock signal is placed at a time $t_{ae}(n), n \in \mathbb{N}^+$, which is given by Eq. 2.3.

$$t_{ae}(n) = (n - 1)T + \phi, n \in \mathbb{N}^+ \quad (2.3)$$

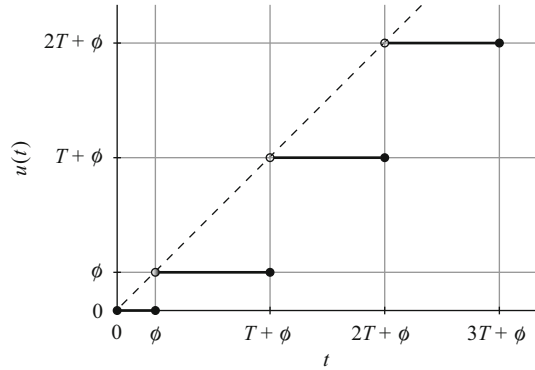
Note that this function places the first active edge at $t = \phi$. We define the *current state function* $s(t)$ of an FSM \mathcal{M} in Eq. 2.4, $s(t) : \mathbb{R}_0^+ \rightarrow \mathcal{S}$.

$$s(t) = \begin{cases} s^0 & \text{for } 0 \leq t \leq \phi \\ \delta(s(u(t)), x(u(t))) & \text{for } t > \phi \end{cases} \quad (2.4)$$

The initial state of the FSM is assumed to be s^0 up to and including the moment in time of the first active edge ($t = \phi$). We examine the validity of this assumption in Sect. 2.2.2. To define the state $s(t)$ for $t > \phi$, we use the time-quantization function $u(t) : \mathbb{R}_0^+ \rightarrow \mathbb{R}_0^+$ to determine the location in time of the closest preceding active edge. The function $u(t)$ is defined in Eq. 2.5 and shown graphically in Fig. 2.4.

$$u(t) = \begin{cases} 0 & \text{for } 0 \leq t \leq \phi \\ (\lceil \frac{t-\phi}{T} \rceil - 1)T + \phi & \text{for } t > \phi \end{cases} \quad (2.5)$$

Fig. 2.4 The time-quantization function $u(t)$



The dashed line in Fig. 2.4 shows the function $v(t) = t$. Note how the function $u(t)$ stays below the function $v(t)$ and yields the time of the closest preceding active edge for each time t . To calculate the next state value at each active edge, the current state of the FSM is combined with the momentary values on the input I of the FSM at the time of the last active edge, given by $x(u(t))$. These two values are subsequently used as the input parameters for the FSM next-state function δ to obtain the new state. This new state is valid for the period $(u(t) + T, u(t) + 2T]$. We define the output function $y(t) : \mathbb{R}_0^+ \rightarrow \mathcal{O}$ of FSM \mathcal{M} using Eq. 2.6.

$$y(t) = \lambda(s(t), x(t)) \quad (2.6)$$

Equation 2.6 uses the FSM output function λ to specify the new output value based on the current state of the FSM at time t , defined by $s(t)$, and the momentary value on the input of the FSM I at time t , given by $x(t)$.

Note how the functions $s(t)$ and $u(t)$ associate respectively a current state and an output value with the FSM for each absolute point in time $t \in \mathbb{R}_0^+$.

2.2 Complicating Factors for Debugging

The FSM description, formalized in Sect. 2.1.1, can be used as a reference while debugging its silicon implementation using the debug process described in Sect. 1.3. In this section we identify six factors that complicate (the application of) this process.

2.2.1 Limited Observability and Controllability

The high integration level of modern SOC's results in a very large state space. The state of the SOC also changes very rapidly over time. The amount of functional data that is therefore generated inside an SOC is multiple orders of magnitude larger than

the amount of data that can be stored on-chip or streamed off-chip in real-time. This severely limits the spatial scope of a debug experiment.

Furthermore, the state of a building block can only be controlled via its inputs. A debug engineer may apply a reset input symbol to start the execution of a building block from its initial state s^0 . It may be difficult, if not impossible, to change the temporal scope between debug experiments and start the execution of a building block from any other state. It may therefore be difficult to control the clock signal and inputs of a silicon implementation of a building block and observe its state and outputs using only off-chip debug instruments.

The use of DfD techniques can improve this debug observability and controllability. The cost of the silicon area associated with these debug instruments has to be carefully weighed against their potential benefits to keep the resulting SOC competitive. As it is not clear in advance what specific debug functionality is the best investment to be able to find the root cause of post-silicon failures, it is best to utilize a generic DfD technique that maximizes the internal observability at an acceptable cost. Based on these observations, we define complicating factors CF-1, CF-2, and CF-3 for debugging.

CF-1: Limited Spatial Observability and Controllability *The intrinsic observability and controllability of a silicon implementation for debugging is limited by the number of input and output pins it has and their maximum operating speed, and is significantly smaller than the amount of data that is generated internally.*

CF-2: Limited Temporal Controllability and Observability *It may be difficult, if not impossible, to start the execution of a silicon implementation from a state other than its initial state s^0 , and observe its internal state at every point in time during its execution.*

CF-3: Implementation Cost *An on-chip debug architecture to improve the debug controllability and observability of a silicon implementation costs implementation effort and silicon area.*

2.2.2 Undefined Substate and Outputs

The assumption that the start-up state of an implementation of a building block is the initial state s^0 (refer to Eq. 2.4) is not true for all implementations of a building block. In a silicon implementation of a building block, its state is stored in flip-flops and embedded RAMs. A single flip-flop in a silicon implementation settles to a random start-up state when its power supply is turned on. The same holds for the start-up state of individual bit-cells in a RAM. As such, when the same silicon implementation is powered up multiple times, this very likely results in different initial states. The reason why these implementations still function correctly despite their different initial states, is because they are designed to respond to the application of a reset input symbol to their input. Most digital ICs have a dedicated reset input

for this purpose. Asserting this reset input forces the state of the building block to its initial state s^0 . Comparing the behavior of two implementations before the assertion of their reset inputs can lead to the incorrect attribution of a difference in their states or outputs to a fault in the silicon implementation, while this difference is in fact caused by a valid difference in start-up state. In order to accurately compare the states, it is therefore important to first assert the reset input of both implementations at the start of each debug experiment. This should force both implementations to their initial state s^0 , after which their behaviors can be checked for errors.

Embedded RAM modules however do not have an input to reset their state. Instead, the designer of a building block that uses an internal RAM module has to guarantee by design that the assertion of the reset input of the building block, or the module this building block is a part of, initializes enough internal state and outputs to make it comparable to the state and outputs of another implementation of that building block. This means that any uninitialized substate, including the state of the embedded RAM module, is not allowed to affect the execution behavior of the implementation. Failure to do so may result in undefined behavior. Similarly, the application or non-application of a certain control sequence to the inputs of the building block may lead to the state of a building block to become undefined. For example, the state of a synchronous, dynamic random accessible memory (SDRAM) module becomes undefined when its content is not refreshed frequently enough. Based on these observations, we define complicating factor CF-4 for debugging.

CF-4: Undefined Substate and Outputs *(A part of) the state and outputs of a (silicon) implementation of a building block may be undefined in certain intervals during its execution.*

Whether a part of the state of an implementation is undefined or not can only be known to the debug engineer if another part of its state has been designed to indicate this. We can subsequently use this indicator to determine the undefined part of the state and outputs, and not use this part in the state comparison with another implementation. We discuss how to do this next.

2.2.3 State Comparison

The abstraction level of the silicon implementation and a reference for a building block may not be the same. In those cases, it is necessary to use *abstraction functions* to bridge this difference and be able to compare their states and outputs. These functions have to translate (semantically interpret) the state and output values at the abstraction level of the silicon implementation to the corresponding data values at the abstraction level of the reference. These functions depend on three aspects: (1) the structural transformations that have taken place in the implementation refinement process going from the abstraction level of the reference to the abstraction level of the silicon implementation, (2) the difference in data types used at the abstraction levels of the reference and silicon implementation, and (3) the behavior of the building block.

Consider, as an example, a building block with a reference FSM \mathcal{M} , which has eight states, i.e., $|\mathcal{S}_{\mathcal{M}}| = 8$. Its silicon implementation needs at least a three-bit state register to represent an element in this set. During debug, we need a state abstraction function, $f_{s,abstract} : \mathbb{B}^3 \rightarrow \mathcal{S}_{\mathcal{M}}$, that translates the state of the silicon implementation to a state that can be compared to the state of the reference FSM. We call an abstraction function *consistent*, when it is a *total function*. We subsequently call a state s of the silicon implementation a *consistent state*, when it is possible to translate this state to a corresponding state of the reference, i.e., when $f_{s,abstract}(s)$ is defined for this state. We call a state s of the silicon implementation *inconsistent* when this function is not defined for the state s .

Consider, for an example of an inconsistent state, another building block with a reference FSM \mathcal{M}' , which has five states, i.e., $|\mathcal{S}_{\mathcal{M}'}| = 5$. The silicon implementation of this building block still needs at least a three-bit state register to represent an element in this set. However, not all possible values of this three-bit register may represent a valid state in the reference FSM, i.e., the associated abstraction function $f'_{s,abstract} : \mathbb{B}^3 \rightarrow \mathcal{S}_{\mathcal{M}}$ may not be a *consistent function*.

Note that we similarly need an output abstraction function, $f_{o,abstract} : \mathbb{B}^x \rightarrow \mathcal{O}$, $x \in \mathbb{N}^+$ for the value on the x binary outputs of a silicon implementation, and can similarly define a *consistent output* and an *inconsistent output* of the silicon implementation.

Not being able to translate and compare the state and outputs of a silicon implementation with a reference complicates the debug process. We therefore investigated possible causes for inconsistent states during debug and found that certain SOC architectures may cause this. We discuss this in more detail in Chap. 3. Based on these observations, we define complicating factor CF-5 for debugging.

CF-5: State Comparison *The state and outputs of a (silicon) implementation of a building block may need to be translated to allow a comparison to the state and outputs of a reference.*

2.2.4 Transient Errors

Some errors may not cause a failure when they are activated, or the failure may not be visible long enough to be observed, because our temporal observability is limited (refer to complicating factor CF-2). We call an error *permanent* when, after its activation, its effects remain observable in the state or in the output values of the building block. Otherwise, the error is called *transient*. This is for example the case, when an erroneous state of a building block is corrected before the effects of the error can propagate to the output of the building block.

The problem with transient errors is that we cannot state with certainty whether a fault is present in the silicon implementation or not, when we observe that its state and outputs match those of the reference at a certain point in its execution. An error may become dormant after its activation, and before it or its effects are

observed in the debug process of Fig. 1.12. We therefore incorrectly change the temporal scope to search for the error later in time instead of earlier. We can only make a correct change in the scope when we actually observe that the state and/or the output(s) of the silicon implementation differ from those of the reference. In the presence of permanent faults, we can therefore apply more advanced and faster search algorithms, e.g., a binary search algorithm, to reduce the temporal scope than we can in the presence of transient faults. Based on these observations, we define complicating factor CF-6 for debugging.

CF-6: Transient Errors *Errors in a (silicon) implementation may be transient.*

2.3 Summary

In this chapter, we introduced a formal FSM description as a cycle-accurate reference for the behavior of a silicon implementation of a single SOC building block. We subsequently analyzed the difficulty in debugging the behavior of this silicon implementation using this reference. We identified six complicating factors that relate to the limited observability and controllability of a silicon implementation, possibly-undefined substate of an implementation after start-up and at other times during its execution, the possibility of inconsistent states, and the occurrence of transient errors.

A good silicon debug approach should (1) address the intrinsic limitations to the spatial and temporal observability and controllability of the SOC state, (2) address the spatial and temporal controllability of the SOC execution, (3) provide state and output abstraction functions to help bridge any differences in abstraction level between the SOC implementation and its reference, and filter out undefined (sub)state, and (4) do so at a low implementation cost. In particular, good temporal observability of the SOC state is required to debug transient errors.

We introduce a debug approach that meets these requirements in Chap. 4. Before then, we first investigate the factors that complicate the debugging of multiple, interacting building blocks in the next chapter, because SOC are composed of multiple building blocks.

References

1. John E. Hopcroft and Jeffrey D. Ullman. *Introduction To Automata Theory, Languages, And Computation*. Addison-Wesley Publishing Co., 1979.
2. George H. Mealy. A method for synthesizing sequential circuits. *Bell Systems Technical Journal*, 34:1045–1079, September 1955.
3. Edward F Moore. Gedanken-experiments on sequential machines. 34:129–153, 1956.

Debugging Systems-on-Chip
Communication-centric and Abstraction-based
Techniques

Vermeulen, B.; Goossens, K.

2014, XV, 311 p. 127 illus., 7 illus. in color., Hardcover

ISBN: 978-3-319-06241-9