

## Chapter 2

# THE VERIFICATION UNIVERSE

**Abstract.** In this chapter we take the reader through a typical microprocessor's life-cycle, from its first high-level specification to a finished product deployed in a end-user's system, and overview the verification techniques that are applied at each step of this flow. We first discuss pre-silicon verification, the process of validating a model of the processor at various levels of abstraction, from an architectural specification to a gate-level netlist. Throughout the pre-silicon phase, two main families of techniques are commonly used: formal methods and simulation-based solutions. While the former provide mathematical guarantees of design correctness, the latter are significantly more scalable and, consequently, are more commonly used in the industry today. After the first few prototypes of a processor are manufactured, validation enters the post-silicon domain, where tests can run on the actual silicon hardware. The raw performance of in-hardware execution is one of the major advantages of post-silicon validation, while lack of internal observability and limited debuggability are its main drawbacks. To alleviate this, designers often augment their creations with special features for silicon state acquisition, which we review here. After an arduous process of pre- and post-silicon validation, the device is released to the market and finds its way into a final system. Yet, it may still contain subtle bugs, which could not be exposed earlier by designers due to very compressed production timelines. To combat these *escaped errors*, vendors and researchers in industry and academia have begun investigating alternative dynamic verification techniques: with minimal impact on the processor's performance, these solutions monitor its health and invoke specialized correction mechanisms when errors manifest at runtime. As we show in this chapter, all three phases of verification, pre-silicon, post-silicon and runtime, have their unique advantages and limitations, which must be taken into account by design houses to attain sufficient verification coverage within their time and cost budgets and to avoid major catastrophes caused by releasing faulty processor products to the commercial market.

## 2.1 Pre-silicon Verification

Pre-silicon verification is a multi-step process, which is aimed at establishing if a design adheres to its specification and fulfills the designer's intentions. Pre-silicon verification, or design-time verification, is conducted before any silicon prototype is available (hence its name), and operates over a range of different descriptions of a digital design: architectural, RTL, gate-level, *etc.* The higher levels of abstraction of the design allow engineers to check, often through mathematical proofs, fundamental properties of a circuit's operation, such as absence of erroneous behaviors and adherence to formally specified invariants in its functionality. The design is then progressively refined to include more detail, requiring the designers to check correctness after each transformation. However, with the growing level of detail, the time and computational effort required to validate the design's functionality increases as well, thus, in practice, only the most critical blocks of a modern processor are fully verified at all levels of abstraction before tape-out.

The history of pre-silicon verification of digital circuits goes hand in hand with the evolution of these devices over the last several decades: what began as a fairly simple in-house activity, quickly became a large industry of its own. Today, tens of electronic design automation (EDA) companies and a handful of the largest digital design houses offer designers a wide variety of software and hardware tools to address various phases of pre-silicon verification; while researchers in industry and academia alike publish thousands of papers yearly on the subject. Some of these techniques can be applied to any logic design, while others are domain-specific heuristics that improve the performance of verification for certain types of designs. Since it would be impossible for us to cover the entire spectrum of solutions available, in this section we simply focus on presenting a concise overview of the most vital verification techniques, and provide a range of references at the end of the chapter for the readers interested in deepening their knowledge on the subject.

### 2.1.1 *From specification to microarchitectural description*

In today's microprocessor industry, verification begins early in the design cycle, when architects of the new design develop a detailed set of specifications, outlining the major component blocks and describing their functionality. This specification document is then used as the yardstick to verify various pre-production implementations of the design at varying levels of detail. Note, however, that the document is typically written by several people in a natural human language, such as English, and, therefore, may contain hidden ambiguities or contradictions. Consequently, when bugs are found during the verification of a behavioral model, these might be due to a poor implementation or may be caused by errors in the specification itself, which then must be clarified and updated. As soon as a design specification becomes available, verification engineers begin crafting a *test plan* that outlines how individ-

ual blocks and the entire system will be verified, how bugs will be diagnosed and tracked, and how verification thoroughness, also known as *coverage*, will be measured. As with the specification, the test plan is not created once and forever, but rather evolves and morphs, as designers add, modify and refine processor's features.

At the same time, with specification at hand, engineers may begin implementing the design at the architectural (or ISA) level, describing the way the processor interacts with the rest of the computer system. For instance, they determine the format of all supported instructions, interrupts, communication protocols, *etc.*, but do not concern themselves with the details of the inner structure of the design. This is somewhat similar to devising the API of a software application without detailed descriptions of individual functions. Note that operating systems, user-level programs and many hardware components in a computing platform interact with the processor at the ISA level, and for the most part, need not be aware of its internals. In the end, the specification is transformed into an *architectural simulator* of the processor, which is typically a program written in a high-level programming language. As an example, Simics [MCE<sup>+</sup>02] and Bochs [Boc07] are fairly popular architectural simulators. An architectural simulator enables the engineers to evaluate the high level functionality of the design by simulating applications and it also provides early estimations of its performance. The latter, however, are very approximate, since at this point the exact latency of various operations is unknown. More importantly, at this stage of the development, the architectural simulator becomes the embodiment of the specification and in later verification stages it is referred to as a *golden model* of the design, to which detailed implementations (netlist-level and circuit-level) can be efficiently compared. The architectural simulator can then be refined into a *microarchitectural description*, where the detailed functionality of individual sub-modules of the processor is specified (examples of microarchitectural simulators are SimpleScalar [BA08] and GEMS [MSB<sup>+</sup>05]). With a microarchitectural simulator designers define the internal behavior of the processor and characterize performance-oriented features such as pipelining, out-of-order execution, branch prediction, *etc.* The microarchitectural description, however, does not specify how these blocks will be implemented in silicon and is usually also written in a high-level language, such as C, C++ or SystemC. Nevertheless, with the detailed information on instruction latency and throughput now available, the performance of the device can be benchmarked through simulation of software applications and other tests.

### 2.1.2 Verification through logic simulation

The microarchitectural level design is then further refined into a register-transfer level (RTL) implementation, typically written in a hardware design language (HDL), such as Verilog [IEE01b], SystemVerilog [IEE07] or VHDL [IEE04]. At this level, the functionality of individual blocks is further broken down into logic/mathematical operations and storage elements to contain the results of computation. Because of this refinement, the size of the code base becomes significantly larger, while the sim-

ulation performance of the design decreases by 4-5 orders of magnitude, compared to the architectural level. With a register-transfer level description it is possible to simulate and evaluate the behavior of the system very accurately, tracking the interaction of all components. The vast majority of the verification effort is dedicated to testing and simulating this level of the design description and most functional design bugs are exposed and corrected at this stage. However, one of the main drawbacks of simulation-based techniques is the fact that they can only identify the presence of errors but cannot guarantee their absence. In other words, with logic simulation designers can check if the processor behaves properly, *i.e.*, adheres to its specification, when running specific program sequences, designed and selected by the design team. However, there still may be latent bugs manifesting only when running other programs, that had not been simulated and verified. Nevertheless, this technique remains the primary method to validate RTL designs, especially in the microprocessor industry, due to its ability to scale and operate even on very complex processor descriptions. During logic simulation, internal design signals are monitored and evaluated one by one, making the time required to complete the simulation directly proportional to the size of the design and the length of the executed test sequence. Therefore, full processor designs can be simulated for millions of cycles, often providing satisfactory confidence that the system is error-free. Example of commercial logic simulators available today include VCS [Syn09] and ModelSim [Men08]. In this chapter we overview the basic components of a logic simulation framework, while the reader is encouraged to investigate surveys, such as [GTKS05, Mei93], and works on functional verification, such as [Mic03, WGR05], for a more detailed analysis of a range of simulation tools, setups and techniques.

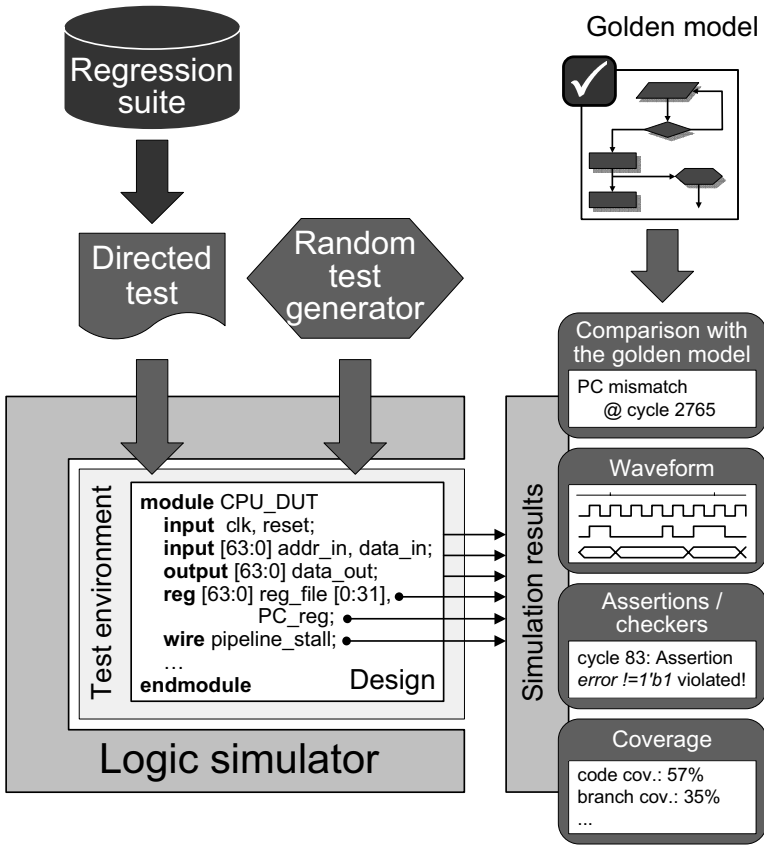
In a typical framework for RTL simulation, illustrated in [Figure 2.1](#), the design under validation is wrapped in a test environment that models the behavior of other components external to the system under verification. For instance, when the entire processor is simulated, the test environment represents systems external to the CPU, such as memory, peripherals, *etc.* The test environment and the design are executed together within the logic simulator. The simulator itself is a software application parsing the RTL code of the design and the test environment and computing circuit's output values when subjected to a given test. The outputs are generated by evaluating each internal design component throughout the entire simulated time interval. A simulator obtains the values of primary inputs to the design from a test and then evaluates the circuit's internal logic, as well as its primary outputs, in discrete timesteps. The obtained data can then be saved into a file for later comparison with the outputs of an architectural simulator, used as a golden model, or can be presented to a verification engineer in form of *waveforms*, *i.e.*, a diagram of the signals' values over time. The latter form is often adopted when diagnosing incorrect test responses to investigate the root cause of a bug, since all signal transitions can be visualized.

Test sequences supplied to the simulator can be deterministic, such as manually-generated assembly tests, or may be pseudo-random. The former are usually short and designed to validate specific features or modules of the device, as required by the test plan. Randomized tests are most commonly focused on system-level aspects and interactions, subjecting the design to a variety of stressful stimuli. To this end,

verification engineers leverage *pseudo-random test generators* (also called RTGs) that can be tuned to produce streams of valid instructions with a wide range of properties, *e.g.*, focused activity on certain functional units, specific inter-instruction dependencies, and so on [BLL<sup>+</sup>04, AAF<sup>+</sup>04]. Some RTGs have the ability to monitor a number of pre-selected internal design signals and use the information to dynamically adjust the test they generate so to boost stimulus quality [WBA07, FZ03]. Furthermore, the most relevant random and directed tests are combined into *regression suites*: sets of tests simulated each time that the design is modified, to guarantee that that the changes did not produce new bugs.

As mentioned before, simulation-based techniques can not provide guarantees of design correctness for the scenarios that are not explicitly tested. To gauge the need for additional verification throughout an industrial-scale digital system development, designers track coverage, which is a measure of thoroughness of verification. For example, achieving a 100% *code coverage* is a typical goal, thus requiring that every statement in the RTL code is activated in simulation least once. Code coverage alone is a fairly weak metric to evaluate the completeness of a test suite; thus, more sophisticated measures such as *functional* and *transaction coverage* are commonly used in addition to code coverage [Glu06]. As discussed in detail in [Piz04], functional coverage metrics allow to evaluate the testing quality of high-level functions in a design's block in a thorough and systematic fashion.

One of the shortcomings of the framework just described is its reliance on the architectural simulator to detect errors. Because modern microprocessors are extremely complex systems, with vast amounts of internal state, subtle errors in inner blocks can often take hundreds or thousands of cycles to manifest at the architectural level. Tracing an error backwards from an erroneous architectural event to the actual malfunction is a tedious and time-consuming process; therefore, the ability to detect issues as soon as they occur is a major advantage in terms of length of the diagnosis effort. One way to achieve this, is to use high-level behavioral models of individual blocks, instead of a monolithic architectural simulator. These models can be derived from the microarchitectural description of the processor and converted to dedicated checking units, running concurrently with the RTL simulation and comparing their output to the corresponding block's simulated output. Sometimes, block-level errors can be identified with simple scoreboards, which could, for instance, keep track of request-response pairs and thus detect rogue messages. Alternatively, designers can use *assertions* or *checkers* to encode invariants of the design's operation and make sure that they are upheld throughout the simulation. More information on this topic can also be found in [FKL04, YPA06]. Assertions on input signals are also helpful in detecting improper communication between blocks or errors arising from misinterpretation of the specification. Digital logic designers may interpret the specification in different ways while developing their components. However, if they encode their assumptions on the behavior of the surrounding components in the form of assertions, mismatches can be detected early in the development process, as soon as blocks are integrated. Finally, assertions may be used to evaluate coverage by monitoring the input signals into the assertion. To enable these advanced verification concepts, in the past few years, the electronic design industry has developed



**Fig. 2.1 A typical framework for simulation-based verification.** Inputs to a logic simulator are typically manually-written directed tests and/or randomized sequences produced automatically by a pseudo-random test generator. Tests are fed into a logic simulator, which, in turn, uses them as stimuli to the integrated test environment and design description. The test environment emulates the behavior of blocks surrounding the design under test. The simulator computed outputs and internal signal values of the design from the test's inputs. These outputs can then be analyzed with a variety of tools: they can be compared with the output of a golden architectural model, can be viewed as waveforms, particularly for error diagnosis, can be monitored by assertions and checkers to detect violations of invariants in the design behavior and, finally, can be used to track coverage, thus evaluating the thoroughness of the test.

dedicated languages for hardware verification (HVLs), some examples of which are OpenVera [HKM01], the *e* language [HMN01] and SystemVerilog [IEE07]. The latter implements a unified framework for both hardware design and verification, providing an object-oriented programming model, a rich assertion specification language, and even automatic coverage collection.

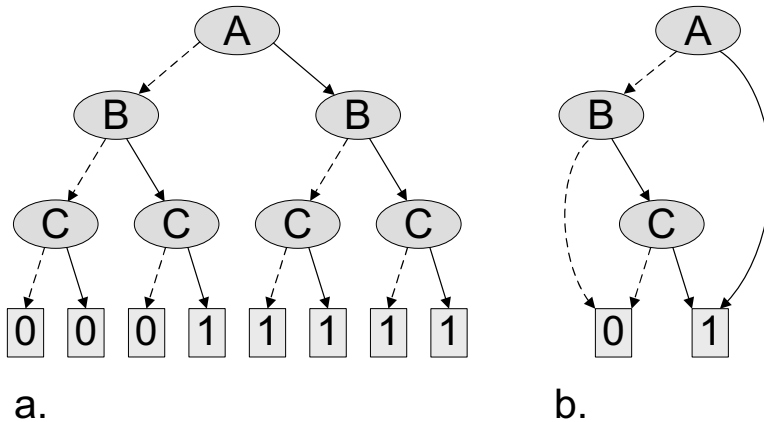
### 2.1.3 Formal verification

Formal verification encompasses a variety techniques that have one common principle: to prove with mathematical guarantees that the design abides to a certain specification under all valid input sequences. If the design does not adhere to the specification the formal techniques should produce a counterexample execution sequence, which describes one way of violating the specification. To this end, the design under verification and the specification must be represented as mathematical/logic formulas subjected to formal analysis. One of the most important advantages of these solutions over simulation-based techniques is the ability to prove correctness for *all legal stimuli sequences*. As we described in the previous section, only explicitly simulated behaviors of the design can be tested for correctness and, consequently, only the presence of bugs can be established. Formal verification approaches, however, can reason about absence of errors in a design, without the need to exhaustively check all of its behaviors one by one. The body of work in the field of formal verification is immense and diverse: for decades researchers in industry and academia have been developing several families of solutions and algorithms, which are all too numerous to be fully discussed in this book. Fortunately for the readers, sources such as [Ber05, PF05, PBG05, BAWR07, WGR05, CGP08, KG99] and many others, describe the solutions in this domain in much greater detail. In this book, we overview some of the most notable techniques in the field in the hope to stir the readers' curiosity to investigate this research field further. However, before we begin this survey, we first must take a look at two main computation engines, which empower a large fraction of formal methods, namely SAT solvers and binary decision diagrams (BDDs).

SAT is a short-hand notation for Boolean satisfiability, which is the classic theoretical computer science NP-complete problem of determining if there exists an assignment of Boolean variables that evaluates a given Boolean formula to *true*, or showing that no such assignment exists. Therefore, given Boolean formulas describing a logic design and a property to be verified, a SAT-based algorithm constructs an instance of a SAT problem, in which a satisfactory assignment of variables represents a violation of the property. For example, given a design that arbitrates bus accesses between two masters, one can use a SAT-solver to prove that it will never assert both grant lines at the same time. Boolean satisfiability can also be applied to *sequential circuits*, which are in this case “unrolled” into a larger design by replication of the combinational logic part and elimination of the internal state elements. Engineers can then check if certain erroneous states can be reached in this “unrolled” design or if invariants of execution are always satisfied. As we will describe in the subsequent section, SAT techniques can also be used for equivalence checking, *i.e.*, establishing if two representations of a circuit behave the same way for all input sequences. Today, there is a variety of stand-alone SAT-solver applications available, typically they either implement a variation of the Davis-Putnam algorithm [DP60] as, for instance, GRASP [MSS99] and MiniSAT [ES03], or use stochastic methods to search for satisfiable variable assignments as in WalkSAT [SKC95]. Heuristics and inference procedures used in these engines can dramatically improve



the performance of the solver; however, in the worst case the satisfiability problem remains NP, that is, as of today, it requires exponential time on the input size to complete execution. As a result, solutions based on SAT solvers cannot be guaranteed to provide results within reasonable time. Furthermore, when handling sequential designs, these techniques tend to dramatically increase the size of the SAT problem, due to the aforementioned circuit “unrolling”, further exacerbating the problem.



**Fig. 2.2 Binary decision diagrams.** **a.** A full-size decision tree for the logic function  $f = A|(B\&C)$ : each layer of nodes represents a logic variable and edges represent assignments to the variable. In the figure, solid edges represent a 1 assignment, while dashed edges represent a 0 value of the variable. The leaf nodes of the tree are either 1- or 0-type and represents the value that the entire Boolean expression for the assignment in the corresponding path from root to leaf. Note that the size of the full-size decision tree is exponential in the number of variables in the formula. **b.** Reduced ordered binary decision diagram for the logic function  $f = A|(B\&C)$ . In this data structure redundant nodes are removed, creating a compact representation for the same Boolean expression.

Binary decision diagrams (BDDs), the second main computational engine used in formal verification, are data structures to represent Boolean functions [Bry86]. BDDs are acyclic directed graphs: each node represents one variable in the formula and has two outgoing edges, one for each possible variable assignment – 0 and 1 (see Figure 2.2). There are two types of terminal (also called “leaf”) nodes in the graph, 0 and 1, which correspond to the value assumed by the entire function for a given variable assignment. Thus, a path from the root of the graph to a leaf node corresponds to a variable assignment. The corresponding leaf node at the end of the path represents the value that the logic function assumes for the assignment. BDDs are reduced to contain fewer nodes and edges and, as a result, can often represent complex Boolean functions compactly [Bry86, BRB90]. An example of this is illustrated in Figure 2.2.b, where redundant nodes and edges in the complete binary decision tree for the function  $f = A|(B\&C)$  are removed, creating a structure that is linear in the number of variables. BDD software packages as, for instance CUDD [Som09], contain routines that allow a fast and efficient manipulation of BDDs and



on-the-fly tree minimization algorithms. Within formal verification, binary decision diagrams can be used for equivalence checking, *i.e.*, testing if two circuits implement the same logical function, in reachability analysis and symbolic simulation, as well as in other techniques. In reachability analysis, BDDs are used to represent the set of states that a design can attain under all possible inputs. This set can be later analyzed to detect the presence of erroneous states. In symbolic simulation, on the other hand, decision diagrams store formulas representing the design's state and the functionality of the outputs in terms of all possible input values. There are, however, a few drawbacks to BDD-based solutions, the most important one being the "memory explosion problem": a situation, where functions cannot be represented compactly by the data structure, causing a prohibitively large number of nodes to be created in the host computer's memory. As a consequence, the host may terminate before completion of the formal proof. The engineering team is then forced to restructure the design or apply simplifying assumptions or *constraints* to the system, abstracting away some of the possible behaviors.

As we already mentioned above, SAT-solvers and BDD libraries enable designers to establish how circuits operate under all possible input conditions and conduct a variety of analyses. Let us now briefly review some of the formal verification solutions that are employed in today's semiconductor industry, outlining their advantages and limitations. Note that only major families of solutions are presented here, since a comprehensive survey is beyond the scope of this book.

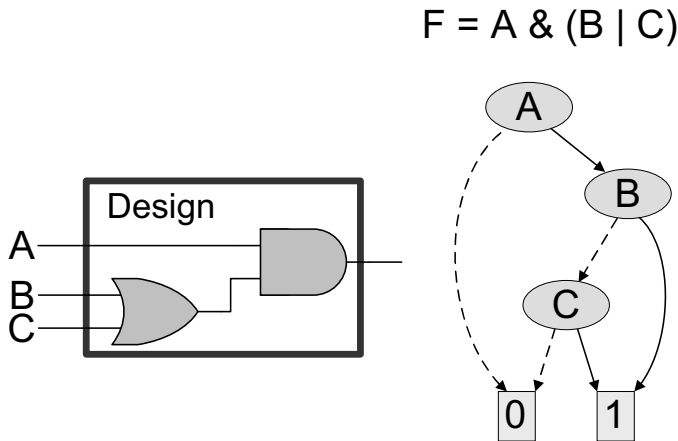
**Theorem proving** is a powerful technique that uses automatic reasoning to derive proof for "theorems" about the system under study. Theorems are mathematical formulas that describe the design's functionality and its properties in abstract form. Theorem provers may use a variety of theories to derive the mathematical transformations required to prove a given theorem. More information may be found in [KK94], which also provides a collection of references on this subject. Theorem provers have found wide acceptance in both the software and hardware verification domains, however, these systems are still not fully automatic and often human assistance is required in directing the proof derivation process. Consequently, engineers spend valuable time investigating conjectures produced by the prover tool and assisting it in reaching its goal. Furthermore, the behavior of the design and the properties are usually specified in abstract form, requiring even more engineering effort. As the result, theorem provers are often applied to high-level protocol or architectural descriptions to ensure that such invariants as absence of deadlock and fairness are upheld. The RTL implementation can then be compared to the formally verified architectural model using more scalable techniques, such as simulation.

The **reachability analysis** problem has been mentioned before and consists of characterizing the set of states that a design can attain during execution. To this end the combinational function of the design is typically represented as a binary decision diagram and the reset state of the circuit is used to initialize the "reached set". Then BDD manipulation functions are used to compute all states that the system can attain after one clock cycle of execution, which are then added to the reached set.

The process continues until the host runs out of memory or the reached set stops increasing, in which case the obtained set contains all states that a system can achieve throughout the execution and can be subsequently analyzed for absence of errors and presence of runtime invariants. Reachability analysis is a very powerful verification tool, although one must keep in mind that this technique, as other BDD-based solutions, suffers from the memory explosion problem and has limited scalability. See also publications such as [CBM89, RS95] for more information on this approach.

**Model checking**, described in detail in [CGP08, CBRZ01], is another powerful technique that establishes if states and transitions within the design adhere to a formal specification. Properties in this technique are typically written as formulas in temporal logic, which reasons about events in time, for instance, a property may require that a given system's event will eventually occur, independently of the type of execution. However, frequently, formal properties have a limited time window in which they need to be considered, in which case bounded model checking (BMC) can be used. In bounded model checking a property is considered only over a finite number of clock cycles of execution: if no violation is exposed then the property holds. Solutions in this domain may use either BDDs or SAT solvers as the underlying engine. A variety of property specification languages exists today, including PSL [Acc04], recently extended to PSL/Sugar [CVK04], System Verilog Assertion language (SVA) [VR05], which is a subset of SystemVerilog, the proprietary Bluespec language [Arv03], TLA+ [Lam02], *etc.* However, since these languages are declarative, they are often fairly complex to use in describing non-trivial properties and in general are more difficult to use than imperative languages. Indeed, industry experts report that, when a property violation is exposed in model checking, it is more often the case that the problem lies in the property description than in the design itself. The scalability of model checking may also be limited due to the underlying engines's complexity and memory demands.

**Symbolic simulation** [CJB79, Bry85, BBB<sup>+</sup>87, BBS90, Ber05] has many similarities with logic simulation, in that output functions are computed based on input values. The main difference here is that inputs are now symbolic Boolean variables and, consequently, outputs are Boolean functions. For instance, the symbolic simulation of the small logic block in Figure 2.3 produces the Boolean expression  $A \& (B|C)$  for the output, which we show in the BDD depicted by the output wire. Expressions for the state of the design and the values of its outputs are updated at each simulation cycle. The Boolean expressions altogether are a compact representation of all the possible behaviors that the design can manifest, for all possible inputs, within the simulated cycles. To find the response of the circuit for a concrete input sequence (a sequence of 0s and 1s), one simply needs to evaluate the computed expressions with appropriate values for the primary inputs. Consequently, symbolic simulation can be used to compute a reached state set (within a fixed number of simulation cycles) or prove a bounded time property. As with other solutions, however, the reliance on BDDs brings the possibility of the simulator exhausting the memory resources of the host before errors in the circuit can be identified.



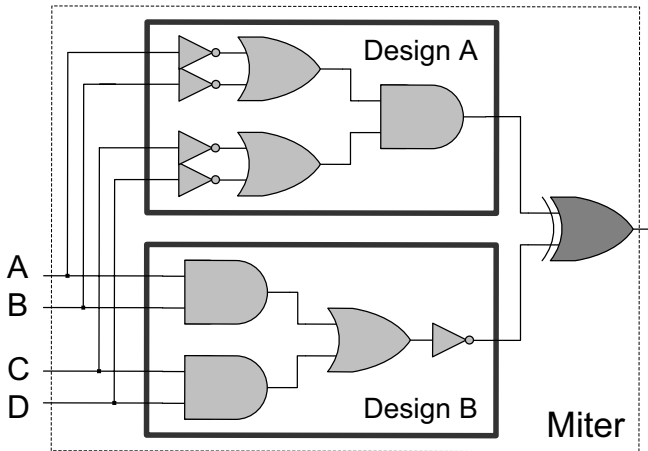
**Fig. 2.3 An example of symbolic simulation.** In symbolic simulation, outputs and internal design nodes are expressed as Boolean functions of the primary inputs, which, in turn, are described by Boolean variables. In the circuit shown in the figure, symbols A, B and C are applied to the primary inputs and the output assumes the resulting expression  $A \& (B \mid C)$ . Functions in symbolic simulation are typically represented by binary decision diagrams, as the one shown at the circuit's output.

As discussed above, modern digital logic designers have access to a wide variety of high-quality formal tools that can be used for different types of analyses and proofs. Yet all of them have limitations, requiring either deep knowledge of declarative languages for specification of formal properties or being prone to exhausting memory and time resources. Consequently, processor designers, who work with extremely large and complex system descriptions, must continue to rely mostly on logic simulation for verification of their products at the pre-silicon level. The guarantees for correctness that formal techniques provide, however, have not been overlooked by the microprocessor industry, and formal tools are often deployed to verify the most critical blocks in modern processors, particularly control units. In addition, researchers have designed methods to merge the power of formal analysis with the scalability of simulation-based techniques, creating *hybrid or semi-formal solutions*, such as the ones surveyed in [BAWR07]. Hybrid solutions use a variety of techniques to leverage the scalability of simulation tools and the depth of analysis of formal techniques to reach better coverage quality with a manageable amount of resources. They often use a tight integration among several tools and make an effort to have them exchange relevant information. Although hybrid solutions combine the best of the formal and simulation worlds, their performance is still outpaced by the growing complexity of processor designs and shortening production schedules. Thus, the verification gap continues to grow and designs go into the silicon prototype stage with latent bugs.

### 2.1.4 Logic optimization and equivalence verification

The RTL simulation and formal analysis described above are, perhaps, the most important and effort-consuming steps in the entire processor design flow. After these steps are completed, the register-transfer level description of the design must be transformed and further refined before a prototype can be manufactured. The next step in the design flow is *logic synthesis*, which automatically converts an RTL description into a gate-level netlist (see [HS06] for more details of logic synthesis algorithms and tools). To this end, expressions in the RTL source code are expanded and mapped to structures of logic gates, based on the target library specific to the manufacturing process to be used. These, in turn, are further transformed through local and global optimizations in order to attain a design with better characteristics, such as lower timing delay, smaller area, *etc.* Again, with the increasing level of detail, the code base grows further and, correspondingly, the performance of the circuit's simulation is reduced. Formal techniques are often deployed in this stage to check that transformations and optimizations applied during synthesis do not alter the functionality of the design. This task is called *equivalence checking* and it is most often deployed to prove the equivalence of the combinational portion of a circuit. In contrast, sequential equivalence checking has not yet reached sufficient robustness to be commonly deployed in the industry. The basis of combinational equivalence checking relies on the construction of a *miter* circuit connecting two netlists from before and after a given transformation. The corresponding primary inputs in both netlists are tied together, while the corresponding outputs are connected through *xor* gates, as shown in Figure 2.4. Several techniques can then be used to prove that the miter circuit outputs 0 under all possible input stimuli, thus indicating that the outputs of the two netlist are always identical when subjected to identical input. Among these, are BDD-based techniques, used in a fashion similar to symbolic simulation, SAT solvers proving that the miter circuit cannot be satisfied (that is, can never evaluate to 1), and also graph isomorphism algorithms, which allow to prune the size of the circuits whose equivalence must be proven.

After the netlist is synthesized, optimized and validated, it must undergo the *place and route* phase, where a specific location on silicon is identified for each logic gate in the design and all connecting wires are routed among the placed gates [AMS08]. Only after this phase the processor is finally described at the level of individual transistors, and thus engineers can validate properties such as electrical drive strength and timing with accurate SPICE (Simulation Program with Integrated Circuit Emphasis) techniques. It is typical at this stage to only be able to analyze a small portion of the design at a time, and to focus mostly to the critical paths through the processor's sub-modules.



**Fig. 2.4 Miter circuit construction for equivalence checking.** To check that two versions A and B of a design implement the same combinational logic function, a miter is built by tying corresponding primary inputs together and *xor*-ing corresponding primary outputs. Several techniques, including formal verification engines such as Binary Decision Diagrams and SAT solvers can then be invoked to prove that under the outputs of such miter circuit can never evaluate to 1, under all possible input combinations.

### 2.1.5 Emulation and beyond

As mentioned before, RTL simulation is several orders of magnitude slower than simulation at the architectural level. However, its performance can be improved by leveraging emulation techniques. In *emulation*, also called fast-prototyping, a design is mapped into programmable hardware components, such as FPGAs. These components can be configured after their manufacturing to implement any logic function in hardware and, thus, can be used to create early prototypes of complex systems before they are manufactured. Several companies provide fast prototyping solutions based on FPGAs or other specialized reconfigurable components. It is typical for these systems to have lower performance and lower device density than the final manufactured version of the system under verification. On the other hand, emulation can be conducted on a hardware prototype at much better performance than in a software simulator. Modern FPGAs can run at speeds of hundreds of megahertz (about one order magnitude slower than today's processors) and one may need dozens of them to model a single processor. Note also that, when a block of logic is mapped to an FPGA, its internal signals become invisible to the engineer, thus error diagnosis may become more challenging, unless other means of access to internal signals are specifically implemented in the prototype. Despite these shortcomings, the emulation of a processor design is an important and useful step in pre-silicon verification, since it allows a detailed netlist description to be executed at fairly high performance, enabling testing with longer sequences of stimuli and achieving higher design coverage.

After the design is deemed sufficiently bug-free, that is, satisfactory coverage levels are reached and the design is stable, the device is *taped out*, *i.e.*, sent to the fabrication facility to be manufactured in silicon. Once the first few prototypes are available, the verification transitions into the post-silicon phase, which we describe in the following section. It is important to remember, that the pre- and post-silicon phases of processor validation are not disjoint ventures - much of the *verification collateral*, generated in early design stages, can and should be reused for validation of the manufactured hardware. For instance, random test generators, directed tests and regression suites can be shared between the two. Moreover, in the case of randomized tests, architectural simulators can be used to check the output of the actual hardware prototypes for correctness. Finally, RTL simulation and emulation provide valuable support in silicon debugging. As we explain later, observability of the design's internal signals in post-silicon validation is extremely limited, therefore, it is very difficult to diagnose internal design conditions that manifest in to an error. To alleviate this, test sequences exposing bugs may be replayed in RTL simulation or emulated, to narrow the region affected by the problem and to diagnose the root cause of the error.

In summary, the pre-silicon verification of a modern processor is a complex and arduous task, which requires deep understanding of the design and the capabilities of validation tools, as well as good planning and management skills. After each design transformation, several important questions must be answered: what needs to be verified? how do we verify it? and how do we know when verification is satisfactory? Time is another important concern in this process, since designers must meet stringent schedules, so their products remain competitive in the market. Therefore, they must often forgo guarantees of full correctness for the circuit, and rely on coverage and other indirect measures of validation thoroughness and completeness. Exacerbating the process further is the fact that the design process is not a straightforward one - some modules and verification collaterals may be inherited from previous generations of the product, and some may be only available in a low level description. For instance, performance-critical units may be first created at gate or transistor level (so called *full-custom design*), and the corresponding RTL is generated later. Moreover, often the verification steps discussed in this section must be revisited several times, *e.g.*, when a change in one unit triggers modifications to others components. Thus, pre-silicon verification goes hand in hand with design, and it is often hard to separate them from each other.

## 2.2 Post-silicon Validation

Post-silicon validation commences when the first prototype of a design becomes available. With the actual silicon in hand, designers can test many physical characteristics of the device, which could not be validated with models of the design. For instance, engineers can determine the *operational region* of the design in terms of such parameters as temperature, voltage and frequency. Physical properties of

the device can also be evaluated using detailed transistor-level descriptions, as discussed in the previous section but, when using such models, these analyses can only be conducted on fairly small portions of the design and on very short execution sequences, making it difficult to attain highly quality measures of electrical aspects. For instance, the operational region of the device cannot be evaluated precisely in pre-silicon and must be checked after the device is manufactured. The actual operational region of the design is compared to the requirements imposed by the specification, driven by the market sector targeted by this product. Processors for mobile platforms, for example, usually must operate at lower voltages, to reduce power consumption, while server processors, for which performance is paramount, can tolerate higher temperatures, but also must run at much higher frequency. Other electrical properties that are also typically checked at this stage include drive strength of the I/O pins, power consumption, *etc.*

Most of the electrical defects targeted during post-silicon validation are first discovered as functional errors. For instance, incorrectly sized transistors on the die may result in unanticipated critical paths. Consequently, when the device is running at high frequency, occasionally data will not propagate through these paths within a single cycle, resulting in incorrect computation results. Similarly, jitter in the clock signal may cause internal flip-flops to latch erroneous data values, or cross-talk between buses may corrupt messages in flight. Such bugs are frequently first found as failures of test sequences to which the prototype is subjected. Designers proceed then to investigate the nature of the bug: by executing the same test sequence on a other prototypes they can determine if the bug is of electrical or functional nature. In fact, functional bugs will manifest on all prototypes, while electrical ones will only occur in a fraction of the chips. Bugs that manifest only in a very small portion of the prototypes are deemed to be due to manufacturing defects. Because each of these issues are diagnosed with distinct methods, a correct classification is key to shorten the diagnosis time.

In debugging electrical defects, engineers try to first determine the boundaries of the failure, *i.e.*, discover the range of conditions that trigger the problem, so it can be reproduced and analyzed in the future. To this end *shmoo plots* that depict failure occurrences as a function of frequency and supply voltage are created. Typically, multiple shmoo plots for different temperature settings are created, adding the third dimension to the failure region of the processor. This data can then be analyzed for characteristic patterns of various bug types. For instance, failures at high temperatures are strong indicators of transistor leakage and critical path violations, while errors that occur at low temperatures are often due to race conditions, charge sharing, *etc.* [JG04]. Designers then try to pinpoint the area of the circuit where the error occurs by adjusting the operating parameters of individual sub-modules with techniques such as on-die clock shrinks, clock skewing circuits [JPG01] and optical probing, which relies on lasers to measure voltage across individual transistors on the die [EWR00]. In optical probing, the silicon substrate on the back side of the die is etched and infrared light is pulsed at a precise point of the die. The silicon substrate is partially transparent to this wavelength, while doped regions of transistors reflect the laser back. If electrical charge is present in these regions, the



power of the reflected light changes, allowing the engineers to measure the voltage. Note that in this case, the top side of the processor, where multiple metal layers reside, does not need to be physically etched or probed, so integrity of the die is not violated. Unfortunately, the laser cannot get to the back side of the die through a heat sink, which, therefore, must be removed around the sampling point. While providing good spacial and timing resolution, optical probing remains a very expensive and often ineffective way of testing, since it requires sophisticated apparatus and enables access to only a single location at a time. Finally, when the issue is narrowed down to a small block, transistor-level simulation can be leveraged to establish the root cause of the bug and determine ways to remedy it.

As we mentioned above, in addition to electrical bugs, two types of issues can be discovered in manufactured prototypes: fabrication defects and functional errors, which are the target of *structural testing* and *functional post-silicon validation*, respectively. Although similar at first glance, these approaches have a very important difference in that testing assumes that the pre-silicon netlist is functionally correct and tries to establish if each prototype faithfully and fully implements it. Validation, on the other hand checks if the prototype's functionality adheres to the specification, that is, if the processor can properly execute software and correctly interact with other components of a computer system. In the following section we overview structural testing approaches and discuss silicon state acquisition techniques, which are typically deployed in complex designs to improve testability. Incidentally, most of these acquisition solutions can be also used in post-silicon validation, discussed in Section 2.2.2.

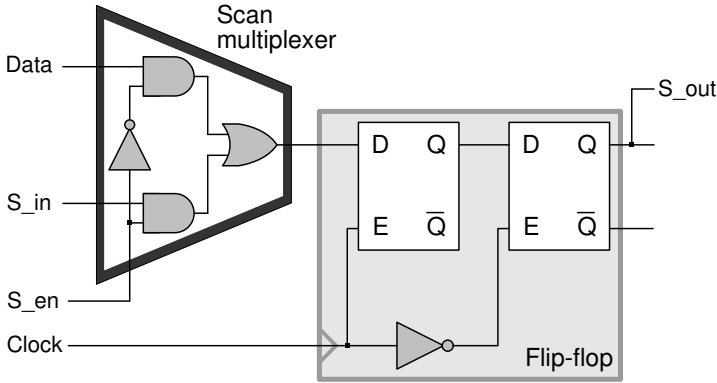
### 2.2.1 Structural testing

Modern integrated circuits are manufactured with a photolithographic process, which “prints” individual transistors, as well as metal interconnect between them, in multiple layers onto a silicon substrate. However, due to the nanometer-scale of transistors and wires, their features' boundaries may be blurry or distorted. This may cause, for example, two metal paths to be shorted together, or result in misalignment of the doped regions of a transistor, impairing the overall functionality of the circuit. The phase of post-silicon validation that tries to uncover these faults is called *structural testing*. In this framework, the gate-level netlist of the design is assumed as the golden model of functionality and used by *automatic test pattern generators* (ATPGs) in producing test sequences that check these design aspects. An ATPG analyzes the combinational logic of the netlist and infers test vectors that would expose a range of possible defects generated in the manufacturing process. In a sense, ATPG-based structural testing can be thought of as equivalence checking between the pre-production and the printed netlists. The type of faults that ATPGs can discover include shorts and opens, “stuck-at” defects, where a wire's value never changes, and violations of the circuit's internal propagation delays. ATPG patterns are then applied to the silicon prototype and the responses are compared to those

predicted by simulation on the pre-production netlist. For example, to test an implementation of a two-input logical *and* gate, an ATPG solution would check that the output of the silicon prototype is consistent with the truth table of the function, that is only when both inputs are high, the output value is equal to one. Modern testing tools typically do not subject the design to an exhaustive set of test vectors, which may be prohibitively large, but use advanced heuristics to minimize the set of test patterns without loss of defect coverage. Note, however, that ATPG testers cannot discover functional errors in the circuit, that is, discrepancies between design intent and the implemented silicon prototype.

In addition to its inability to discover functional errors in the circuit, the scalability of structural testing is severely limited in sequential designs: *i.e.*, systems that have internal storage elements in addition to combinational logic blocks. This is especially pronounced in complex microprocessor designs, where data is retained by internal storage elements for many cycles. As a consequence, it is virtually impossible for an ATPG technique to create test vectors to be applied to primary inputs of the circuit that can test behavior of logic functions deep inside of the design and control all types of manufacturing faults. Likewise, it may be impossible to propagate the information about the error to primary outputs to be observed by the designer. Faced with the dual problem of *controllability* and *observability* in such complex sequential circuits, it is mainstream today to augment the design with structures that allow comparatively easier access to internal logic nodes through I/O pins. This is commonly referred to as *design for testability*, or DFT. In particular, DFT techniques often provide ways to sample and write state elements of the circuits, such as flip-flops and latches, so combinational logic can efficiently be tested by ATPG patterns. Furthermore, as discussed later in this chapter, DFT techniques play an important role in functional post-silicon validation and debug, where they are used to analyze the internal behavior of a prototype that leads to an error. Research on structural testing and DFT has been carried out for decades in both industry and academia and it would be impossible to overview the most successful techniques within the scope of this section. Therefore, we will limit ourselves to briefly discuss a handful of the most notable solutions for silicon state acquisition, and recommend two textbooks as starting points for a more in depth study: “Digital Systems Testing and Testable Design” [ABF94] and “Essentials in Electronic Testing” [BA00].

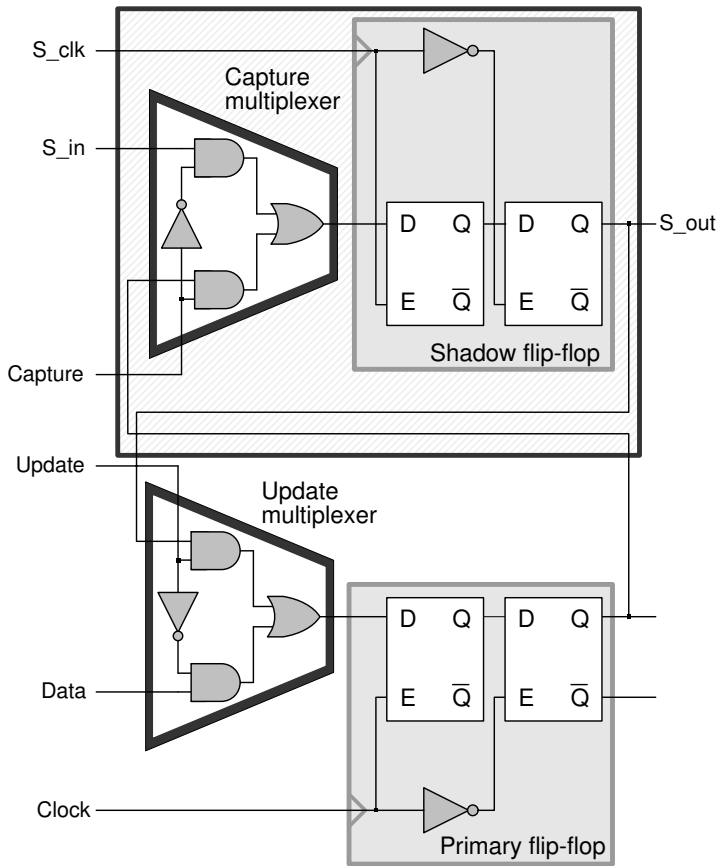
One of the most basic and classic techniques in the DFT domain are *scan chain* components, an example of which is shown in [Figure 2.5](#). To include a scan chain in a design, flip-flops are augmented with an additional data input (*scan in*) and output (*scan out*), as well as an enable line. The scan in and scan out lines of different storage elements are then connected serially into a *scan chain*, the ends of which are tied to special I/O pins of the device. When the scan enable signal is de-asserted, the flip-flops act as regular storage cells and the processor operates in normal mode. When enable is asserted, on the other hand, the scan chain reconfigures itself into a serial register, so that data can be passed serially from one flip-flop to another. With this tool at hand, verification engineers can suspend execution and scan out the values of the chained storage elements through a single output wire and analyze it. Moreover, an arbitrary internal state can be pre-set through the scan in functionality, enabling



**Fig. 2.5 Scan flip-flop design.** To insert a regular D flip-flop in a scan chain, the flip-flop is augmented with a scan multiplexer, which selects the source of data to be stored. During regular operation, the  $S_{en}$  (scan enable) signal is de-asserted and the flip-flop stores bits from the Data line. When enable is asserted, however, the flip-flop samples the scan in ( $S_{in}$ ) signal instead. This allows the designers to create a scan chain, by connecting the scan out ( $S_{out}$ ) output with a scan in input of the next flip-flop in the chain. The last output in this chain is connected to a dedicated circuit output port, so that the internal state of the system may be shifted out by asserting  $S_{en}$  signal and pulsing the clock. Likewise, since the  $S_{in}$  input of the first chain element is driven from a circuit's primary input, engineers can quickly pre-set an arbitrary internal state in the design for testing and debugging purposes. Note that during scan chain operations the regular design functionality is suspended.

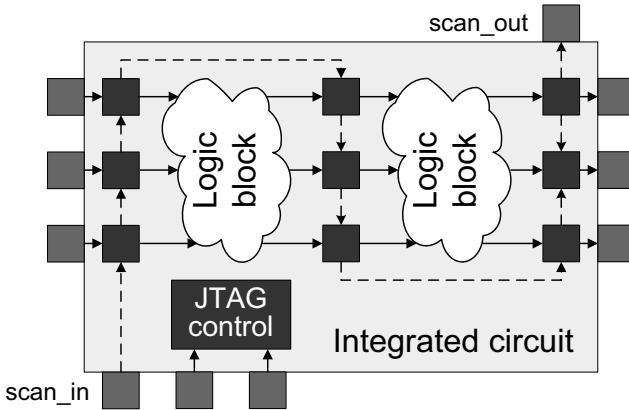
fine-grain controllability of the device, in addition to observability. Particular implementations of the scan technique vary in different designs and include full-scan (all state elements are connected), partial scan (a subset of flip-flops is connected), and multiple parallel chains. The drawbacks of the approach are a somewhat slower latch operation, the additional area and interconnect overheads, and, most importantly, the need to suspend operation of the device under test to shift the state out or bring a new state in, an activity that requires several clock cycles.

Modern scan chains often rely on an even more complex scan flip-flop design, that overcomes the limitation of having to suspend execution while routing state in and out of the system. *Hold-scan* flip-flops consists of two basic scan flip-flops connected together, called a primary and a shadow flip-flop, as shown in Figure 2.6. When the capture line is asserted, the shadow element samples and stores the value of the main latch. This flip-flop can be reconfigured into a scan chain connection at any later time, so the sampled data can be routed outside of the device. Since the chain comprises only shadow flip-flops and operates on an independent clock, the device can continue to function normally, while the captured snapshot is being transferred out. Similarly, a new state can be loaded into the shadow latches without interrupting the operation of the device and then propagated to main flip-flops by the asserting the update line. Although hold-scan flip-flops have significantly larger area than the baseline design of Figure 2.5, it provides the designers with much more flexibility in the debugging process and, consequently, and it is frequently deployed in microprocessor systems [KDF<sup>+</sup>04].



**Fig. 2.6 Hold-scan flip-flop design.** Hold-scan flip-flops provide the ability to overlap system execution with scan activity. The component comprises two scan flip-flops, a primary and a shadow flip-flop. The shadow flip-flop can capture and hold the state of the primary storage element. Shadow elements are connected in chain and can transmit the captured values without the need to suspend regular system operation. Similarly they can be used to load a system state, which is then transferred to the primary flip-flops by asserting the update signal. Note that shadow flip-flops operate on a separate clock ( $S\_clk$ ), so that the transmit frequency can be decoupled from the system's operating frequency.

*Boundary-scan* is another technique often used in structural testing and validation, to allow individual modules of the processor to be tested in isolation [IEE01a]. Boundary-scan was developed by the Joint Test Action Group (JTAG), which was formed by the industry to enable testing of complex circuits boards. JTAG calls for inputs and outputs of the chip, or a block of logic, to be tied to dedicated, scan storage elements, as shown in Figure 2.7. These cells can be configured to perform several distinct functions by means of specialized control logic. For instance, inputs to the block can be captured to verify operation of other modules or can be used provide stimulus to the block. The response to the stimulus can be then observed



**Fig. 2.7 Boundary-scan technique (JTAG).** Scan storage cells are inserted and connected to the I/O pins of a chip or at the boundary of internal logic blocks. These cells are also chained together as indicated by the dashed lines in the figure. Through an additional on-die JTAG controller, designers can control the operation of the cells and use them to test the interconnect among chips on the same board or among blocks within a chip, and test individual blocks in isolation.

in the boundary-scan flip-flops tied to its outputs. Today, JTAG technology is often extended to perform additional configuration and testing operations, such as on-chip programming, access to special memory or registers, *etc.*

Despite all these functionalities, scan-based techniques still lack an important feature, that is, automatic acquisition of the internal state triggered by a design's behavior. Scan chain technology requires that an enable pin is externally asserted in order to initiate sampling. On-chip logic analyzers (OCLA) alleviate this limitation, augmenting the device with programmable trigger-detection and response logic, thus making the sampling activity programmable [CKM<sup>+</sup>04]. OCLAs are designed to observe internal signals continuously and to match them against pre-set triggering conditions such as, for instance, a signal assuming a specific, user-programmed logic value or undergoing a certain transition. When a triggering condition occurs, the analyzer begins a continuous sampling of the design's state for a given time period and stores the data into dedicated memory. A designer can later download this data and monitor the behavior of the circuit throughout several cycles of execution. Unfortunately, the advanced functionality of on-chip logic analyzers comes at the price of high area penalty and the need to route the sampling wires throughout the die. In the highly competitive settings of the microprocessor industry, this area overhead can rarely be afforded, compelling companies to limit the use of OCLAs [Lit02] and/or diagnose bugs by relying simply on the scan-based testing and verification techniques overviewed above.

### 2.2.2 *Functional post-silicon validation*

Identification of functional bugs is another crucial part of the processor design and manufacturing effort, whose importance has been quickly growing in recent years. As we mentioned earlier, functional post-silicon validation differs from structural testing in that it does not target errors due to manufacturing issues, but strives to establish if the prototype adheres to its specification and faithfully embodies the designer's intent. The main advantage that designers can leverage in post-silicon, compared to pre-manufacturing validation, is the test execution speed. Indeed, instead of an architectural or RTL simulation, in post-silicon validation programs execute on the actual hardware, outperforming the former two techniques by multiple orders of magnitude. Therefore, significantly longer sequences can be tested and, consequently, higher coverage may be attained. To put things in perspective, consider that during the many months of validation of the Pentium 4 processor, an experience discussed in Section 1.3, a total of about 200 billion cycles were simulated before tape-out [Ben01]. After manufacturing, these accounts for only three minutes of runtime in the actual processor, which operates at 1GHz frequency. Functional post-silicon validation is carried out through directed and parameterized random tests. The drawback of the latter is the lack of a known correct outcome, needed to determine the correctness of the silicon prototype's output. Thus, to find the correct responses, engineers turn to architectural simulators. These have much lower performance than prototype hardware, and thus vendors are often forced to employ large server farms to simulate the same random tests that they run on silicon and derive correct outputs to compare against the prototype's results [Rot00]. Nevertheless, verification with randomized programs remains a central component of the post-silicon validation process, since it often exposes many unexpected behaviors of the system missed by directed tests. Following the same rationale, designers often randomize the manner in which primary inputs of the prototype, such as external interrupts, are asserted during a test and alter the delay of messages sent to and from the processor on the system bus. Typically, this is done with custom hardware components, which reside on the same board as the prototype under test and can be programmed to exhibit a variety of behaviors [SFH<sup>+</sup>03].

In addition to the functional validation of the prototype itself, the post-silicon phase must also evaluate the compatibility of the processor with a number of deployment platforms. For this purpose, the device is plugged into a typical system board with commercially available peripherals. On this complete system the verification team runs a broad range of directed benchmarks and software applications. Examples include the boot up software of several operating systems, commercial applications, performance benchmarks, legacy software, and so on. The operating conditions of the prototype in this case, are not as stressful as with randomized tests, however, these tests are still necessary to establish system compatibility throughout hours of actual runtime.

Identifying an erroneous behavior in functional post-silicon validation is only the beginning of a long and arduous journey to determine the root cause of the problem and fix it. As with electrical defects, the process begins with trying to reproduce

the bug and determine the conditions under which it occurs. Throughout this activity, DFT techniques, discussed above, play a vital role, since with these solutions designers can sample the internal state of the prototype and unwind the events that caused erroneous output behavior all the way back to the bug. In addition to generic state acquisition solutions, such as scan and OCLA, in the microprocessor industry, engineers often deploy a number of domain-specific techniques, which take advantage of the system's built-in performance counters and special interrupt modes. For instance, an external interrupt can be programmed to invoke a software routine which dumps the processor's state to memory, and then it can be asserted several times during a test execution to facilitate bug diagnosis [SFH<sup>+</sup>03]. It is important to note, however, that silicon state acquisition techniques such as this may alter the timing of events occurring within the processor, obfuscating the error.

Once a set of traces leading to an error is obtained, engineers can go back to pre-silicon techniques to reproduce and identify the bug in simulation or through formal methods. Since pre-silicon simulation has much lower performance than a prototype, full replay of long traces – typical of post-silicon runs – cannot be completed in reasonable time. Two possible remedies to this problem are trace minimization, a technique that can shorten a trace while still exposing the bug under study, *e.g.*, [CBM07b, SVM07], or the identification of a subset of events necessary for the occurrence of the error, typically accomplished by state analysis through DFT structures. A shorter trace can then be replayed in simulation, perhaps at full chip-level first and then at block granularity, and the root cause of the issue can be established. At this point formal tools may also be used to identify conditions under which the error manifest, as well as error location in an RTL design description [CWBM07]. The intermediate states that the design enters on its path from the reset condition to the bug can be used for guidance, in practice decomposing the problem into smaller subproblems, each solvable formally [HTB<sup>+</sup>09]. In addition to pinpointing the error location, formal analysis can be used to suggest the way the issue can be corrected to restore proper design functionality [SV06, CBM07a]. Finally, when the issue is diagnosed, designers propose corrections that are evaluated in pre-silicon first and then put into new versions of the prototype. Because silicon manufacturing is a costly and time consuming process, engineers always strive to develop work-arounds for a problem so to continue the validation of a prototype as long as possible without the need for a *re-spin*, that is a new tape-out. Moreover, non-critical bugs not get fixed through hardware design modifications, and instead may be simply documented with the processor's *errata* or be overcome through other non-hardware solutions.

Despite a wide range of solutions available in this domain, post-silicon validation remains a difficult and expensive activity. Processor vendors are forced to invest into such expensive hardware as high-speed logic analyzers, optical probing machines and simulation server farms, in addition to employing large teams of professional verification engineers. As a consequence, design teams have traditionally relied on high-quality pre-silicon verification to expose the majority of errors in a processor. Unfortunately, with the growing complexity of today's processors and lagging speeds of RTL simulation and verification, this is becoming harder and harder to



accomplish. Novel technologies, such as hyper-threading, multi-core architectures, variable-delay on-die interconnect and virtualization, exacerbate the problem further, stretching to the limit pre-silicon verification capabilities of even the largest processor vendors. Post-silicon functional verification has been called in to pick up the slack, and as a consequence, the fraction of design costs devoted to post-silicon validation has been increasing steadily in recent years [Yer06]. In the future, we expect this trend to continue, and anticipate the appearance of novel and exciting processor features that advance the capabilities of post-silicon validation.

## 2.3 Runtime Verification

In the previous sections we covered the main pre- and post-silicon validation steps, occurring behind the doors of the design house, before a processor is released to the market. With these techniques, the engineering team detects the vast majority of design bugs. However, a very small fraction may, and often does, escape into the final silicon product. The reasons for this lack of completeness lie in the inherent limitation of pre- and post-silicon technology: formal pre-silicon approaches are not scalable to designs as large as today's microprocessors and require that device's specifications and verification goals are carefully crafted as mathematical expressions - a feat unachievable in many cases. Simulation-based pre-silicon techniques, on the other hand, can be applied to significantly larger designs, but, in return, they provide much lower levels of correctness guarantees. In other words, simulation can only vouch for the error-free execution of cases that were exercised during the test execution and, consequently, bugs can be left lurking in the areas of a design unexplored by the tests. Furthermore, as the design is taken from its architectural description to RTL, the level of detail increases, reducing simulation performance by orders of magnitude. As a result, only fairly short test sequences can be executed on a full-system RTL model, before the design is manufactured. Post-silicon validation exposes a silicon prototype to much harsher and longer testing, revealing additional bugs that could not be discovered before manufacturing, such as electrical and manufacturing defects, as well as hard-to-reach functional bugs. The crucial shortcoming of this development phase is the lack of observability of the device's internal signals and state. While silicon state acquisition methods alleviate this process, hardware prototype diagnosis and debugging remains to this day one of the most challenging design tasks.

Overall, both pre- and post-silicon validation can expose the vast majority of bugs in a modern microprocessor, but, unfortunately, they cannot identify all of them in a reasonable time. Thus, there is always a small number of *escaped bugs*, which find their way into the final silicon product shipped to end-users. Sometimes vendor companies make information about such errors available, publishing specification updates or *errata sheets*. Typically, these escaped bugs occur rarely - which is why they are very hard to find - and may include cases of unanticipated interactions between on-chip components or between a processor and the peripherals connected

to it. Nevertheless, they can cause significant damage to the company, which may be forced to issue a recall or delay the release of the product. More importantly, unlike buggy software, circuits printed onto a silicon substrate cannot be easily modified to correct an issue, especially if they reside in millions of computers distributed all over the world. Therefore, processor vendors and researchers in academia alike have begun developing solutions to guarantee correct operation of their products in the field, even in presence of hardware bugs, with so-called *dynamic*, or *runtime verification* techniques.

The most primitive form of runtime verification consists of modifying any software application running on the hardware platform to avoid the occurrence of the erroneous sequence. This method is referred to as *instruction patching* and it is most commonly used in the embedded domain, where both hardware and the lower-level software, such as drivers and system libraries, are strictly controlled by the same vendor. In general purpose computing platforms, however, it is practically impossible to rewrite all possible software applications that a buggy processor may execute once in the field. In these platforms, engineers attempt to disable the functional blocks responsible for the issue, whenever possible. This can be accomplished through basic input-output system (BIOS) re-configuration. A BIOS is bootable firmware program stored in non-volatile memory on the motherboard of the system, the program executes at computer startup to configure the system before an operating system is loaded. To allow in-the-field system reconfiguration (through component disabling), designers augment some blocks of the processor with special programmable flags, sometimes known as *chicken bits*. If a bug is discovered that can be isolated to one of this logic blocks, the manufacturer can re-write the BIOS firmware so to disable it at startup, and thus fix the issue. However, disabling a hardware module will often lead to lower system performance or, possibly, a reduced feature set. For instance, in a recent AMD's Phenom processor a look-aside buffer had to be disabled in the cache to avoid a race condition that would hang the whole system. The consequence was a 10% performance drop [AMD08], an amount sufficient to cause a notable financial impact to the manufacturer. An alternative solution that exercises more fine-grain control on the design's features, and, therefore, has lower performance penalty, was developed at Intel after the FDIV error fiasco [SC04]. This technique, called *microcode update*, allows the processor to change the semantics of execution of individual ISA instructions. Essentially, the decode stage of a processor pipeline is made programmable and, through firmware updates, it becomes possible to re-program the execution semantics of individual instructions. At runtime, when instructions leading to a bug are fetched and decoded, the execution sequence for those instruction is replaced based on the firmware information and the occurrence of the bug can be circumvented. Note that, similar to BIOS firmware update, microcode updates must be developed offline by the processor vendor company after the escaped error is discovered in a released product and the root cause of the bug is diagnosed.

While microcode update technology faithfully served the industry in the last decade, it was not without problems of its own. For instance, AMD's implementation of this technology itself was found at some point to contain an uncorrectable

bug [AMD03]. More importantly, as the complexity of processors increased and multi-core designs appeared on the consumer market, this relatively simple technology became insufficient to detect and patch errors arising from complex interactions between on-die modules. Subsequently, researchers in academia have begun investigating approaches that enable error patching at the level of individual control logic signals [WBA06, SNC<sup>+</sup>07]. These techniques augment the processor with programmable pattern matchers, which monitor control logic signals of the device and compare them with bug patterns loaded at startup. When a match occurs, an error recovery mechanism is triggered to bypass the issue. Patterns describing the bugs, also called bug signatures or fingerprints, are created and distributed by processor vendors similarly to BIOS and microcode updates.

The most important drawback of all patching-based approaches, however, is the fact that a significant amount of time passes between when a bug is first discovered and reported and when a patch is released. During this time the error could be exploited by malicious users to breach the security of a computer system or render it inoperable. Therefore, in many applications it is desirable to be able to guarantee the correctness and security of a processor's operation even in presence of unknown escaped errors. Traditionally, in domains where errors cannot be tolerated, such as aero-space, or military applications, this goal has been accomplished through N-modular redundancy design, where multiple distinct implementations of the same hardware are deployed together and operate concurrently. All modules in this case are derived from a same specification, but are implemented independently and, therefore, are unlikely to contain identical errors. At runtime, the modules operate concurrently and errors are detected (and possibly corrected) when results differ among the various implementations. Although this redundancy may provide fairly good protection from errors, its cost and area overheads are far beyond what can be afforded in commercial processors. As a result, in recent years, researchers have proposed a number of novel, low cost, runtime verification solutions. Many of these solutions are based on hardware checkers, small blocks encoding invariants of correctness for the design's operation: for instance they can be used to dynamically validate the operation of individual cores [Aus00, DBB07] or the communication between them [MS05]. Other solutions have also proposed the insertion of local checkers distributed throughout the design to monitor the correctness of functional invariants [NdPC<sup>+</sup>04, BM05]. When an error is detected, a recovery mechanism is invoked, so the system can circumvent the troublesome spot and then return to normal execution. The key difference between checker- and patching-based solutions is that the former encode the correct scenarios of processor operation, while the latter compare the state of the design with known erroneous patterns. Typically, checker-based techniques incur higher area penalty; on the other hand, they ensure operation of the system even in the presence of undiscovered bugs.

Overall, runtime verification is a new and exciting field in processor design and validation, which promises to dramatically improve the quality of commercial processors and protect both vendors and end-users from escaped errors. Solutions in this area have just began to appear in the research literature and are yet to be fully embraced by the industry. However, we believe that the growing complexity of modern

processors will inevitably lead to the deployment of runtime verification techniques into commercial designs. Nevertheless, they will not become a substitute for pre- and post-silicon validation efforts. Each one of these domains has unique and irreplaceable advantages, and it is most likely that all of them will come together into a balanced solution, complementing each other and providing utmost efficiency in microprocessor validation.

## 2.4 Summary

In this chapter we presented a broad overview of verification techniques at all levels of a processor's life-cycle: from pre-silicon to in-the-field deployment. As we reported, verification goes hand in hand with the design process, and often must quickly adapt to modification in the specifications of a device. In pre-silicon verification, which is conducted on a range of software models of the design, simulation-based techniques remain the workhorse of the the industry, primarily, due to their scalability. On the other hand, formal verification, where a processor is subjected to rigorous mathematical proofs, does not scale to large designs, but can provide significantly higher guarantees of correctness. Therefore, formal tools are often used to selectively target relatively small internal modules of a processor that are more prone to errors and are of vital importance to the functionality of the device.

Post-silicon validation commences once a prototype of the processor is manufactured and targets electrical, functional and manufacturing defects. The latter ones are often diagnosed with efficient ATPG solutions, which subject the prototype to a variety of structural test vectors. To achieve efficient testing and debugging in this domain, engineers gain access to the internal state of a prototype through a range of DFT mechanisms, such as scan-chains, JTAG, *etc.* Functional post-silicon validation, on the other hand, aims to establish that a design truly adheres to its original specification and operates properly in a wide range of computing platforms. In particular, functional post-silicon validation in today's industry consists of executing a mix of directed and randomized tests on the hardware prototype, and then validating the corresponding outcomes against those produced by an architectural simulator or checking for a pre-designed result.

When the prototype is deemed sufficiently stable in the post-silicon phase, it is released to the market and deployed in the field. Due to short production schedules in today's competitive market, which often result in insufficient pre- and post-silicon validation, processors are frequently released with a small number of subtle bugs. To bypass such errors in the field engineers make their designs re-configurable through BIOS and firmware patches, or augment them with hardware checkers to monitor execution correctness, in addition to providing recovery mechanisms upon bug detection. Each of the three classes of verification solutions, pre-silicon, post-silicon and runtime, has unique advantages and drawbacks and therefore, all of them must be used in concert to achieve the highest level of protection from errors and to ensure minimum performance, cost and silicon area impact.

## References

- [AAF<sup>+</sup>04] Allon Adir, Eli Almog, Laurent Fournier, Eitan Marcus, Michal Rimón, Michael Vinov, and Avi Ziv. Genesys-pro: Innovations in test program generation for functional processor verification. *IEEE Design & Test of Computers*, 21(2):84–93, 2004.
- [ABF94] Miron Abramovici, Melvin A. Breuer, and Arthur D. Friedman. *Digital Systems Testing and Testable Design*. Wiley-IEEE Press, revised edition, September 1994.
- [Acc04] Accellera. *Property Specification Language Reference Manual, Rev 1.1*, June 2004. <http://www.eda.org/vfv/docs/PSL-v1.1.pdf>.
- [AMD03] Advanced Micro Devices, Inc. *AMD Athlon™ Processor Model 6 Revision Guide*, October 2003. [http://www.amd.com/us-en/assets/content\\_type/white\\_papers\\_and\\_tech\\_docs/24332.pdf](http://www.amd.com/us-en/assets/content_type/white_papers_and_tech_docs/24332.pdf).
- [AMD08] Advanced Micro Devices, Inc. *Revision Guide for AMD Family 10h Processors*, April 2008. [http://www.amd.com/us-en/assets/content\\_type/white\\_papers\\_and\\_tech\\_docs/41322.PDF](http://www.amd.com/us-en/assets/content_type/white_papers_and_tech_docs/41322.PDF).
- [AMS08] Charles J. Alpert, Dinesh P. Mehta, and Sachin S. Sapatnekar. *Handbook of Algorithms for Physical Automation*. Taylor and Francis, first edition, 2008.
- [Arv03] Arvind. Bluespec: A language for hardware design, simulation, synthesis and verification. In *MEMOCODE, Proceedings of the ACM/EEE International Conference on Formal Methods and Models for Co-Design*, page 249, June 2003.
- [Aus00] Todd Austin. DIVA: A dynamic approach to microprocessor verification. *Journal of Instruction-Level Parallelism*, 2:1–26, May 2000.
- [BA00] Michael Bushnell and Vishwanti Agrawal. *Essentials of Electronic Testing for Digital, Memory and Mixed-signal VLSI Circuits*. Springer, 2000.
- [BA08] David Burger and Todd Austin. The SimpleScalar toolset, version 3.0, 2008. <http://simplescalar.com>.
- [BAWR07] Jayanta Bhadra, Magdy S. Abadir, Li-C. Wang, and Sandip Ray. A survey of hybrid techniques for functional verification. *IEEE Design and Test of Computers*, 24(2):112–122, March 2007.
- [BBB<sup>+</sup>87] Randal E. Bryant, Derek L. Beatty, Karl Brace, Kyeongsoon Cho, and Thomas Shaffer. COSMOS: A compiled simulator for MOS circuits. In *DAC, Proceedings of the Design Automation Conference*, pages 9–16, June 1987.
- [BBS90] Derek L. Beatty, Randal E. Bryant, and Carl-Johann H. Seger. Synchronous circuit verification by symbolic simulation: An illustration. In *Proceedings of Sixth MIT conference on advanced research in VLSI*, pages 98–112, April 1990.
- [Ben01] Bob Bentley. Validating the Intel® Pentium® 4 microprocessor. In *DAC, Proceedings of the Design Automation Conference*, pages 224–228, June 2001.
- [Ber05] Valeria Bertacco. *Formal Verification Scalable Hardware Verification With Symbolic Simulation*. Springer, first edition, 2005.
- [BLL<sup>+</sup>04] Michael Behm, John Ludden, Yossi Lichtenstein, Michal Rimón, and Michael Vinov. Industrial experience with test generation languages for processor verification. *DAC, Proceedings of the Design Automation Conference*, pages 36–40, June 2004.
- [BM05] Ali A. Bayazit and Sharad Malik. Complementary use of runtime validation and model checking. In *ICCAD, Proceedings of the International Conference on Computer Aided Design*, pages 1052–1059, 2005.
- [Boc07] Bochs: The open source IA-32 emulation project, September 2007. <http://bochs.sourceforge.net/>.
- [BRB90] Karl Brace, Richard Rudell, and Randal E. Bryant. Efficient implementation of a BDD package. In *DAC, Proceedings of the Design Automation Conference*, pages 40–45, June 1990.
- [Bry85] Randal E. Bryant. Symbolic verification of MOS circuits. In *Proceedings of Chapel Hill Conference on VLSI*, pages 419–438, May 1985.
- [Bry86] Randal E. Bryant. Graph-based algorithms for Boolean function manipulation. *IEEE Transactions on Computers*, 35:677–691, 1986.

- [CBM89] Oliver Coudert, Christian Berthet, and Jean C. Madre. Verification of synchronous sequential machines based on symbolic execution. In *Proceedings of the International Workshop on Automatic Verification Methods for Finite State Systems*, pages 365–373, June 1989.
- [CBM07a] Kai-hui Chang, Valeria Bertacco, and Igor Markov. Automating post-silicon debugging and repair. In *ICCAD, Proceedings of the International Conference on Computer Aided Design*, pages 91–98, November 2007.
- [CBM07b] Kai-hui Chang, Valeria Bertacco, and Igor Markov. Simulation-based bug trace minimization with BMC-based refinement. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, 26(1):152–165, 2007.
- [CBRZ01] Edmund M. Clarke, Armin Biere, Richard Raimi, and Yunshan Zhu. Bounded model checking using satisfiability solving. *Formal Methods in System Design*, 19(1):7–34, 2001.
- [CGP08] Edmund M. Clarke, Orna Grumberg, and Doron A. Peled. *Model Checking*. MIT Press, sixth edition, 2008.
- [CJB79] William C. Carter, William H. Joyner, and Daniel Brand. Symbolic simulation for correct machine design. In *DAC, Proceedings of the Design Automation Conference*, pages 280–286, June 1979.
- [CKM<sup>+</sup>04] William D. Corti, Robert Kenny, Joseph O. Marsh, Steven C. Parker, Frank X. Scanzano, and Michael Won. *U.S. Patent no. 6834360: On-chip logic analyzer*. International Business Machines Corporation, December 2004.
- [CVK04] Ben Cohen, Srinivasan Venkataramanan, and Ajeetha Kumari. *Using PSL/Sugar for Formal and Dynamic Verification*. VhdlCohen Publishing, second edition, 2004.
- [CWBM07] Kai-hui Chang, Ilya Wagner, Valeria Bertacco, and Igor Markov. Automatic error diagnosis and correction for RTL designs. In *HLDVT, Proceedings of the International Workshop on High Level Design Validation and Test*, pages 65–72, November 2007.
- [DBB07] Andrew DeOrio, Adam Bauserman, and Valeria Bertacco. Chico: An on-chip hardware checker for pipeline control logic. In *MTV, Proceedings of the International Workshop on Microprocessor Test and Verification*, pages 91–97, December 2007.
- [DP60] Martin Davis and Hilary Putnam. A computing procedure for quantification theory. *Journal of the Association for Computing Machinery*, 7(3):201–215, July 1960.
- [ES03] Niklas Een and Niklas Sorensson. An extensible SAT-solver. In *Theory and Applications of Satisfiability Testing*, volume 2919 of *Lecture Notes in Computer Science*, pages 502–518. Springer, 2003.
- [EWR00] Travis Eiles, Gary Woods, and Valluri Rao. Optical probing of flip-chip-packaged microprocessors. In *ISSCC, Proceedings of the International Solid State Circuits Conference*, pages 220–221, February 2000.
- [FKL04] Harry Foster, Adam Krolnik, and David Lacey. *Assertion-based design*. Springer, second edition, 2004.
- [FZ03] Shai Fine and Avi Ziv. Coverage directed test generation for functional verification using Bayesian networks. In *DAC, Proceedings of the Design Automation Conference*, pages 286–281, June 2003.
- [Glu06] Alon Gluska. Practical methods in coverage-oriented verification of the Merom microprocessor. In *DAC, Proceedings of the Design Automation Conference*, pages 332–337, July 2006.
- [GTSK05] Mehmet H. Gunes, Mitchell Thornton, Fatih Kocan, and Stephen Szygenda. A and comparison of digital logic simulators. In *MWSCAS, Proceedings of the Midwest Symposium on Circuits and Systems*, pages 744–749, August 2005.
- [HKM01] Faisal I. Haque, Khizar A. Khan, and Jonathan Michelson. *The Art of Verification with Vera*. Verification Central, first edition, 2001.
- [HMN01] Yoav Hollander, Matthew Morley, and Amos Noy. The  $\epsilon$  language: A fresh separation of concerns. In *TOOLS, Proceedings of the, International Conference on Technology of Object-Oriented Languages*, pages 41–50, March 2001.



- [HS06] Gary D. Hachtel and Fabio Somenzi. *Logic Synthesis and Verification Algorithms*. Springer, first edition, 2006.
- [HTB<sup>+</sup>09] Richard Ho, Michael Theobald, Brannon Batson, J.P. Grossman, Stanley C. Wang, Joseph Gagliardo, Martin M. Deneroff, Ron O. Dror, and David E. Shaw. Post-silicon debug using formal verification waypoints. In *DVCon, Proceedings of the Design and Verification Conference and Exhibition*, pages 1–7, February 2009.
- [IEE01a] Institute of Electrical and Electronics Engineers. *Standard test access port and boundary-scan architecture. IEEE Std. 1149.1-2001.*, 2001. <http://ieeexplore.ieee.org/servlet/opac?punumber=7481>.
- [IEE01b] Institute of Electrical and Electronics Engineers. *Standard Verilog hardware description language. Std 1364-2001.*, 2001. <http://ieeexplore.ieee.org/servlet/opac?punumber=7578>.
- [IEE04] Institute of Electrical and Electronics Engineers. *Behavioural languages - Part 1-1: VHDL language reference manual. IEC 61691-1-1 First edition 2004-10; IEEE 1076.*, 2004. <http://ieeexplore.ieee.org/servlet/opac?punumber=9649>.
- [IEE07] Institute of Electrical and Electronics Engineers. *Standard for SystemVerilog - Unified Hardware Design, Specification, and Verification Language. Std 1800-2007*, 2007. <http://ieeexplore.ieee.org/servlet/opac?punumber=4410438>.
- [JG04] Doug Josephson and Bob Gottlieb. The crazy mixed up world of silicon debug. In *CICC, Proceedings of the IEEE Custom Integrated Circuits Conference*, pages 665–670, October 2004.
- [JPG01] Doug Josephson, Steve Poehhnan, and Vincent Govan. Debug methodology for the McKinley processor. In *ITC, Proceedings of the International Test Conference*, pages 451–460, October 2001.
- [KDF<sup>+</sup>04] Ravishankar Kuppuswamy, Peter DesRosier, Derek Feltham, Rehan Sheikh, and Paul Thadikaran. Full hold-scan systems in microprocessors: Cost/benefit analysis. *Intel Technology Journal*, 08:63–72, February 2004.
- [KG99] Christoph Kern and Mark R. Greenstreet. Formal verification in hardware design: a survey. *ACM Transactions on Design Automation of Electronic Systems*, 4(2):123–193, 1999.
- [KK94] Ramayya Kumar and Thomas Kropf. *Theorem provers in circuit design: theory, practice, and experience*. Springer Berlin/ Heidelberg, first edition, 1994.
- [Lam02] Leslie Lamport. *Specifying systems: the TLA+ language and tools for hardware and software engineers*. Addison-Wesley Professional, 2002.
- [Lit02] Timothe Litt. Support for debugging in the Alpha 21364 microprocessor. In *ITC, Proceedings of the International Test Conference*, pages 584–589, October 2002.
- [MCE<sup>+</sup>02] Peter S. Magnusson, Magnus Christensson, Jesper Eskilson, Daniel Forsgren, Gustav Hallberg, Johan Hogberg, Frederik Larsson, Andreas Moestedt, and Bengt Werner. Simics: A full system simulation platform. *IEEE Computer*, 35(2):50–58, 2002.
- [Mei93] Gerd Meister. A survey on parallel logic simulation. Technical report, University of Saarland, Department of Computer Science, 1993.
- [Men08] Mentor Graphics®Inc. *ModelSim - a comprehensive simulation and debug environment for complex ASIC and FPGA designs*, 2008. <http://www.model.com/>.
- [Mic03] Alexander Miczo. *Digital logic testing and simulation*. John Wiley and Sons, second edition, 2003.
- [MS05] Albert Meixner and Daniel J. Sorin. Dynamic verification of sequential consistency. *SIGARCH Computer Architecture News*, 33(2):482–493, 2005.
- [MSB<sup>+</sup>05] Milo M. K. Martin, Daniel J. Sorin, Bradford M. Beckmann, Michael R. Marty, Min Xu, Alaa R. Alameldeen, Kevin E. Moore, Mark D. Hill, and David A. Wood. Multifacet’s general execution-driven multiprocessor simulator (GEMS) toolset. *SIGARCH Computer Architecture News*, 33(4):92–99, 2005.



- [MSS99] Joao P. Marques-Silva and Karem A. Sakallah. GRASP: a search algorithm for propositional satisfiability. *IEEE Transactions on Computers*, 48(5):506–521, 1999.
- [NdPC<sup>+</sup>04] Jose A. Nacif, Flavio M. de Paula, Claudionor N. Coelho, Fernando C. Sica, Harry Foster, Antonio O. Fernandes, and Diogenes C. da Silva. The Chip is Ready. Am I done? On-chip Verification using Assertion Processors. In *Symposium on Integrated Circuits and System Design*, pages 55–59, September 2004.
- [PBG05] Mukul R. Prasad, Armin Biere, and Aarti Gupta. A survey of recent advances in SAT-based formal verification. *STTT, Journal on Software Tools for Technology Transfer*, 2:156–173, April 2005.
- [PF05] Douglas L. Perry and Harry Foster. *Applied Formal Verification: For Digital Circuit Design*. McGraw-Hill Professional, first edition, 2005.
- [Piz04] Andrew Piziali. *Functional verification coverage measurement and analysis*. Springer, first edition, 2004.
- [Rot00] Hemant Rotithor. Post-silicon validation methodology for microprocessors. *IEEE Design and Test of Computers*, 17(4):77–88, October 2000.
- [RS95] Kavita Ravi and Fabio Somenzi. High-density reachability analysis. In *ICCAD, Proceedings of the International Conference on Computer Aided Design*, pages 154–158, February 1995.
- [SC04] Tom Shanley and Bob Colwell. *The Unabridged Pentium 4: IA32 Processor Genealogy*. Addison-Wesley Professional, illustrated edition, 2004.
- [SFH<sup>+</sup>03] Isic Silas, Igor Frumkin, Eilon Hazan, Ehud Mor, and Genadiy Zobin. System-level validation of the Intel® Pentium® M processor. *Intel Technology Journal*, 07:38–43, May 2003.
- [SKC95] Bart Selman, Henry Kautz, and Bram Cohen. Local search strategies for satisfiability testing. In *DIMACS Series in Discrete Mathematics and Theoretical Computer Science*, pages 521–532, 1995.
- [SNC<sup>+</sup>07] Smruti Sarangi, Satish Narayanasamy, Bruce Carneal, Abhishek Tiwari, Brad Calder, and Josep Torrellas. Patching processor design errors with programmable hardware. *IEEE Micro*, 27(1):12–25, 2007.
- [Som09] Fabio Somenzi. *CUDD: CU Decision Diagram Package*, 2009. <http://vlsi.Colorado.edu/~fabio/CUDD>.
- [SV06] Sean Safarpour and Andreas Veneris. Abstraction and refinement techniques in automated design debugging. In *MTV, Proceedings of the International Workshop on Microprocessor Test and Verification*, pages 88–93, December 2006.
- [SVM07] Sean Safarpour, Andreas Veneris, and Hratch Mangassarian. Trace compaction using SAT-based reachability analysis. In *ASP-DAC, Proceedings of the Asian-South Pacific Design Automation Conference*, pages 932–937, January 2007.
- [Syn09] Synopsys® Inc. *VCS: Comprehensive RTL Verification Solution*, 2009. <http://www.synopsys.com/tools/verification/functionalverification/pages/vcs.aspx>.
- [VR05] Srikanth Vijayaraghavan and Meyyappan Ramanathan. *A practical guide for system Verilog assertions*. Springer, illustrated edition, 2005.
- [WBA06] Ilya Wagner, Valeria Bertacco, and Todd Austin. Shielding against design flaws with field repairable control logic. In *DAC, Proceedings of the Design Automation Conference*, pages 344–347, July 2006.
- [WBA07] Ilya Wagner, Valeria Bertacco, and Todd Austin. Microprocessor verification via feedback-adjusted Markov Models. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, 26(6):1126–1138, 2007.
- [WGR05] Bruce Wile, John C. Goss, and Wolfgang Roesner. *Comprehensive functional verification the complete industry cycle*. Morgan Kaufmann, illustrated, annotated edition, 2005.
- [Yer06] Siva Yerramilli. On the need for convergence between design validation and test. In *ITC, Proceedings of the International Test Conference*, page 14, October 2006.
- [YPA06] Jun Yuan, Carl Pixley, and Adnan Aziz. *Constraint-Based Verification*. Springer, first edition, 2006.

Post-Silicon and Runtime Verification for Modern  
Processors

Wagner, I.; Bertacco, V.

2011, XVII, 224 p., Hardcover

ISBN: 978-1-4419-8033-5