

Chapter 2

DESIGN AND VERIFICATION OF DIGITAL SYSTEMS

Before delving into the discussion of the various verification techniques, we are going to review how digital ICs are developed. During its development, a digital design goes through multiple transformations, from the original set of specifications to the final product. Each of these transformations corresponds, coarsely, to a different description of the system, which is incrementally more detailed and which has its own specific semantics and set of primitives. This chapter provides a high-level overview of this design flow in the first two sections. We then review the mathematical background (Section 2.3) and cover the basic circuit structure and finite state machine definitions (Section 2.5) that are required to present the core algorithms involved in verification.

The remaining sections present the algorithms that are at the core of the current technology in design verification. Section 2.6 presents the approach of *compiled-level logic simulation*. This technique was first introduced in the late 80's and it is still today the industry's mainstream verification approach. Section 2.7 provides an overview of formal verification and a few of the solutions in this space; we leave the discussion of symbolic simulation and other symbolic techniques to Chapter 3.

2.1 The design flow

Figure 2.1 presents a conceptual design flow from the specifications to the final product. The flow in the figure shows a top-down approach that is very simplified – as we discuss later in this section, the reality of an industrial development is much more complex, involving many iterations through various portions of the flow in the figure, until the final design converges to a form that meets the requirements of functionality, area, timing, power and cost. The design specifications are generally presented as a document describing a set of functionalities that the final solution will have to provide and a set constraints

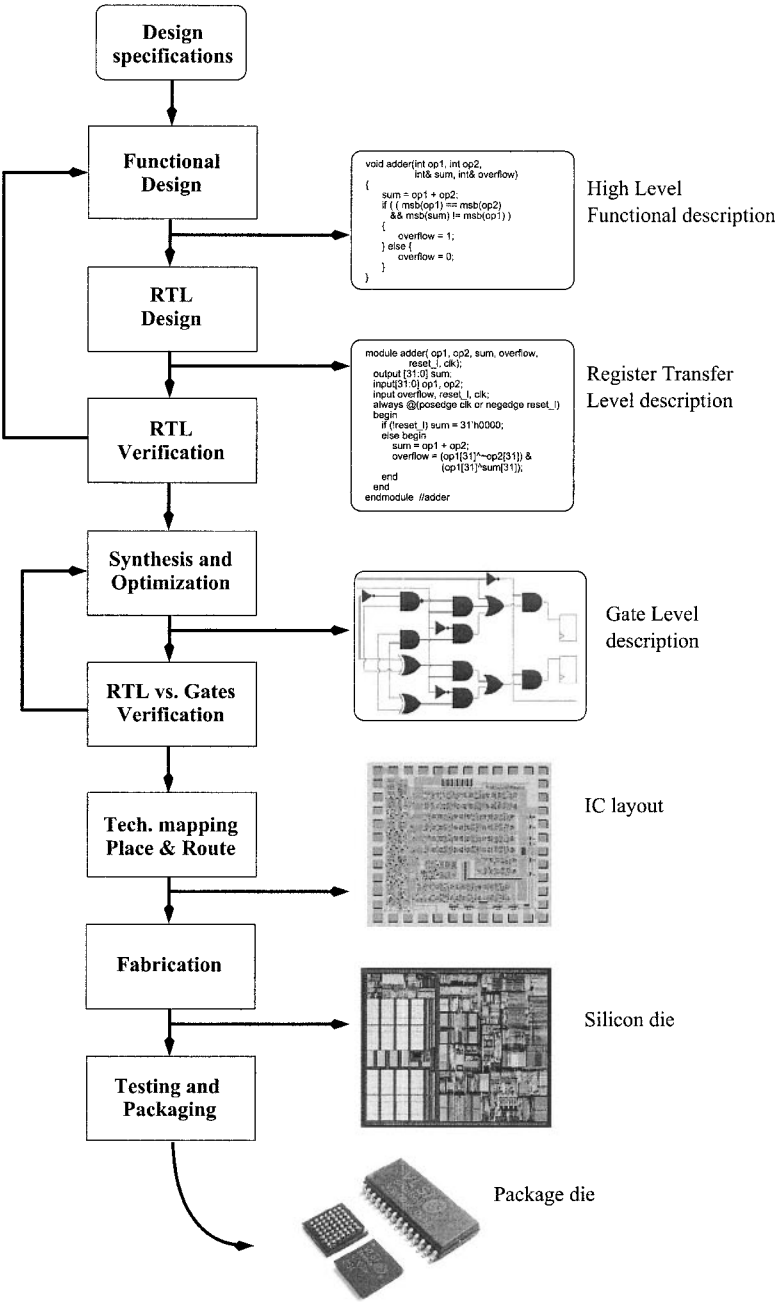


Figure 2.1: Conceptual design flow of a digital system

that it must satisfy. In this context, the *functional design* is the initial process of deriving a potential and realizable solution from these specifications and requirements. This is sometimes referred to as *modeling* and includes such activities as hardware/software tradeoffs and micro-architecture design.

Because of the large scale of the problem, the development of a functional design is usually carried out using a hierarchical approach, so that a single designer can concentrate on a portion of the model at any given time. Thus, the architectural description provides a partition of the design in distinct modules, each of which contributes a specific functionality to the overall design. These modules have well-defined input/output interfaces and protocols for communicating with the other components of the design. Among the results of this design phase is a high-level functional description, often a software program in C or in a similar programming language, that simulates the behavior of the design with the accuracy of one clock cycle and reflects the module partition. It is used for performance analysis and also as a reference model to verify the behavior of the more detailed designs developed in the following stages.

From the functional design model, the hardware design team proceeds to the *Register Transfer Level (RTL)* design phase. During this phase, the architectural description is further refined: memory elements and functional components of each model are designed using a Hardware Description Language (HDL). This phase also entails the development of the clocking system of the design and architectural trade-offs such as speed and power.

With the RTL design, the functional design of our digital system ends and its verification begins. *RTL verification* consists of acquiring reasonable confidence that a circuit will function correctly, under the assumption that no manufacturing fault is present. The underlying motivation is to remove all possible design errors before proceeding to the expensive phase of chip manufacturing. Each time functional errors are found, the model needs to be modified to reflect the proper behavior. During RTL verification, the verification team develops various techniques and numerous suites of tests to check that the design behavior corresponds to the initial specifications. When that is not the case, the functional design model needs to be modified to provide the correct behavior specified and the RTL design updated consequently. It is also possible that the RTL verification phase reveals incongruous or overlooked aspects in the original set of specifications and it is found that the specification document is to be updated instead of the RTL description.

In the diagram of Figure 2.1, RTL verification appears as one isolated phase of the design flow. However, in practical designs, the verification of the RTL model is carried on in parallel with the other design activities and it often lasts until chip layout. An overview of the verification methodologies that are common in today's industrial developments is presented in the next section.

The next design phase consists of the *Synthesis and Optimization* of the RTL design. The overall result of this phase is to generate a detailed model of a circuit, which is optimized based on the design constraints. For instance, a design could be optimized for power consumption or the size of its final realization (IC area) or for the ease of testability of the final product. The detailed model produced at this point describes the design in terms of its basic logic components, such as *AND*, *OR*, *NOT* or *XOR*, in addition to memory elements. Optimizing the netlist, or gate-level description, for constraints such as timing and power requirements is an increasingly challenging aspect of current developments and it usually involves multiple iterations of trial-and-error attempts before reaching a solution that satisfies the requirements. Such optimizations may, in turn, introduce functional errors that require additional RTL verification.

All the design phases, up to this point, have minimal support from Computer-Aided Design (CAD) software tools and are almost entirely hand-crafted by the design and verification team. Consequently, they absorb a preponderant fraction of the time and cost involved in developing a digital system. Starting with synthesis and optimization, most of the activities are semi-automatic or at least heavily supported by CAD tools. Automating the RTL verification phase, is the next challenge that the CAD industry is facing in providing full support for digital systems development.

The synthesized model needs to be verified. The objective of *RTL versus gates verification*, or equivalence checking, is to guarantee that no errors have been introduced during the synthesis phase. It is an automatic activity, requiring minimal human interaction, that compares the pre-synthesis RTL description to the post-synthesis gate-level description in order to guarantee the functional equivalence of the two models.

At this point, it is possible to proceed to *technology mapping and placement and routing*. The result is a description of the circuit in terms of geometrical layout used for the fabrication process. Finally, the design is *fabricated*, and the microchips are *tested and packaged*.

This design flow is obviously a very ideal, conceptual case. For instance, usually there are many iterations of synthesis, due to changes in the specification or to the discovery of flaws during RTL verification. Each of the new synthesized versions of the design needs to be put again through all of the subsequent phases. One of the main challenges faced by design teams, for instance, is satisfying the ever-increasing market pressure to produce digital systems with better and better performance. These challenging specifications force engineering teams to push the limits of their designs by optimizing them at every level: architectural, component (optimizing library choice and sizing), placement and routing. Achieving timing closure, that is, developing a design that satisfies the timing constraints set in the specifications while still operat-

ing correctly and reliably, most often requires optimizations that go beyond the abilities of automatic synthesis tools and pushes engineers to intervene manually, at least in critical portions of the design. Often, it is only possible to check if a design has met the specification requirements after the final layout has been produced. If these requirements are not met, the engineering team must devise alternative optimizations or architectural changes and create a new design model that must be put through the complete design flow all over again.

2.2 RTL verification

As we observed in the previous section, the correctness of a digital circuit is a major consideration in the design of digital systems. Given the extremely high and increasing costs of manufacturing microchips, the consequences of flaws going unnoticed in system designs until after the production phase, are very expensive. At the same time, RTL verification, that is, verifying the correctness of an RTL description, is still one of the most challenging activities in digital system development: as of today, it is still carried on mostly with ad-hoc tests, scripts and, often, even ad-hoc tools developed by design and verification teams specifically for the present design effort. In the best scenarios, the development of this verification infrastructure can be amortized among a family of designs with similar architecture and functionality. Moreover, verification methodology still lacks any standard or even a commonly accepted plan of attack, with the consequence that each hardware engineering team has its own distinct verification practices, which often change with subsequent designs by the same team, due to the insufficient “correctness confidence-level” that any of the current approaches provide. Given this scenario, it is not only easy to see why many digital IC development teams report that more than 70% of the design time and engineering resources are spent in verification, but it is clear why verification is, thus, the bottleneck in the time-to-market odyssey for integrated circuit development [Ber03a].

The workhorse of the industrial approach to verification is *functional validation*. The functional model of a design is simulated with meaningful input stimuli and the output is then checked for the expected behavior. The model used for simulation is the RTL description. The simulation involves applying patterns of test data at the inputs of the model, then using the simulation software (or hardware) to compute the simulated values at the outputs and, finally, checking the correctness of the values obtained.

Validation is generally carried on at two levels: module level and chip level. The first verifies each module of the design independently of the other modules. It involves producing entire suites of *stand-alone tests*, each of which checks the proper behavior of one specific aspect or functionality of that module. Each test includes a set of input patterns to stimulate the module. These tests also include a portion that verifies that the output of the module corresponds to what

is expected. The design of these tests is generally very time consuming, since each of them has to be handcrafted by the verification engineering team. Moreover, their reusability is very limited because they are specific to each module. Recently, a few CAD tools have become available to support functional validation, meaning, they mainly provide more powerful and compact language primitives to describe the test patterns and to check the outputs of the module, thereby saving some test development time [HKM01, HMN01, Ber03a].

During chip-level validation, the design is verified as a whole. Often, this is done after sufficient confidence is obtained regarding the correctness of each single module. The focus is mainly in verifying the proper interaction between modules. This phase, while more computationally intensive, has the advantage of being carried on in a semi-automatic fashion. In fact, input test patterns are often randomly generated, with the only constraint being that they must be compatible with what the specification document defines to be the proper input format for the design. During chip-level validation, it is usually possible to use a golden model for verification. That is, run in parallel the simulation of both the RTL and a high-level description of the design, and check that the outputs of the two systems and the values stored in their memory elements match one-to-one at the end of each clock cycle (this is called lock-step).

The quality of all these verification efforts is usually analytically evaluated in terms of coverage: a measure of the fraction of the design that has been verified [KN96, LMUZ02]. Functional validation can provide only partial coverage because of its approach. The objective therefore is to maximize coverage for the design under test.

Various measures of coverage are in use: for instance *line coverage* counts the lines of the RTL description that have been activated during simulation. Another common metric is *state coverage*, which measures the number of all the possible configurations of a design that have been simulated (*i.e.* validated). This measure is particularly valuable when an estimate of the total-state space of the design is available. In this situation the designer can use state coverage to quantify the fraction of the design that has been verified.

With the increasing complexity of industrial designs, the fraction of design space that the functional validation approach can explore is becoming vanishingly small, indicating more and more that it is an inadequate solution to the verification problem. Since only one state and one input combination of the design under test are visited during each step of simulation, it is obvious that neither of the above approaches can keep up with the exponential growth in circuit complexity¹.

¹The state space of a system doubles for each additional state bit added. Since, as we discussed earlier, the area available doubles every 18 months, and assuming that a fixed fraction of this area is dedicated to memory elements, the overall complexity growth is exponential.

Because of the limitations of functional validation, new, alternative techniques have received increasing interest. The common trait of these techniques is the attempt to provide some type of mathematical proof that a design is correct, thus guaranteeing that some aspect or property of the circuit behavior holds under every circumstance. and, therefore, its validity is not limited only to the set of test patterns that have been checked. These techniques go under the name of *formal verification* and have been studied mostly in academic research settings for the past 25 years. Formal verification constitutes a major paradigm shift in solving the verification problem. As the qualitative sketch in Figure 2.2 shows, with logic simulation we probe the system with a few hand-crafted stimuli which are sent through the system and produce an output that must be interpreted in order to establish the correctness of the system for that specific setting. On the other hand, with formal verification the correctness of a design is shown by generating an analytical proof that the system is compatible with each of the properties derived from the specification. Compared to a functional validation approach, this is equivalent to simulating a design with all possible input stimuli, thus providing 100% coverage.

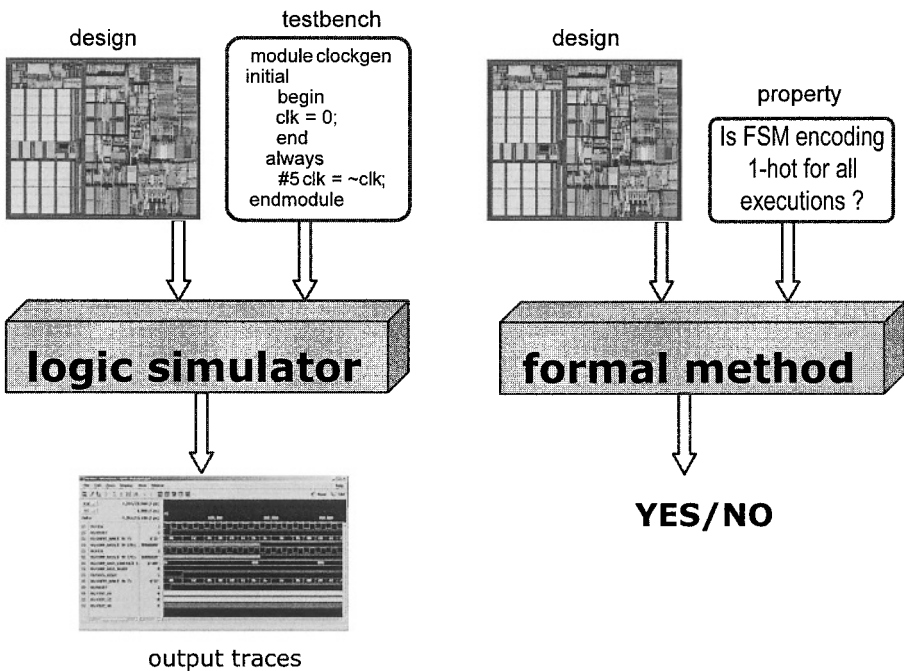


Figure 2.2: Approaches to verification: validation vs. formal verification

It is obvious that the promise of such thorough verification makes formal verification a very appealing approach. While, on one hand, the solution to the verification problem seems to lie with formal verification approaches, on the other hand, these techniques have been unable to tackle industrial designs due to the complexity of the underlying algorithms, and thus have been applicable only to smaller components. They have been used in industrial development projects only at an experimental level. So far they generally have not been part of the mainstream verification methodology.

The next sections review some of the model abstractions for digital systems in use, in the context of functional verification. We then present, in depth, an algorithm underlying the mainstream approach of functional validation: logic simulation.

2.3 Boolean functions and their representation

Boolean functions are the most common vehicle to describe the functionality of a digital block. We dedicate this section to review a few basic aspects of Boolean algebra and Boolean functions. The concepts outlined here will be referenced throughout the book.

We use the symbol \mathcal{B} to denote the Boolean algebra defined over the set $\{0, 1\}$. A **symbolic variable** is a variable defined in \mathcal{B} . A **logic function**, or Boolean function, is a mapping $F : \mathcal{B}^n \rightarrow \mathcal{B}^m$. In the attempt to ease the reading of the theoretical presentations in this book, we use lower-case letters to denote symbolic variables, and upper-case to denote functions. In addition, **scalar** functions, $F(x_1, \dots, x_n) : \mathcal{B}^n \rightarrow \mathcal{B}$ are represented by regular face literals, while vector-valued functions are represented in boldface. The majority of our presentation will be concerned with scalar functions. The i^{th} component of a vector function \mathbf{F} is indicated by F_i .

An important aspect of a logic function is its **support**, that is, the set of variables the function effectively depends on. For instance the support of the function $F(a, b, c, d) = a + b$ is $\mathcal{S}(F) = \{a, b\}$. To find which variables are in the support of a function, we need to use the concept of cofactor:

Definition 2.1. *The **1-cofactor** of a function F with respect to a variable x_i is the function F_{x_i} obtained by substituting 1 for x_i in F . Similarly, the **0-cofactor**, F_{x_i} , is obtained by substituting 0 for x_i in F .*

By computing the cofactors w.r.t. (with respect to) c for the function used in the previous example, we can easily find that $F_c = F_c = a + b$. Here is a formal definition of support:

Definition 2.2. *Let $F : \mathcal{B}^n \rightarrow \mathcal{B}$ denote a non-constant Boolean function of n variables x_1, \dots, x_n . We say that F **depends** on x_i iff $F_{x_i} \neq F_{x_i}$. We call **support** of F , indicated by $\mathcal{S}(F)$, the set of Boolean variables F depends on. In the most*

general case, when F is a vector function, we say that $\mathbf{F} : \mathcal{B}^m \rightarrow \mathcal{B}^n$ depends on a variable x_i , if at least one of its components F_i depends on it.

The **size** of $S(F)$ is the number of its elements, and it is indicated by $|S(F)|$. Two functions F, G are said to have **disjoint support** if they share no support variables, i.e. $S(F) \cap S(G) = \emptyset$. The concept of disjoint support is the core of the presentation in Chapter 4.

Another aspect of Boolean functions which is central to this entire book is **range**, that is, the co-domain spanned by a logic function. Using again our little example, the range of $F = a + b$ is $\{0, 1\}$. When discussing vector functions the concept of range becomes more meaningful, since each output value is a Boolean vector. The notion of range is relevant to our discussion for the following reason: in symbolic simulation, each cycle computes a symbolic vector to represent the next states of the design. This vector is effectively a Boolean function, say \mathbf{NS} . The next step of simulation transfers this vector to the present state and then proceeds by simulating the combinational logic of the design. However, as we point out again in later chapters, the only relevant information to be transferred among symbolic steps is the range spanned by \mathbf{NS} , because this range describes the set of states that have been visited by the simulator up to that point. The key advantage of parametrization techniques is that of considering the \mathbf{NS} vector function and devising an alternative vector, \mathbf{P} , which will span the same range but will involve smaller functions. This is a formal definition for the range of a function:

Definition 2.3. The **range** of a function $\mathbf{F} : \mathcal{B}^n \rightarrow \mathcal{B}^m$ is the set of m -tuples that can be asserted by \mathbf{F} , and is denoted by $\mathcal{R}(\mathbf{F})$:

$$\mathcal{R}(\mathbf{F}) = \{y \in \mathcal{B}^m \mid \exists x \in \mathcal{B}^n, \mathbf{F}(x) = y\} \quad (2.1)$$

For scalar functions the range reduces to $\mathcal{R}(F) = \mathcal{B}$ for all except the two constant functions 0 and 1.

A special class of functions that will be used frequently is that of *characteristic functions*. Characteristic functions are scalar functions that represent sets implicitly – they are asserted if and only if their input value belongs to the set represented. Characteristic functions can be used, for instance, to describe implicitly all the states of a system that have been explored by a symbolic technique.

Definition 2.4. Given a set $\mathcal{V} \subset \mathcal{B}^n$, whose elements are Boolean vectors, its **characteristic function** $\chi_{\mathcal{V}}(x) : \mathcal{B}^n \rightarrow \mathcal{B}$ is defined as:

$$\chi_{\mathcal{V}}(x) = \begin{cases} 1 & \text{when } x \in \mathcal{V} \\ 0 & \text{otherwise} \end{cases} \quad (2.2)$$

For example, the characteristic function of the set $S = \{00, 11\}$ is $\chi_S(x_1, x_2) = x_1x_2 + \overline{x_1}\overline{x_2}$. When sets are represented by their characteristic function, the operations of set intersection, union and set complement correspond to the *AND*, *OR* and *NOT* respectively, on their corresponding functions.

We conclude this section by defining two additional operations between Boolean functions. These are function composition and generalization of cofactor and will be useful in the presentation of the disjoint-support decomposition algorithm in Chapter 4. Its application will be discussed in Chapter 6.

Definition 2.5. *Given two functions $F(x_1, \dots, x_n)$ and $X_i(y_1, \dots, y_n)$, the **function composition** $F \circ X_i$ is the function obtained by replacing the function X_i in F for each occurrence of the variable x_i .*

Definition 2.6. *Given two functions F and G , the **generalized cofactor** of F w.r.t. G is the function F_G such that for each input combination satisfying G the outputs of F and F_G are identical.*

Notice that, in general, there are multiple possible functions F_G satisfying the definition of the generalized cofactor. Moreover, if F and G have disjoint supports, then one possible solution for F_G is the function F itself.

2.3.1 NP-equivalence

NP-equivalence, or negation-permutation equivalence, is a family of transformations among Boolean functions. They are of interest in the Computer-Aided Design world because two functions that are NP-equivalent can be realized by the same circuit. The implementation of the two NP-equivalent functions will differ only in the mapping of the function's inputs on the inputs of the circuit, and in the need of complementing some of the circuit's inputs. The difficulty, however, lies in identifying when two functions are NP-equivalent. Canonical data structures, such as BDDs (see Section 2.4) provide many benefits by recognizing easily when two functions are identical. On the other hand, two NP-equivalent functions may have completely different BDD representations and can be, therefore, very hard to identify.

An NP-function is a function that can be connected in front of another function F to generate a function G that is NP-equivalent to F [BL92, MD93]:

Definition 2.7. *A function $\mathbf{F}(x_1, \dots, x_n): \mathcal{B}^n \rightarrow \mathcal{B}^n$ is termed an **NP-function** if, for each of its components F_i , either $F_i = x_j$ or $F_i = \overline{x_j}$ for some j and $\mathcal{S}(F_i) \cap \mathcal{S}(F_k) = \emptyset, i \neq k$.*

In other words, an NP-function can only permute and/or complement its inputs.

Definition 2.8. Two functions $F(x_1, \dots, x_n)$ and $G(x_1, \dots, x_n)$ are said to be **NP-equivalent** if there is a NP-function $\text{NP}(x_1, \dots, x_n)$ such that

$$F(x_1, \dots, x_n) = G(\text{NP}(x_1, \dots, x_n)) \quad (2.3)$$

that is, F is obtained by composing G with NP .

2.4 Binary decision diagrams

Binary Decision Diagrams (BDDs) are a compact and efficient way of representing and manipulating Boolean functions. Because of this, BDDs are a key component of all symbolic techniques of verification. They form a canonical representation, making the testing of functional properties, such as satisfiability and equivalence, straightforward. BDDs are *directed acyclic graphs* that satisfy a few restrictions for canonicity and compactness. BDDs have two terminal nodes, 0 and 1, all other internal nodes are labeled by a symbolic variable and have two outgoing edges: one corresponding to the 0-cofactor w.r.t. the variable labeling the node, and one corresponding to the 1-cofactor. Each path from root to leaves, in the graph, corresponds to an evaluation of the Boolean function for a specific assignment of its input variables. An important factor in the success of BDDs is the ease of manipulating Boolean functions through this representation: the *apply operation* is implemented by a simple recursive function which traverses one or more BDDs (the operands) bottom-up and applies a specified Boolean operator at each intermediate node. We provide here a brief presentation. The interested reader is referred to [Bry86, Bry92] for an in-depth introduction and an overview of their applications.

Example 2.1. Figure 2.3.a represents the BDD for the function $F = (\bar{x} + \bar{y})pq$. Given any assignment for the four input variables it is possible to find the value of the function by following the corresponding path from the root F to a leaf. At each node, the 0-edge (dashed) is chosen if the corresponding variable has a value 0, the 1-edge otherwise.

Figure 2.3.b represents the BDD for the function $G = w \oplus x \oplus y \oplus z$. Observe that the number of BDD nodes needed to represent XOR functions with BDDs, is $2 \cdot \#vars$. At the same time, other canonical representations, such as truth tables or sum of minterms require a number of terms that is exponential with respect to the number of variables in the function's support.

For a given ordering of the variables, it was shown in [Bry86] that a function has a unique BDD representation. Therefore, checking the identity of two functions corresponds to checking for BDD identity, which is accomplished in constant time. The following definition formalizes the structure of BDDs:

Definition 2.9. A BDD is a DAG with two sink nodes labeled “0” and “1” representing the Boolean functions **0** and **1**. Each non-sink node is labeled

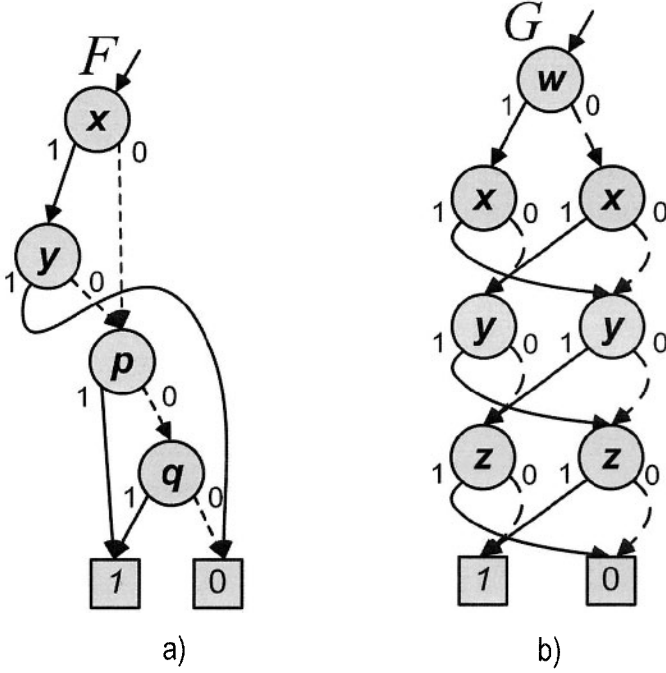


Figure 2.3: Examples of binary decision diagrams

with a Boolean variable x_i and has two out-edges labeled 0 and 1. Each non-sink node represents the Boolean function $\bar{x}_i F_0 + x_i F_1$, where F_0 and F_1 are the cofactors w.r.t. x_i , and are represented by the BDDs rooted at the 0 and 1 edges respectively.

Moreover, a BDD satisfies two additional constraints:

- 1 There is a complete (but otherwise arbitrary) ordering of the input variables. Every path from source to sink in the BDD visits the input variables according to this ordering.
- 2 Each node represents a distinct logic function, that is, there is no duplicate representation of the same function.

A common optimization in implementing BDDs is the use of *complement edges* [BRB90]. A complement edge indicates that the connected function is to be interpreted as the complement of the ordinary function. When using complement edges, BDDs have only one sink node “1”, whereas the sink node “0” is represented as the complement of “1”. Boolean operations can be easily implemented as graph algorithms on the BDD data structure by simple recursive routines making Boolean function manipulation straightforward when using a BDD representation.

A critical aspect that contributes to the wide acceptance of BDDs for representing Boolean functions is that, in most applications, the amount of memory required for BDDs remains manageable. The number of nodes that are part of a BDD, also called the BDD size, is proportional to the amount of memory required, and thus the peak BDD size is a commonly used measure to estimate the amount of memory required by a specific computation involving Boolean expressions. However, the variable order chosen may affect the size of a BDD. It has been shown that for some type of functions the size of a BDD can vary from linear to exponential based on the variable order. Because of its impact, much research has been devoted to finding algorithms that can provide a good variable order. While finding the optimal order is an intractable problem, many heuristics have been suggested that find sufficiently good orders, from static approaches based on the underlying logic network structure in [MWBSV88, FFK88], to dynamic techniques that change the variable order whenever the size of the BDD grows beyond a threshold [Rud93, BLW95].

Moreover, much research work has been dedicated to the investigation of alternative representations to BDDs. For instance BMDs [BC01] target the representation of circuit with multiplicative cores, Zero-Suppressed BDDs [Min93] are suitable for the representation of sets, and MTBDDs [FMY97] can represent multi-valued functions. An example application which uses MTBDDs is presented in Chapter 5.

Binary decision diagrams are used extensively in symbolic simulation. The most critical drawback of this method is its high demand on memory resources, which are mostly used for BDD representation and manipulation. This book discusses recent techniques that transform the Boolean functions involved in symbolic simulations through parametrization and approximation. The objective of parametrization is to generate new functions that have a more compact BDD representation, while preserving the same results of the original symbolic exploration. The reduced size of the BDDs involved translates to a lower demand of memory resources, and thus it increases the size of IC designs that can be effectively tackled by this formal verification approach.

2.5 Models for design verification

The verification techniques that we present in this book rely on a structural gate-level network description of the digital system, generally obtained from the logic-synthesis phase of the design process. In the most general case, such networks are sequential, meaning that they contain storage elements like latches or banks of registers. Such circuits store state information about the system. Hence, the output at any point in time depends not only on the current input but also on historical values of the input. State transition models are a common abstraction to describe the functionality of a design. In this section

we review both their graph representation and the corresponding mathematical model.

2.5.1 Structural network model

A digital circuit can be modeled as a network of ideal combinational logic gates and a set of memory elements to store the circuit state. The combinational logic gates that we use are: *AND*, *OR*, *NOT* or *XOR*. Figure 2.4 reproduces the graphic symbol for each of these types.

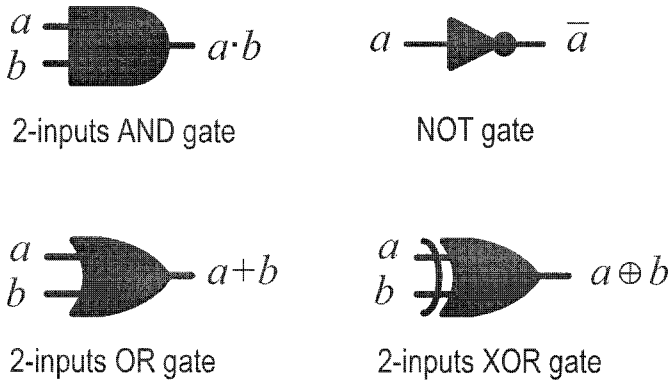


Figure 2.4: Graphic symbols for basic logic gates

A synchronous sequential network has a set of primary inputs and a set of primary outputs. We make the assumption that the combinational logic elements are ideal, that is, that there is no delay in the propagation of the value across the combinational portion of the network. Figure 2.5 represents such a model for a general network, also called netlist.

We also assume that there is a single clock signal to latch all the memory elements. In the most general case where a design has multiple clocks, the system can still be modeled by an equivalent network with a single global clock and appropriate logic transformations to the inputs of the memory elements.

Example 2.2. Figure 2.6 is an example of a structural network model for a 3-bits up/down counter with reset. The inputs to the system are the reset and the count signals. The outputs are three bits representing the current value of the counter. The clock input is assumed implicitly, and not represented in the figure. This system has four memory elements that store the current counter value and control if the counter is counting up or down. At each clock tick the system updates the values of the counter if the count signal is high. The value is incremented until it reaches the maximum value seven. Subsequently, it is decremented down to zero. Whenever the reset signal is held high, the counter is reset to zero.

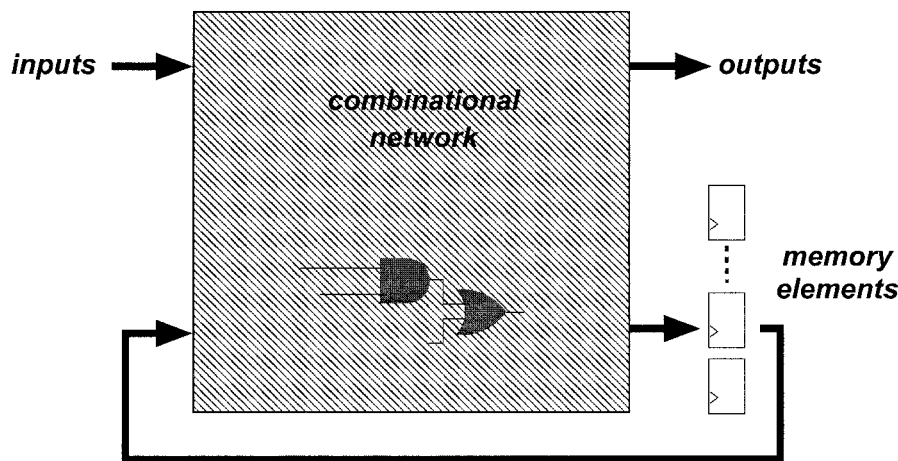


Figure 2.5: Structural network model schematic

The dotted perimeter in the figure indicates the combinational portion of the circuit's schematic.

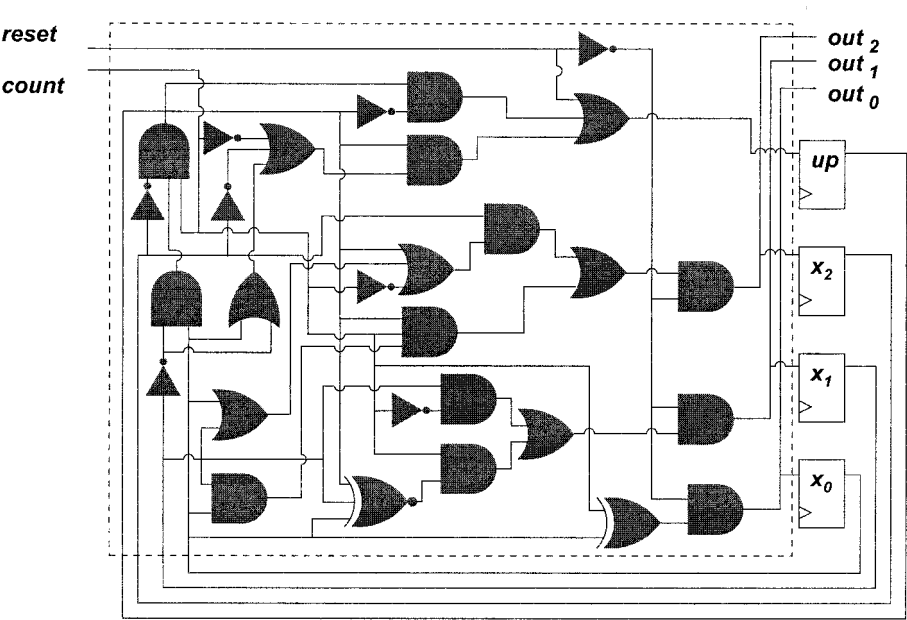


Figure 2.6: Network model of a 3-bits up/down counter with reset

2.5.2 State diagrams

A representation that can be used to describe the functional behavior of a sequential digital system is a *Finite State Machine* (FSM) model.

Such a model can be represented through state diagrams. A *state diagram* is a labeled, directed graph where each node represents a possible configuration of the circuit. The arcs connecting the nodes represent changes from one state to the next and are annotated by the input combinations which would cause the transition in a single clock cycle. State diagrams present only the functionality of the design, while the details of the implementation are not considered. Any implementation satisfying this state diagram will perform the function described. State diagrams also contain the required outputs at each state and/or at each transition. In a Mealy state diagram, the outputs are associated to each transition arc, while in a Moore state diagram outputs are specified with the nodes/states of the diagram. The initial state is marked in a distinct way to indicate the starting configuration of the system.

Example 2.3. Figure 2.7 represents the Moore state diagram corresponding to the counter of Example 2.2. Each state indicates the value stored in the three flip-flops x_0 , x_1 , x_2 in bold and in the up/down flip-flop under it. All the arcs are marked with the input signal required to perform that transition. Notice also that the initial state is indicated with a double circle.

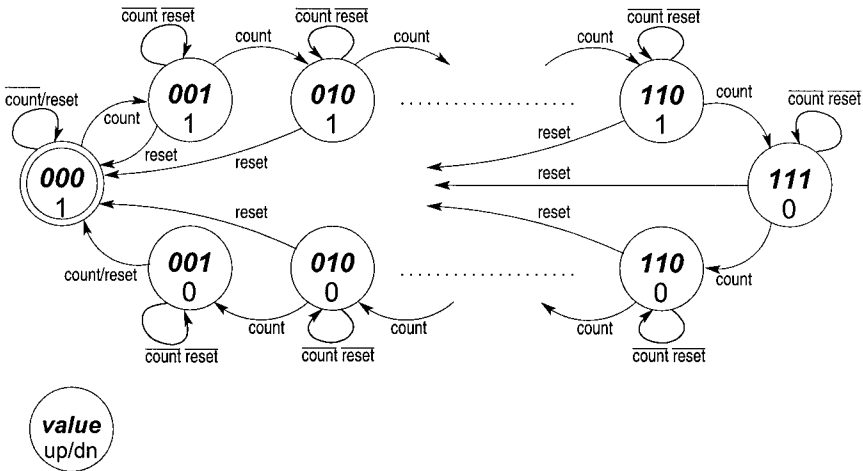


Figure 2.7: State diagram for a 3-bits up/down counter

In the most general case the number of configurations, or different states a system can be in, is much smaller than the number of all possible values that its memory elements can assume.

Example 2.4. Figure 2.8 represents the finite state machine for a 3-bits counter 1-hot encoded. Notice that even if the state is encoded using three bits, only the three configurations 001, 010, 100 are possible for the circuit. Such configurations are said to be reachable from the Initial State. The remaining five configuration 000, 011, 101, 110, 111 are said to be unreachable, since the circuit will never be in any of these states during normal operation.

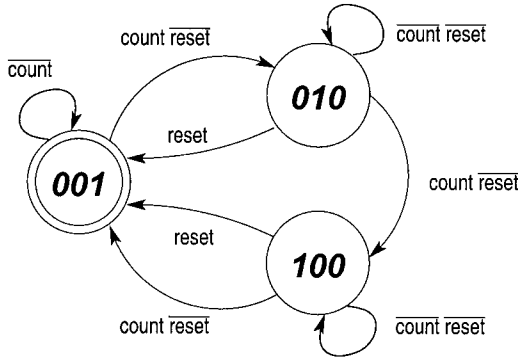


Figure 2.8: State diagram for a 1-hot encoded 3-bits counter

2.5.3 Mathematical model of finite state machines

An alternative way of describing a Finite State Machine (FSM) is through a mathematical description of the set of states and the rules to perform transitions between states. In mathematical terms, a completely specified, deterministic finite state machine is defined by a 6-tuple:

$$\mathcal{M} = (I, O, S, \delta, S_0, \lambda)$$

where:

- I is an ordered set (i_1, \dots, i_m) of Boolean input symbols,
- O is an ordered set (o_1, \dots, o_p) of Boolean output symbols,
- S is an ordered set (s_1, \dots, s_n) of Boolean state symbols,
- δ is the next-state function: $\delta : S \times I : \mathcal{B}^{n+m} \rightarrow S : \mathcal{B}^n$,
- λ is the output function $\lambda : S \times I : \mathcal{B}^{n+m} \rightarrow O : \mathcal{B}^p$,
- and S_0 is an *initial assignment* of the state symbols.

The definition above is for a Mealy-type FSM. For a Moore-type FSM the output function λ simplifies to: $\lambda : S : \mathcal{B}^n \rightarrow O : \mathcal{B}^p$.

Example 2.5. *The mathematical description of the FSM of Example 2.4 is the following:*

- $I = \{count, reset\},$
- $O = \{x_0, x_1, x_2\},$
- $S = \{001, 010, 100\},$

■ $\delta =$

<i>count reset</i>	<i>001</i>	<i>010</i>	<i>100</i>
<i>0 0</i>	<i>001</i>	<i>010</i>	<i>100</i>
<i>0 1</i>	<i>001</i>	<i>001</i>	<i>001</i>
<i>1 0</i>	<i>010</i>	<i>100</i>	<i>001</i>
<i>1 1</i>	<i>010</i>	<i>001</i>	<i>001</i>

- $\lambda = \{001 \rightarrow 001, 010 \rightarrow 010, 100 \rightarrow 100\},$
- $S_0 = \{001\}.$

While the state diagram representation is often much more intuitive, the mathematical model gives us a means of building a formal description of a FSM or, equivalently, of the behavior of a sequential system. The formal mathematical description is also much more compact, making it possible to describe even very complex systems for which a state diagram would be unmanageable.

2.6 Functional validation

This section is dedicated to the presentation of an algorithm for cycle-based logic simulation, a mainstream technique in functional validation. The next section outlines some of the techniques used in formal verification.

The most common approach to functional validation involves the use of a logic simulator software. A commonly deployed architecture is based on the leveled, compiled-code logic simulator approach by Barzilai and Hansen [BCRR87, Han88, WHPZ87].

Their algorithm starts from a gate-level description of a digital system and chooses an order for the gates based on their distance from the primary inputs – in fact, any order compatible with this partial ordering is valid. The name “leveled” of the algorithm is due precisely to this initial ordering of the gates in “levels”. The algorithm then builds an internal representation in assembly language where each gate corresponds to a single assembly instruction. The order of the gates and, equivalently, of the instructions, guarantees that the values for the instructions’ inputs are ready when the program counter reaches a specific instruction. This assembly block constitutes the internal representation of the circuit in the simulator.

Example 2.6. *Figure 2.9 reproduces the gate-level representation of the counter we used in Example 2.2. Each combinational gate has been assigned a level*

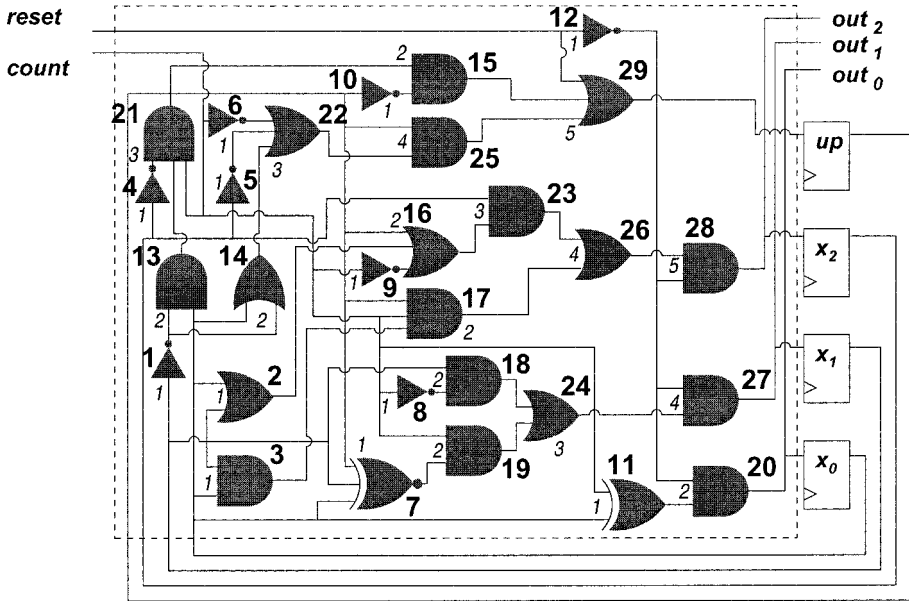


Figure 2.9: Compiled logic simulator

number (italicized in the graphic) based on its distance from the inputs of the design. Subsequently, gates have been numbered sequentially (gates numbers are in boldface), in a way compatible with this partial order. From this diagram it is possible to write the corresponding assembly block:

Note, that there is a one-to-one correspondence between each instruction in the assembly block and each gate in the logic network.

The assembly compiler can then take care of mapping the virtual registers of the source code to the physical registers' set available on the specific simulating host. Multiple input gates can be easily handled by composing their functionality through multiple operations. For instance, with reference to Example 2.6, the 3-input *XNOR* of gate 7, can be translated as:

```
7.      r7tmp = XOR(up, x1)
7bis.  r7      = XNOR(r7tmp, x0)
```

At this point, simulation is performed by providing an input test vector, executing the assembly block, and reading the output values computed. Such output values can be written to a separate file to be further inspected later to verify the correctness of the results. Figure 2.10 shows an outline of the algorithm, where the core loop includes the assembly code generated from the network.

```

Logic_Simulator(network_model)
{
    assign(present_state_signals, reset_state_pattern);
    while (input_pattern != empty)
    {
        assign(input_signals, input_pattern);
        CIRCUIT_ASSEMBLY;
        output_values = read(output_signals);
        state_values = read(next_state_signals);
        write_simulation_output(output_values);
        assign(present_state_signals, state_values);
        next input_pattern;
    }
}

```

Figure 2.10: Pseudo-code for a cycle-based logic simulator

Notice that, in first approximation, each of the assembly instructions can be executed in one CPU clock cycle of the host computer, thus providing a very high performance simulation. Moreover, this algorithm scales linearly with the length of the test vector and with the circuit complexity. The high performance and linear scalability of logic simulation are the properties that make this approach to functional validation widely accepted in industry.

The model just described is called a cycle-based simulator, since values are simulated on a cycle-by-cycle basis. Another family of simulators are event-driven simulators. The key difference is that each gate is simulated only when there is a change of the values at its inputs. This alternative scheduling approach makes possible to achieve a finer time granularity in the simulation, and also facilitates simulating events that occur between clock cycles.

Various commercial tools are available that use one or both of the approaches described above, and that have proven to have the robustness and scalability to handle the complexity of designs being developed today. Such commercial tools are also very flexible. Practical, cycle-based simulators allow for circuits with multiple clocks and the ability to mix cycle-based and event-based simulation to optimize performance [DeV97]. When deployed in a digital-system development context, simulators constitute the core engine of the functional validation process. However, the bulk of the time spent in verification is in the development of meaningful test sequences. Generally, the test stimuli are organized so that distinct sequences cover different aspects of the design functionalities. Each test sequence needs to be hand-crafted by verification engineers.

The simulated output values then are checked again by visual inspection. Both these activities require an increasing amount of engineering resources.

As mentioned before, some support in such development is available from specialized programming languages that make it possible for the verification engineer to use powerful primitives to create stimuli for the design, and to then generate procedures to automatically check the correctness of the output values [HKM01, KOW⁺01]. These test programs are then compiled and executed side-by-side with the simulation, exchanging data with it at every time step.

Another module that is often run in parallel to the simulator, or as a post-processing tool, is a coverage engine. Coverage engines collect analytical data on the portions of the circuit that have been exercised. Since designs are developed and changed on a daily basis, it is typical to make use of verification farms – thousands of computers running logic simulators – where the test suites are run every day for weeks at a time. Another common validation methodology approach in industry is pseudo-random simulation. Pseudo-random simulation is mostly used to provide chip-level validation and to complement stand-alone testing validation at the module level. This approach involves running logic simulation with stimulus generated randomly, but within specific constraints. For instance, a constraint could specify that the reset sequence is only initiated 1% of time. Or, it could specify some high-level flow of the randomly generated test, while leaving the specific vectors to be randomly determined [AGL⁺95, CIJ⁺95, YSP⁺99]. The major advantage of pseudo-random simulation is that the burden on the engineering team for test development is greatly reduced. However, since there is very limited control on the direction of the design-state exploration, it is hard to achieve a high coverage with this approach and to avoid just producing many redundant tests that have limited incremental usefulness.

Pseudo-random simulation is also often run using emulators which, conceptually, are hardware implementations of logic simulators. Usually they use configurable hardware architectures, based on FPGAs (Floating Point Gate Arrays) or specialized reconfigurable components that are configured to reproduce the gate-level description of the design to be validated [Pfi82, Hau95, CMA02]. While emulators can perform one to two orders of magnitude faster than software-based simulators, they constitute a very expensive solution. It is expensive because of the high raw cost of acquisition and the time-consuming process of configuring them for a specific design, which usually requires several weeks of engineering effort. Because of these reasons, emulators are mostly used for IC designs with a large market.

Even if design houses put forth great effort in developing tests for their designs and in maximizing the amount of simulation in order to achieve thorough coverage, simulation can only stimulate a small portion of the entire design and

can, therefore, potentially miss subtle design errors that might only surface under particular sets of rare conditions.

2.7 Formal verification

On the other side of the verification spectrum are formal verification techniques. These methods have the potential to provide a quantum leap in the coverage achievable on a design, thus improving significantly the quality of verification. Formal verification attempts to establish universal properties about the design, independent of any particular set of inputs. By doing so, the possibility of letting corner situations go untested in a design is removed. A formal verification system uses rigorous, formalized reasoning to prove statements that are valid for all feasible input sequences. Formal verification techniques promise to complement simulation because they can generalize and abstract the behavior of the design.

Almost all verification techniques can be roughly classified in one of two categories: model-based or proof-theoretic. *Model-based techniques* usually rely on a brute-force exploration of the whole solution space using symbolic techniques and finite state machine's representations. The main successful results of these methods are based on *symbolic state traversal* algorithms which allow the full exploration of digital systems, although with very limited scalability. Typical design sizes that can be handle by these solutions are up to a few hundreds latches, which is far from what is needed in an industrial context. At the root of state traversal approaches is some type of implicit or explicit representation of all the states of a systems that have been visited up to a certain step of the traversal. Since there is an exponential relationship between the number of states and the number of memory elements in a system, it is easy to see how the complexity of these algorithms grows exponentially with the number of memory elements in a system. This problem is called the *state explosion problemsymbolic state traversal*,state explosion problem and it is the main reason for the very limited applicability of the method. At the same time, the approach has the advantage of being fully automatic. A variant within this family is *bounded model checking*, which allows for the handling of much more complex systems. Although, as the name suggests, it has a limited (or bounded) depth of analysis.

An alternative approach, that belongs to the model-based category, is *symbolic simulation*. This method verifies a set of scalar tests with a single symbolic vector. Symbolic functions are assigned to the inputs and propagated through the circuit to the outputs. This method has the advantage that large input spaces can be covered *in parallel* with a single symbolic sweep of the circuit. Again, the bottleneck of this approach lies in the explosion of symbolic functions' representations. The next chapter is dedicated to discuss in depth symbolic simulation and a range of related solutions.

Symbolic approaches are also at the base of *equivalence checking*, another verification technique. In equivalence checking, the goal is to prove that two different network models provide the same functionality. In recent years, this problem has found heuristic solutions that are scalable to industrial-size circuits, thereby achieving full industrial acceptance. The success of scalable symbolic solutions in the domain of equivalence checking, gives hope that symbolic techniques will be also the basis for viable industrial-level solutions for formal verification.

A different family of approaches, *proof-theoretic methods*, are based on abstractions and hierarchical techniques aimed at proving the correctness of a system [Hue02, Joh01]. Verification within this framework uses theorem-proving software to provide support in reasoning and deriving proofs about the specifications and the implementation model of a design. They use a variety of logic representations, called theories. The design complexity that a theorem prover can handle is unlimited. However, currently available theorem provers require significant human guidance: even with a state-of-the-art theorem prover, proving that a model satisfies a specification is a very hand-driven process. Thus, this approach is still impractical for most industrial applications.

We conclude the section by describing in more detail one of the techniques outlined above, to give the reader a sense of the computational procedures involved in formal verification. Symbolic simulation techniques will be described in the next chapter.

2.7.1 Symbolic finite state machine traversal

One approach used in formal verification is to focus on a property of a circuit and to prove that this property holds, for any configuration of the circuit, that is reachable from its initial state. For instance, such property could specify that if the system is properly initialized, it never deadlocks. Or, in the case of pipelined microprocessors, one property could be that any issued instruction completes within a finite number of clock cycles. The proof of properties such as these, requires, first of all, to construct a global state-graph representing the combined behavior of all the components of the system. After this, each state of the graph needs to be inspected to check if the property holds for that state. Many problems in formal hardware verification are based on reachable-state computation of finite state machines. A *reachable state* is just a state that is reachable for some input sequence from a given set of possible initial states (see Example 2.4). This type of computation uses a symbolic breadth-first approach to visit all reachable states, also called *reachability analysis*. The approach, described below, has been published in seminal papers by Madre, Coudert and Berthet [CBM89] and later in [TSL⁺90, BCL⁺94].

In the context of FSMs, reachable-state computations are based on implicit traversal of the state diagram (Section 2.5.2). The key step of the traversal is in computing the *image* of a given set of states in the diagram, that is, computing the set of states that can be reached from the present state with one single transition (following one edge in the diagram).

Example 2.7. Consider the state diagram of Figure 2.7. The image of the one state set $\{000 - 1\}$ is $\{000 - 1, 001 - 1\}$ since there is one edge connecting the state $000 - 1$ to both of these states. The image of the set $\{110 - 1, 111 - 0\}$ is $\{000 - 1, 110 - 1, 111 - 0, 110 - 0\}$.

The following definition formalizes the operation of image computation:

Definition 2.10. Given a FSM \mathcal{M} and a set of states R , its image is the set of states that can be reached by one step of the state machine. With reference to the model definition of Section 2.5.3, the image is:

$$Img(\mathcal{M}, R) = \{s' | s' = \delta(s, i), s \in R, i \in I\}$$

It is also possible to convert the next-state function $\delta()$ into a *transition relation* $TR(s, s')$, which is asserted when there is some input i such that $\delta(s, i) = s'$. This relation is defined by existentially quantifying the inputs from $\delta()$:

$$TR(s, s') = \exists i \left[\bigwedge_{k=1}^n \delta_k(s, i) \equiv s'_k \right] \quad (2.4)$$

where δ_k represents the transition function for the k -th bit. As it could be imagined, the transition relation can be represented by a corresponding characteristic function – see Definition 2.4 – χ_{TR} which equals 1 when $TR(s, s')$ holds true.

Finally, the image of a pair $\langle \mathcal{M}, R \rangle$ can be defined using characteristic functions. Given a set of states R with characteristic function χ_R , its *image under transition relation* TR is the set Img having the characteristic function:

$$\chi_{Img}(s') = \exists s (\chi_{TR}(s, s') \cdot \chi_R(s)) \quad (2.5)$$

Symbolic FSM traversal performs image computations iteratively starting from the initial state. At each steps it accumulates the states visited (that is, the images) into a *reached* set. The traversal ends when a fixed point is found, which is detected when the reached set does not grow from iteration to iteration. At the end of the computation, the reached set represents the characteristic function of all the states that can be reached by the system, and it can be used to prove properties specified over the states of the design. The fixed point computation of symbolic FSM traversal will be discussed in more detail in Section 3.4.1.

2.8 Summary

This chapter presented an overview of the design and verification flow involved in the development of a digital integrated circuit. It discussed the main techniques used to verify such circuits, namely functional validation (by means of logic simulation) and formal verification, using a range of techniques.

We reviewed basic concepts and representations for Boolean functions and for sequential systems, and described how a logic simulator works. The models discussed in the earlier sections will be needed to present all the main techniques in the later chapters. The last part of the chapter was dedicated to skim over a range of formal verification techniques, and give a sense of this methodology through the presentation of symbolic FSM traversal. The next chapter covers in great detail another technique, symbolic simulation, and draws the similarities between that and reachability analysis.

References

- [AGL⁺95] Aharon Aharon, Dave Goodman, Moshe Levinger, Yossi Lichtenstein, Yossi Malka, Charlotte Metzger, Moshe Molcho, and Gil Shurek. Test program generation for functional verification of PowerPc processors in IBM. In *DAC, Proceedings of Design Automation Conference*, pages 279–285, June 1995.
- [BC01] Randal E. Bryant and Yirng-An Chen. Verification of arithmetic circuits using binary moment diagrams. *International Journal on Software Tools for Technology Transfer*, 3(2):137–155, 2001.
- [BCL⁺94] Jerry R. Burch, Edward M. Clarke, David E. Long, Ken L. McMillan, and David L. Dill. Symbolic model checking for sequential circuit verification. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, 13(4):401–424, 1994.
- [BCRR87] Zeev Barzilai, J. Lawrence Carter, Barry K. Rosen, and Joseph D. Rutledge. HSS - a high-speed simulator. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, pages 601–617, July 1987.
- [Ber03] Janick Bergeron. *Writing Testbenches: Functional Verification of HDL Models*. Kluwer Academic Publishers, 2nd edition, 2003.
- [BL92] Jerry R. Burch and David E. Long. Efficient Boolean function matching. In *ICCAD, Proceedings of the International Conference on Computer Aided Design*, pages 408–411, November 1992.
- [BLW95] Beate Bollig, Martin Löbbing, and Ingo Wegener. Simulated annealing to improve variable orderings for OBDDs. In *International Workshop on Logic Synthesis*, pages 5.1–5.10, May 1995.
- [BRB90] Karl Brace, Richard Rudell, and Randal E. Bryant. Efficient implementation of a BDD package. In *DAC, Proceedings of Design Automation Conference*, pages 40–45, 1990.

- [Bry86] Randal E. Bryant. Graph-based algorithms for Boolean function manipulation. *IEEE Transactions on Computers*, 35(8):677–691, August 1986.
- [Bry92] Randal E. Bryant. Symbolic Boolean manipulation with ordered binary–decision diagrams. *ACM Computing Surveys*, 24(3):293–318, September 1992.
- [CBM89] Olivier Coudert, Christian Berthet, and Jean Christophe Madre. Verification of synchronous sequential machines based on symbolic execution. In *Automatic Verification Methods for Finite State Systems, International Workshop*, volume 407 of *Lecture Notes in Computer Science*, pages 365–3. Springer, June 1989.
- [CIJ⁺95] A. Chandra, V. Iyengar, D. Jameson, R. Jawalekar, I. Nair, B. Rosen, M. Mullen, J. Yoon, R. Armoni, D. Geist, and Y. Wolfsthal. AVPGEN - a test generator for architecture verification. *IEEE Transactions on Very Large Scale Integration (VLSI) Systems*, 3(2):188–200, June 1995.
- [CMA02] Srihari Cadambi, Chandra S. Mulpuri, and Pranav N. Ashar. A fast, inexpensive and scalable hardware acceleration technique for functional simulation. In *DAC, Proceedings of Design Automation Conference*, pages 570–575, June 2002.
- [DeV97] Charles J. DeVane. Efficient circuit partitioning to extend cycle simulation beyond synchronous circuits. In *ICCAD, Proceedings of the International Conference on Computer Aided Design*, pages 154–161, November 1997.
- [FFK88] Masahiro Fujita, Hisanori Fujisawa, and Nobuaki Kawato. Evaluation and improvements of Boolean comparison method based on binary decision diagrams. In *ICCAD, Proceedings of the International Conference on Computer Aided Design*, pages 2–5, November 1988.
- [FMY97] Masahiro Fujita, Patrick McGeer, and Jerry Yang. Multi-terminal binary decision diagrams: An efficient datastructure for matrix representation. *Formal Methods in System Design*, 10(2-3):149–169, 1997.
- [Han88] Craig Hansen. Hardware logic simulation by compilation. In *DAC, Proceedings of Design Automation Conference*, pages 712–716, June 1988.
- [Hau95] Scott Hauck. *Multi-FPGA Systems*. PhD thesis, University of Washington, Dept. of Computer Science and Engineering, 1995.
- [HKM01] Faisal I. Haque, Khizar A. Khan, and Jonathan Michelson. *The Art of Verification with Vera*. Verification Central, 2001.
- [HMN01] Yoav Hollander, Matthew Morley, and Amos Noy. The *e* language: A fresh separation of concerns. In *Technology of Object-Oriented Languages and Systems*, volume TOOLS-38, pages 41–50, March 2001.
- [Hue02] Gérard Huet. Higher order unification 30 years later. In *Theorem Proving in Higher Order Logics*, volume 2410 of *Lecture Notes in Computer Science*, pages 3–12. Springer-Verlag, August 2002.

- [Joh01] Steven Johnson. View from the fringe of the fringe. In *CHARME*, volume 2144 of *Lecture Notes in Computer Science*, pages 1–12. Springer-Verlag, September 2001.
- [KN96] Michael Kantrowitz and Lisa M. Noack. I’m done simulating; now what? verification coverage analysis and correctness checking of the DECchip 21164 Alpha microprocessor. In *DAC, Proceedings of Design Automation Conference*, pages 325–330, June 1996.
- [KOW⁺01] Tommy Kuhn, Tobias Oppold, Markus Winterholer, Wolfgang Rosenstiel, Marc Edwards, and Yaron Kashai. A framework for object oriented hardware specification, verification and synthesis. In *DAC, Proceedings of Design Automation Conference*, pages 413–418, June 2001.
- [LMUZ02] Oded Lachish, Eitan Marcus, Shmuel Ur, and Avi Ziv. Hole analysis for functional coverage data. In *DAC, Proceedings of Design Automation Conference*, pages 807–812, June 2002.
- [MD93] Frédéric Mailhot and Giovanni DeMicheli. Algorithms for technology mapping based on binary decision diagrams and on Boolean operations. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, 12:599–620, May 1993.
- [Min93] S.-I. Minato. Zero-suppressed bdds for set manipulation in combinatorial problems. In *DAC, Proceedings of Design Automation Conference*, pages 272–277, June 1993.
- [MWBSV88] Sharad Malik, Albert Wang, Robert K. Brayton, and Alberto Sangiovanni-Vincentelli. Logic verification using binary decision diagrams in a logic synthesis environment. In *ICCAD, Proceedings of the International Conference on Computer Aided Design*, pages 6–9, November 1988.
- [Pfi82] Gregory F. Pfister. The yorktown simulation engine: Introduction. In *DAC, Proceedings of Design Automation Conference*, pages 51–54, January 1982.
- [Rud93] Richard Rudell. Dynamic variable ordering for ordered binary decision diagrams. In *ICCAD, Proceedings of the International Conference on Computer Aided Design*, pages 42–47, November 1993.
- [TSL⁺90] Hervé Touati, Hamid Savoj, Bill Lin, Robert K. Brayton, and Alberto L. Sangiovanni-Vincentelli. Implicit state enumeration of finite state machines using BDDs. In *ICCAD, Proceedings of the International Conference on Computer Aided Design*, pages 130–133, November 1990.
- [WHPZ87] Laung-Terng Wang, Nathan E. Hoover, Edwin H. Porter, and John J. Zasio. SSIM: A software leveled compiled-code simulator. In *DAC, Proceedings of Design Automation Conference*, pages 2–8, June 1987.
- [YSP⁺99] Jun Yuan, Kurt Schultz, Carl Pixley, Hiller Miller, and Adnan Aziz. Modeling design constraints and biasing using bdds in simulation. In *ICCAD, Proceedings of the International Conference on Computer Aided Design*, pages 584–590, November 1999.



<http://www.springer.com/978-0-387-24411-2>

Scalable Hardware Verification with Symbolic Simulation

Bertacco, V.

2006, XX, 180 p. 40 illus., Hardcover

ISBN: 978-0-387-24411-2