Chapter 4. Logic Validation

Logic emulation is one of the most promising application domains for multi-FPGA systems. In this chapter we explore logic emulation in greater depth, and show how emulation fits into a complete logic validation methodology.

Logic validation is the process of determining whether a realization of a logic circuit exhibits the desired features. While this statement sounds simple, logic validation is in fact a fairly complex procedure, with many different possible approaches. The major issue is in defining what the "desired features" are for a given piece of logic. There are many different answers. One possibility is that the behavior of an implementation matches the behavior of some predefined specification. However, whether the specification actually matches the users' desires is not always clear, especially if the specification is quite complex. We could define "desired features" as simply whatever the users desire, but the users may not know exactly what they want. Thus, giving the users a system they can examine and test to help define their desires can also be an important part of logic validation. Some validation goals can be quite easy to define. These include the requirements that the system achieve a certain cycle time (performance), or that it avoid certain erroneous behavior, such as deadlock or other failure modes.

Another important facet of the logic validation issue is the definition of what is a "realization of a logic circuit". Logic validation will often be performed on a version of the logic somewhat abstracted from the final, physical realization. Early in the design cycle, all that may be available is a high-level specification of the circuit functionality. The designers may want to test this specification before implementing it. Catching errors in the specification can allow much simpler fixes, with much less wasted effort, than if the errors are allowed to propagate all the way to the completed implementation. In general, errors corrected earlier in the design cycle are much easier to fix than errors fixed later in the process, since as the design progresses the implementation becomes more detailed, and thus more complex. Even when the implementation is completed, there may still be the need to test an abstracted version of the circuit. Some logic validation techniques require that the circuit be simplified somewhat, either because the technique cannot handle all the details of the system, or because adequate performance can only be achieved by simplifying the logic. For example, software simulators can simulate most of the electrical behavior of a circuit, modeling the voltage levels on all signals in the system, but a much faster simulator can be built for pure functional testing, where signal voltage values are abstracted to purely "true" and "false" values.

As we will see in the following sections, there are a number of different methods for performing logic validation. They each have different strengths and goals, with different models of how user desires for the logic are expressed, and operate on different realizations of the circuit's functionality. As we will discuss in the final section of this chapter, the proper method for logic validation is not a single technique, but instead an integrated system of techniques that takes advantage of the differing strengths of each of these approaches.

Simulation

Probably the most powerful and most widely used logic validation method is software simulation. Here, the logic realization is executed in software. The software models whatever portions of the electrical and functional behavior of the elements in the logic realization the user desires, and the resulting behavior can be observed. For example, in a unit-delay functional simulator (i.e., a simulator where all electrical effects are ignored, the circuit is represented as gates, and all gates take exactly one time unit to react to new inputs), the software examines the inputs to all gates, and computes the appropriate values for each gate's output(s). By assigning different values to the inputs of the system, the user can see how the system will react to different situations. By applying sequences of inputs, the sequential behavior of the system can also be examined. Simulators need not only be unit-delay functional simulators, but can also incorporate more complex models of circuit timing and electrical behavior. For example, a simulator can include a detailed model of the capacitances on a signal line, and the drive strengths of the transistors generating this signal, to compute the predicted voltage levels on the wire at any given time.

The advantages of software simulation are that they have good observability and controllability, and are easy to use. Since each signal value must be maintained by the software to process the simulation, the software can tell the user the value of a wire at any time in the simulation, and can follow the changing value over time. In this way, the user gains a great deal of information about the system's behavior, information that might not be available in the final hardware realization. Also, since the values on the wires are simply numbers stored inside the computer, these values can quite easily be manipulated by the user. For example, the user may want to understand the behavior of the system once it is in some specific configuration. In the final logic implementation, it may be quite difficult to force the system into this state. The user of a software simulator can simply instruct the simulator to put the system into the desired state, and then observe the subsequent behavior. The user can also experiment with the system, evaluating different configurations and designs. This allows the designer to efficiently explore the design space, leading to better implementations. In terms of ease of use, software simulators can be constructed to accept any reasonable circuit description. Thus, simulators can execute high-level specifications, specifications that are quite distant from their eventual hardware implementations. In this way, users can begin testing their designs very early in the design cycle, thus saving significant effort by fixing bugs before they propagate to more detailed (and more time-consuming to modify) implementations. Also, portions of a given specification can be tested in isolation, making it easier to understand that subcircuit's behavior.

The flexibility of software-simulation is also its biggest weakness. It takes a significant amount of instructions for a standard processor to process the logic elements in the realization to be tested. This means that the software simulation is going to operate significantly slower than the real circuit. Thus, a second of circuit operation can take a software simulator minutes, hours, or more to process. Work has been done on providing specialized simulation-accelerators to speed up software simulation [Blank84, Zycad94b, Zycad94c]. Also, simulators can operate on abstracted versions of the logic, ignoring some details of the implementation such as detailed timing and voltage levels, thus providing faster simulation. While software accelerators and simulators that work on abstracted versions of the logic can give boosts in performance, it still takes a software simulation significantly longer to execute than a true hardware implementation of the circuit. This tends to relegate software simulation to those situations where either the time penalty is not severe, or the flexibility and observability of software simulation is required. These include testing subcircuits of complex designs and early testing of specifications, where short executions of the software can discover problems in the realizations, as well as executions of trouble situations where bugs have been discovered by other methods, where software simulation can help the designer understand the failure.

Prototyping

Prototyping is the process of creating a physical implementation of the circuit under validation. For example, one method of testing a circuit is to actually construct it, and then test this implementation under normal working conditions. Since one now has the real circuit, results obtained from testing it will be the most accurate, since there is no abstraction, modeling, or other hidden assumptions in the process. Thus exact performance, suitability, reliability, and other evaluations can be made. Not only can more accurate tests be performed, but the prototype can also be given to users for evaluation. This allows the users to determine whether the system will actually handle their needs and demands, something that is difficult to do with just the initial specification. Also, prototypes usually operate at or near target system speeds, allowing tests to be completed much faster than software simulation. Note that a prototype of the system can be made differently than the eventual production system. For example, techniques of breadboarding and wire-wrap allow a prototype to be created quickly. These techniques are simply methods to wire together standard components to implement the system. As such, they allow a prototype to easily be constructed, but the implementation technology (chips scattered across generic boards with individual wire connections) is different from the final product. This introduces some discrepancies between the prototype and the eventual system, introducing some inaccuracy in the modeling. One specific example is in performance, where wire-wrap and breadboard prototypes cannot handle cutting-edge performance, and thus such systems will need to be slowed down. Also, both wire-wrap and breadboarding are somewhat unreliable. Thus, the debugging process for these technologies is more complex, since the implementation technology must always be considered a potential culprit in any anomalous behavior.

While prototyping has the benefits of a high degree of accuracy and high-speed testing, it also suffers from some significant problems. First, to build a prototype one must have the system to be prototyped completely designed. Because of this in general one cannot prototype a specification or behavioral description. As a result prototyping is only useable fairly late in the design process. Also, constructing the prototype can be a costly process, both in terms of materials (since a complete implementation of a circuit may involve circuit board or ASIC fabrication) and in construction time. These prototypes are usually difficult to alter, so to avoid the costs of multiple fabrications prototyping is again relegated to very late in the design process, when (hopefully) errors will be few. Finally, it is hard to test a prototype, since there is usually little or no access to the internal state of the system, and it is difficult to control system behavior. Specifically, while a software simulator can display and alter the value of any signal in the system, this is not possible in most prototypes, making it much more difficult to detect and isolate erroneous behavior.

Even though prototypes are costly to create and difficult to use, they are the only truly accurate method of logic validation. All other methods of logic validation make abstractions and assumptions about the electrical and behavioral characteristics of actual circuits, making the results somewhat suspect. Thus, testing a prototype is an essential portion of any logic validation process, though it is usually relegated to quite late in the design process. Detection and correction of most bugs in the system is handled by other validation methods.

Emulation

Logic emulation is a method of logic validation that shares many of the advantages (and disadvantages) of both prototyping and software simulation. Like a prototype, the circuit to be evaluated is implemented in hardware so that it can achieve high performance test cycles. However, like software simulation, the emulation can easily be observed and altered to help isolate bugs. Logic emulation takes a gate-level description of a logic circuit and maps it onto a multi-FPGA system. This multi-FPGA system is a prefabricated, reprogrammable compute engine that can be configured to implement the desired circuit functionality in a matter of seconds. However, to transform the circuit description into a mapping suitable for this multi-FPGA system can take multiple hours to complete. This mapping process is usually completely automated by the emulator's system software. Once the circuit is mapped to the multi-FPGA system, the emulator provides a complete, functional implementation of the circuit that can evaluate millions of circuit cycles per second. This is orders of magnitude faster than even simulation-accelerators, since the multi-FPGA system can implement the complete circuit functionality in parallel, while accelerators simply provide one or more sequential logic evaluation processors.

Emulators provide a middle-ground between software simulation and prototyping. Compared to software simulation, an emulation executes much faster than a simulation. However, it can take a significant amount of time to map a circuit onto the emulator, and it is more difficult to observe and modify the behavior of the circuit. Thus, software simulation is a better choice for testing small subcircuits or small numbers of complete circuit cycles, where the software's flexibility and ease of use outweighs the performance penalties. Compared to a prototype, an emulation is much easier and faster to create, and it has much greater observability, controllability, and modifiability than a prototype. However, the emulation cannot run at the same speed as the target system. Thus, the emulator is a much better choice for providing a huge number of test cycles than a prototype when one expects to find bugs in the system, but it is no replacement for final checkout of the system via a prototype. For circuits that will execute software programs, an emulator can be used to debug this software much earlier in the design process than a physical prototype. This is because an emulation can be created from a high-level specification of the circuit, while prototyping must wait until the complete circuit has been designed. Simulation in general cannot be used for software development, since it is much too slow to execute enough cycles of the software. Also, just like a prototype, an emulation can be given to the end-user so that the circuit can be evaluated before the design is completed. In this way, the user can get a much better feeling for whether or not the design will fulfill the user's needs, something that is difficult with just a written specification. The emulation can be inserted into the target environment (as long as some method for reducing the performance demands on the system can be provided, such as those described in Chapter 8), and the system can be evaluated in a more realistic setting. This helps both to debug the circuit, and also to test the circuit interfaces and environment. For example, often a custom ASIC and the circuit board that will contain it will be developed simultaneously. An emulation of the ASIC can be inserted into this circuit board prototype, testing both the ASIC functionality as well as the board design.

One limitation of emulation is that it retains only the functional behavior of the circuit, which means that validation of the performance and timing features of a circuit cannot be performed on a logic emulator. Once a prototype is constructed, both logic emulation and software simulation are still valuable tools. When an error is found in a physical prototype, it can be difficult to isolate the exact cause in the circuit. An emulator can be used to reproduce the failure, since it can execute nearly as many cycles as the prototype, and the emulator's observability can be used to isolate the failure. Then, detailed testing can be performed by software simulation. Thus, logic emulation plays a complementary role to both software simulation and prototyping in logic validation.

Formal Verification

Formal verification [Yoeli90] includes techniques that can determine whether two logic realizations are identical, as well as techniques to decide whether a given logic realization could ever exhibit a certain behavior. For example, the designer of a circuit may work from a high-level specification to create an implementation of the circuit. Formal verification techniques can examine both of these realizations of the logic and determine whether they have exactly the same behavior. Note however that most formal verification techniques are restricted to functional validation (that is, they ignore the detailed timing and electrical characteristics of circuits). Thus, an actual implementation of the circuit could not be tested via formal verification, and would have to be abstracted into a functional description of the implementation. This abstraction process could be somewhat inexact, adding the potential for errors. However, ignoring this issue, formal verification techniques can tell designers whether or not they have successfully implemented a given specification. This is not the same as determining whether the implementation is actually what the user desires, since the specification may not have reflected these properly. However, if the two realizations differ, it is quite likely that there is an error in the implementation. Also, by determining that an implementation is the same as a specification, a user can then examine the specification to determine whether the implementation meets their needs. Since specifications will usually be much simpler than their implementations, this can simplify the validation process. Note that there is another approach to this process of ensuring that specification and implementation are identical: automatic synthesis methods that ensure "correct-by-construction" implementations. These systems translate a specification into an implementation by applying a set of transformations that have been formally proven correct. That is, the creator of the transformational system has examined each transformation, and proven that the circuit created by applying the transformation is always identical to the circuit to which the transformation is applied. Thus, if an implementation is derived from a specification via a "correct-by-construction" system, the designer already knows that the implementation is identical to the specification, and thus it is not necessary to apply formal verification to determine this.

There is a second portion of formal verification: the determination of whether a given realization can ever exhibit certain behaviors. For example, the designer of a resource controller may need to insure that two requesters can never access the resource at the same time (i.e., ensure mutual exclusion). The designer might use software simulation, as well as the execution of a completed prototype, to demonstrate that this requirement is met. However, in most complex systems it is impossible to exhaustively test the circuit, and thus all that the simulation and/or prototype execution can prove is that the requirement is not violated *in the situations that have been tested*. For many applications, especially safety-critical applications, this guarantee is not sufficient. Formal verification techniques hold perhaps the only solution to this problem. Formal verification methods examine a given realization of a circuit and efficiently determine either that the realization can never violate the constraint, or provide a counter-example where the circuit does violate the constraint. The counter-example is an allowable excitation of the circuit (i.e., a series of input signals that are not prohibited by the input signaling protocols) that will cause the constraint to be violated. Both of these results are valuable. In the case where the constraint is proven to be upheld, we have much greater confidence in the system than any other technique can achieve (note that this result may not be an absolute proof, since the representation of the circuit tested will at best be an abstracted version of the implementation, and this abstraction process may introduce errors). In the case where a counter-example is provided, this counter-example serves as a guide to understanding the failure mode and determining a fix.

While formal verification and "correct-by-construction" techniques offer the best hope for creating completely correct circuits, these techniques are not sufficient by themselves to handle the complete logic validation task. There are three problems with these techniques that somewhat restrict their utility. First, when applying these techniques to generate an implementation, we assume that the specification is in fact what the user desires. However, user needs and requirements are often ill-defined, and any attempt to codify them in a formal specification will inevitably introduce errors, misunderstandings, and unexpected results. Many of these problems are due to the fact that before users have actually used a system they may not know exactly what they really want, and early attempts to express these needs may be naïve. Second, these formal techniques in general require abstract mathematical models of the actual circuit behavior. These abstractions will be somewhat inexact, and there is a potential for errors both in translating a "correct-by-construction" realization into a real circuit implementation as well as in abstracting a circuit implementation into a form that formal verification techniques can examine. Finally, while formal verification techniques can efficiently handle some circuits, real circuits can be quite complex, with huge state spaces, greatly increasing the task of formal verification. These techniques, at least as developed so far, are not capable of handling complete, complex systems. Thus, formal verification techniques can be applied only to smaller subcircuits. However, even if individual subcircuits are verified to be correct, when they are composed together there is still the potential for failures due to their interactions.

Complete Logic Validation

As we have just discussed, there are numerous methods for performing logic validation, each with its own strengths and weaknesses. Current practice in general involves only software simulation during early design phases, followed by the construction and testing of a hardware prototype. Unfortunately, this ignores the fact that only formal verification can discover some kinds of unlikely, but still possible, failure modes of the circuit. Also, logic emulation can greatly improve the completeness of pre-prototype testing.

In an ideal validation environment - one where some of the current shortcomings of formal verification and logic emulation have been overcome - the methodology would take advantage of the strengths of each approach, while using multiple approaches to avoid their individual deficiencies. In early design phases, the user would be creating and modifying a specification of the circuit to be designed. Software simulation would handle most of the validation needs in this step, though formal verification techniques could be applied to discover failure modes in the specification. As this specification is subsequently refined into a complete implementation, correct-by-construction logic synthesis systems or formal verification techniques could be applied to avoid adding errors during this refinement process. At some point, software simulation will have uncovered most of the simple bugs, and further testing would only discover new failures after a large number of test cycles. At this point, emulation would become a better testing method than software simulation simply on performance grounds. Emulation may also be necessary earlier in the process to provide a platform for concurrent software development, and to provide a demonstration vehicle for end-user evaluation. Once failures are discovered via emulation, software simulation could help isolate the errors and develop the bug fixes. Eventually, the designers would have confidence that the current design is relatively bug-free. At this point, the time and expense of prototype construction are justified, and a prototype would be built. This would probably be done via a complete fabrication of the circuit, avoiding the unpredictable failures and inaccuracies of wire-wrap or breadboarding. The prototype could then be completely tested under true operating conditions. When failures occur, both emulation and software simulation could be brought to bear to isolate and correct the errors.

Under this ideal validation process several interesting things occur. Formal verification is applied to uncover failures that no other method would ever uncover. With the complexity of current systems, it is impossible to exhaustively test the circuit, and some unlikely situations would not be examined. Without formal verification, errors that occur only in these unlikely situations would only be discovered after the user is bitten by these bugs. Emulation also provides improved capabilities to the validation process. Since emulation could be applied much earlier in the design process than prototyping, many more test cycles could be completed in the early design phases. This would uncover failures earlier in the process, making the resulting debugging and correction easier and faster. Also, emulation provides a software execution platform much sooner than prototyping, allowing software development to begin much earlier in the process. It provides an early, working prototype of the system, which could be used for interface testing (such as when an emulated ASIC is inserted into the circuit board designed to contain it so that the board and board-chip interactions could be tested), as well as for early end-user evaluation. The latter is quite important, since the goal of logic validation is not just to determine whether the circuit meets the specification, but also make sure that the design is actually what the end-user really wants. Finally, emulation could delay the need for the construction of a physical prototype, saving much of the time and money that would otherwise be wasted on multiple prototype fabrications.

As we have argued, the best validation methodology is a combined methodology, taking advantage of the benefits of software simulation, emulation, prototyping, and formal verification. Software simulation helps the designers understand the design they are working on, and detect problems on a small scale. Emulation expands this capability to the complete system, giving the designers and end-users their earliest look at the complete system in operation. Prototyping provides the final sanity check, with no hidden errors due to false assumptions or incorrect abstractions of the true system behavior. Formal verification guards against the unlikely (yet still possible) error conditions, detecting problems that could be discovered via no other method. In this way, the best overall validation solution could be delivered.

In the next chapter we discuss current multi-FPGA systems, concentrating on their hardware structures. Then in Chapter 6 we present Springbok, a system for the logic emulation of board-level designs.