Quantum Circuit Simulation

George F. Viamontes • Igor L. Markov • John P. Hayes

Quantum Circuit Simulation

Springer

George Viamontes
Department of Electrical Engineering &
Computer Science
Advanced Computer Architecture Laboratory
gviamont@eecs.umich.edu
http://www.eecs.umich.edu/~gviamont/

Igor L. Markov
Department of Electrical Engineering &
Computer Science
University of Michigan
Ann Arbor, MI 48109-2121
USA
imarkov@eecs.umich.edu

John P. Hayes
Department of Electrical Engineering &
Computer Science
University of Michigan
Ann Arbor, MI 48109-2121
USA
jhayes@eecs.umich.edu

ISBN 978-90-481-3064-1 e-ISBN 978-90-481-3065-8
Springer Dordrecht Heidelberg London New York

Library of Congress Control Number: 2009929020

Preface

Recent scientific advances, both experimental and algorithmic, have brought quantum information processing closer to reality. The ability to process information "quantumly" already allows ultra-precise metrology and more secure communication, and quantum algorithms allow computations such as factoring to be done significantly faster than we know how to do them on classical computers. With these advances, quantum simulation has become an increasingly important topic for both theoretical and engineering reasons. On the theoretical front, progress toward defining the class of circuits that can be simulated efficiently on a classical computer has and will continue to lead to a deeper understanding of the power of quantum computation. Even though efficient simulation of all quantum circuits may not be possible, the circuits that will most likely make up the majority of operations on a quantum computer can in fact be simulated efficiently. These are fault-tolerant error-correction circuits; they are composed of only Clifford-group gates, which, as this book demonstrates, can be simulated surprisingly efficiently on a classical computer.

Recent results suggest that larger classes of circuits can also be simulated efficiently on classical computers. From a circuit perspective, efficient simulation can result from operating with a restricted set of gates, such as the Clifford-group gates, or from operating with an arbitrary gate set if the circuit has a small treewidth, as Markov and Shi have shown. From a physical perspective, efficient simulation can also result from limiting the amount of entanglement in the intermediate states of the computation. For example, Vidal has shown that one-dimensional many-body systems can be simulated efficiently and more recently, Bravyi and Terhal have identified a large class of quantum Hamiltonians for which the adiabatic evolution can be simulated efficiently on a classical computer. Ideally, we would like to characterize the class of quantum circuits that are computationally no more efficient than the equivalent classical versions. Such developments would lead to new theoretical results, sharpening our understanding of the differences between classical complexity classes and their quantum counterparts. It would also help us understand the true power of quantum computation.

On the engineering front, efficient simulation provides the ability to validate and perform sanity checks on circuit components prior to their implementation. It is also useful for evaluating circuits, in particular for benchmarking the error rates of gates and memory. Through simulation, engineers can save on both the cost and effort involved in building hardware. Quantum simulation can also offer software support for quantum control and algorithm design.

The main requirement of a quantum simulator is that its output match the information produced by an ideal quantum computer. It may also produce additional information, such as descriptions of states before measurement. A quantum circuit can be simulated efficiently only when the input and output can be expressed in a compact form, since in general a quantum circuit may require an exponential-size matrix to express the evolution and an exponentially long vector to express the state. Thus, a quantum simulator should be able to operate on limited amounts of entanglement, to represent (and possibly compress) quantum evolution and states in a classical data structure, and to perform classical algorithms. The QuIDDPro simulator described in this book offers precisely this functionality.

This book is a major step forward in the development of algorithms and software tools for designing and simulating quantum circuits. Using a consistent notation, it describes in detail simulation techniques that up to this point had been scattered throughout the research literature in physics, computer science, and computer engineering journals.

The book describes an innovative software system, called QuIDDPro, that can be used to describe and simulate quantum circuits. The QuIDDPro software is freely available and can be used as a "quantum calculator" by researchers interested in developing new quantum algorithms and by students interested in learning quantum information processing. QuIDDPro has already been used by students in quantum computing courses to learn the behavior of quantum circuits. The book also describes algorithms and techniques that will be indispensable to computer scientists and engineers developing languages, compilers, optimizers, and CAD tools for quantum computation.

The algorithmic toolbox implemented in QuIDDPro is based on a powerful new data structure called a quantum information decision diagram (QuIDD). QuIDDs are the secret behind QuIDDPro's ability to simulate important classes of quantum circuits with remarkable efficiency. QuIDDPro simulations of important classes of quantum circuits can be several orders of magnitude faster than similar computations done with common numerical software tools such as Matlab. The book describes in detail the underlying mathematical models, algorithms, and data structures that are used to implement QuIDDPro. Two appendices introduce the reader to the QuIDDPro software and show how to implement some common quantum algorithms in QuIDDPro, including Grover's search algorithm and Shor's integer factoring algorithm.

The book is self contained, covering the fundamentals of linear algebra and quantum mechanics needed to understand quantum circuits and to use the

simulator. It requires only basic familiarity with algebra, graph algorithms, and computer engineering.

The experience of the authors that combines quantum computing research with electronic design automation has yielded a book that is a must read for anyone interested in the simulation of quantum algorithms and in the implementation of design tools for quantum computers.

Alfred V. Aho
Columbia University, New York, NY

Krysta M. Svore
Microsoft Research, Redmond, WA

Contents

1 Introduction 1
- 1.1 Quantum Circuits 1
- 1.2 Quantum Simulation 3
- 1.3 Book Outline 4

2 Gate Modeling and Circuit Simulation 7
- 2.1 Classical Digital Circuits 7
- 2.2 Simulation with Binary Decision Diagrams 11
- 2.3 Sequential Circuits and Synchronization 17
- 2.4 Summary 18

3 Linear Algebra and Quantum Mechanics 19
- 3.1 Linear Algebra 19
- 3.2 Quantum Mechanics 24
- 3.3 Summary 32

4 Quantum Information Processing 33
- 4.1 Quantum Gates 33
- 4.2 Quantum Circuits 38
- 4.3 Synchronization of Quantum Circuits 42
- 4.4 Sample Algorithms 43
- 4.5 Summary 46

5 Special Case: Simulating Stabilizer Circuits 47
- 5.1 Basics of a Quantum Circuit Simulator 47
- 5.2 Stabilizer States, Gates and Circuits 49
- 5.3 Data Structures 51
- 5.4 Algorithms 52
- 5.5 Summary 55

6 Generic Circuit Simulation Techniques 59
6.1 Qubit-wise Multiplication 59
6.2 P-blocked Simulation 61
6.3 Tensor Networks 63
6.4 Slightly-entangled Simulation 66
6.5 Summary 70

7 State-Vector Simulation with Decision Diagrams 71
7.1 Quantum Information Decision Diagrams 71
7.2 Scalability of QuIDD-based Simulation 80
7.3 Empirical Validation 88
7.4 Related Decision Diagrams 92
7.5 Summary 100

8 Density-Matrix Simulation with QuIDDs 103
8.1 QuIDD Properties and Density Matrices 103
8.2 QuIDD-based Outer Product 105
8.3 QuIDD-based Partial Trace 106
8.4 Empirical Validation 109
8.5 Summary 114

9 Checking Equivalence of States and Circuits 115
9.1 Quantum Equivalence Checking 115
9.2 Global-Phase Equivalence 117
9.3 Relative-Phase Equivalence 122
9.4 Empirical Validation 126
9.5 Summary 128

10 Improving QuIDD-based Simulation 133
10.1 Gate Algorithms 133
10.2 Dynamic Tensor Products and Partial Tracing 139
10.3 Empirical Validation 145
10.4 Summary 150

11 Closing Remarks 153

A QuIDDPro Simulator 155
A.1 Running the Simulator 155
A.2 Functions and Code in Multiple Files 158
A.3 Language Reference 160

B QuIDDPro Examples 177
B.1 Well-known Quantum States 177
B.2 Grover's Search Algorithm 178
B.3 Shor's Integer Factoring Algorithm 179

References 181

Index 187

1

Introduction

The construction of computer algorithms and software models that simulate physical systems plays a fundamental role in all branches of science and engineering. The physicist and Nobel laureate Richard Feynman, among others, observed in the 1980s that the important task of simulating quantum-mechanical processes on a standard computer requires an extraordinary amount of computer memory and runtime [41]. Such observations gave rise to the notion of *quantum computing*, where quantum mechanics itself is used to simulate quantum behavior. The key insight is to replace the familiar 0 and 1 bits of conventional or *classical* computing with information units called qubits (quantum bits) that capture quantum states of elementary particles or atomic nuclei. By operating on qubits, a quantum computer can, in principle, process exponentially more data than a classical computer in a similar number of steps. In the 1990s, several fast quantum methods were discovered for such applications as searching large databases [38] and factoring large numbers [82]; the latter is a basic step in some forms of codebreaking.

1.1 Quantum Circuits

Implementing quantum algorithms in physical hardware, that is, building quantum computing circuits, has proven to be extremely difficult. A method called liquid-state nuclear magnetic resonance (NMR) was used around 2001 to demonstrate Shor's factoring algorithm running on a 7-qubit quantum computer [103]. Atoms in an organic molecule were used to represent individual qubits, and radio-frequency (RF) pulses were used to address the qubits. However, this technique is not scalable to greater numbers of qubits, in large part because of the practical difficulties of addressing individual qubits in big molecules. Significant progress was also reported in the 2000s using several unrelated implementation techniques, but most still suffer from scalability limitations.

A particularly promising class of quantum circuit technologies rely on semiconductor devices, and so can draw on the massive investment already made by the semiconductor industry in reliable and scalable chip manufacturing techniques for very large-scale integrated (VLSI) circuits. A quantum dot [104] is a particle—a single electron, for example—that is trapped in a semiconductor in a way that enables it to represent a qubit. Quantum dots can be fabricated in gallium arsenide (GaAs) chips and addressed individually by means of conventional wires. Ion traps [69], originally developed at the U.S. National Institute of Standards and Technology (NIST) for atomic clocks, use carefully engineered electric fields to suspend electrically charged atoms (ions) of cesium or beryllium in long chains, with vibrational coupling between them. These ions also represent individual qubits and can be addressed by lasers. While the first laboratory demonstrations of ion traps used very bulky equipment, more recent ion traps reside entirely within tiny GaAs chips. A third semiconductor technology relies on superconducting qubits that are implemented with Josephson junctions. Despite its need for an extremely low-temperature (cryogenic) environment, this technology promises to significantly reduce power consumption in large-scale quantum circuits, and has also been proposed for conventional super-computers. Prototypes of superconducting quantum circuits were built in a number of laboratories using variants of the technology.

Quantum information processing by means of quantum circuits and algorithms has already proven practical in secure optical communications, where photons serve as the qubits. Quantum communication has several distinct advantages over conventional methods. For instance, in classical communication, eavesdroppers can escape detection. When qubits rather than bits are being communicated, on the other hand, eavesdropping is readily detected due to the fact that it requires a quantum-mechanical measurement that alters the state of the measured information. Moreover, quantum states cannot be copied, therefore quantum messages cannot easily be saved for future decryption. In 2007, demonstrations of single-photon quantum communication in optical fiber and free space achieved ranges in excess of 100 kilometers.

Currently, fiber-based quantum communication technologies are being commercialized by several companies in the US and Europe. Their potential customers include financial institutions, military organizations, and governments. For example, the results of the October 2007 general election in Geneva, Switzerland were transmitted using quantum communications. Another application that is being actively explored is quantum communication between satellites and ground stations, where the transmission medium is inherently insecure, and physical contact between the communicating parties is impractical. As with traditional communication technologies, increasingly sophisticated quantum communication protocols and networks must be supported by quantum information-processing circuits, whose simulation is the central topic of this book.

1.2 Quantum Simulation

Software simulation has long been an invaluable tool for the design and testing of classical systems, such as electrical circuits, digital logic circuits, communication systems, etc. [39]. In the digital circuit domain, which is most relevant to this work, simulation is typically considered a computer-aided design (CAD) task, and was itself once thought to be computationally intractable. Naive simulation and synthesis techniques for n-bit digital circuits require $\Omega(2^n)$ runtime and memory, that is, the computational complexity of these techniques tends to grow exponentially with the circuit size n. Later advances in algorithm design brought about the ability to perform circuit simulation far more efficiently in many practical cases. One such advance was the development of a data structure called the *reduced ordered binary decision diagram* (ROBDD) [20], which greatly compresses large collections of digital signals, and allows direct manipulation of the compressed form. Similarly, software simulation can be expected to play a vital role in the development of quantum computing hardware by enabling the modeling and analysis of large-scale designs before they are implemented physically.

The mathematics needed for simulating quantum processes, including quantum computational algorithms, is the linear algebra of complex-valued vector spaces. Such processes can often be modeled by quantum circuits, which are analogous to classical digital circuits. Unfortunately, straightforward simulation of quantum circuits by classical computers executing standard linear-algebraic routines requires $\Omega(2^n)$ time and memory [41, 61].[1] However, just as ROBDDs and other innovations have made the simulation of very large classical circuits tractable, clever algorithmic techniques can allow the efficient simulation of quantum circuits in many important cases. Interestingly, if a classical computer can simulate a quantum circuit or algorithm solving a particular problem, then this implies that a classical computer is computationally as powerful as a quantum computer for the problem in question. Therefore, by discovering new classical algorithms which can efficiently simulate quantum algorithms in certain cases, we are probing the limitations of quantum computing.

One unavoidable limitation of quantum computing is that it is extremely error-prone due to the fact that, unlike classical bits, qubits interact with their environment in ways that cause their values to decay as time passes. Another is that reading the value of a qubit—quantum measurement—is a non-deterministic process, which gives quantum information processing a

[1] Quantum circuits can be simulated using polynomial-sized memory resources, but the dramatic increase in runtime required by such techniques to periodically recompute intermediate results makes them impractical. The best known complexity-theoretic results are discussed in [32]. In particular, quantum circuit simulation belongs to the complexity class P^{PP} which includes decision problems solvable in polynomial time with the help of an oracle for solving problems from PP, i.e., those solvable in probabilistic polynomial time.

probabilistic character quite different from most classical computation. In this work, we describe the development of practical software methods for simulating general quantum circuits. Such simulation can be used as a tool to address the following problems:

1. Characterizing the effect of errors in physical quantum circuits
2. Evaluating error-correction techniques to cope with such errors
3. Verifying the correctness of synthesized quantum circuits
4. Exploring the boundaries between quantum and classical computation

A number of algorithms may form the basis for a quantum simulator Such algorithms typically rely on some type of structure present in quantum circuits or intermediate quantum states. We illustrate these issues by describing the theoretical framework of several existing quantum algorithms. A particular goal of the book is to demonstrate how to take any of these theoretical methods and develop a practical software tool to address the analysis and synthesis issues for quantum circuits. Several chapters focus on a particular data structure, the quantum information decision diagram (QuIDD), that we developed in our research [95, 97, 99]. QuIDD-based simulation differs from most other techniques in that it automatically detects many types of useful structures in the target circuits. For example, it is surprisingly effective at simulating instances of Grover's algorithm, both theoretically and empirically. This book presents and proves some key results about QuIDD-based simulation. It also provides an in-depth look at the implementation of a QuIDD-based simulator, QuIDDPro, to demonstrate the design and evaluation of a complete simulation package.

Several other quantum simulation techniques have been developed that excel under different circumstances. For example, an important class of quantum circuits known as stabilizer circuits can be simulated very efficiently with the specialized algorithms of Gottesman, Knill and Aaronson [36, 1]. A broad range of "slightly-entangled" circuits can be simulated by a technique due to Vidal [106, 107], while circuits implementing the frequently-used Quantum Fourier Transform can be approximately simulated by the tensor-contraction technique of Markov and Shi [55], as pointed out by Aharonov et al. [4] as well as by Yoran and Short [112]. The book discusses all of these techniques, although in less detail than QuIDD-based simulation.

1.3 Book Outline

Chapter 2 reviews circuit models used in digital logic, as well as current methods of simulating and verifying traditional logic circuits. A key data structure for representing and manipulating logic functions, the binary decision diagram (BDD) and its applications, is discussed. For readers who are less familiar with the relevant mathematical and quantum mechanical background, Chapter 3 reviews the basic concepts relevant to the topics covered in this book. Chapter

4 introduces the quantum circuit model with particular emphasis on connections to its linear-algebraic underpinnings.

Chapter 5 outlines the major features that every quantum circuit simulator must implement using a well-known simulation technique called the stabilizer formalism as an example. Chapter 6 expands this discussion by describing a variety of other recently developed computational techniques. The quantum information decision diagram (QuIDD) and the simulator QuIDDPro are presented in detail in Chapter 7 and expanded upon in Chapter 8. QuIDDPro implements the QuIDD data structure and all related algorithms with an expressive front-end language. This chapter also discusses a practical class of quantum states and operators that can be simulated efficiently using QuIDDs, with quantum search serving as the main benchmark algorithm for evaluation purposes. Chapter 9 addresses the verification of synthesized quantum circuits by simulating the circuits and checking for equivalence among the resultant states and operators. We show in Chapter 10 that QuIDDPro's input language enables some automatic speed-up techniques for QuIDD-based simulation. Finally, Chapter 11 summarizes the trends of quantum information science addressed in the book, and gives some perspectives on possible future applications. Appendix A introduces the reader to the QuIDDPro software that implements many of the algorithms covered in this book. Appendix B presents QuIDDPro source code for several well-known quantum circuits.

Acknowledgments

Our work on this book was partially supported by the DARPA QuIST program, US AirForce Research Laboratory (agreement No. FA8750-05-1-0282) and National Science Foundation (award No. 0208959). We are also thankful to colleagues and students who contributed valuable comments at various stages of our work, including Al Aho, Héctor Garcia, Smita Krishnaswamy, Manoj Rajagopalan and Yaoyun Shi. A number of users of our quantum circuit simulator QuIDDPro reported bugs and made valuable suggestions, which helped us improve the software. Full documentation and latest versions are going to be available at `http://vlsicad.eecs.umich.edu/Quantum/qp/`

2

Gate Modeling and Circuit Simulation

As with other sophisticated technologies in information processing, the practical implementation and use of electronic circuits is preceded by their mathematical modeling and simulation. Detailed calculations of physical parameters are employed to characterize individual circuit elements with respect to environmental conditions and manufacturing variations, but are difficult to scale to very large integrated circuits. In this context, a compact model captures the essence of a circuit's operation without unnecessary details. Such models express the expected outputs of each circuit component given its inputs, and are key to fast algorithms for simulating the functional behavior of the entire circuit: its speed, power dissipation, temperature distribution, etc. Simulation is thus an important prerequisite to validating the operation of the circuit.

This chapter discusses the modeling and simulation of classical digital circuits to introduce and motivate the methods used for quantum circuit simulation. Emphasis is placed on functional simulation and its use in formal verification of both combinational and sequential circuits.

2.1 Classical Digital Circuits

The literature on modeling and simulating traditional electronic circuits is extensive, and is stratified by the level of detail sought. For example, the electrical behavior of wires, transistors and small circuits is typically modeled in terms of ordinary differential equations using the SPICE [85, 72] simulation tool and its variants. Reduced models are used in fast timing analysis [72, 71], while logic-level simulation that propagates 0s and 1s from a circuit's inputs to its outputs, requires even fewer computational resources because it can represent gates by simple look-up tables. Figure 2.1 illustrates the modeling of a standard inverter (NOT gate) at several different levels of abstraction.

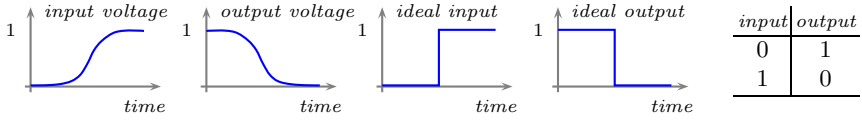


Fig. 2.1. Modeling inverter behavior at the electrical and logical levels: Analog voltage signals at (a) the input and (b) the output; digital voltage signals at (c) the input and (d) the output; (e) compact logic-level model (truth table).

Simulating Single Input Combinations

We illustrate logic-level simulation using the small circuit example in Figure 2.2a, whose function is expressed by the Boolean equation:

$$F(x, y, z) = \text{NAND}(\text{NAND}(x, y), \text{NOT}(z))$$

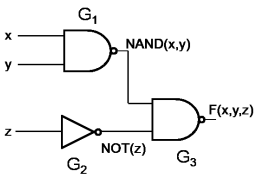
This is an example of a *combinational* logic circuit, meaning no memory elements are present and time is effectively ignored. Later, in Section 2.3, we will discuss the simulation of *sequential* circuits, which include memory and take time into account.

Standard simulators assume fixed input values, and traverse the circuit from primary inputs to primary outputs in topological order, implying that a gate is evaluated only after all of its predecessors have been evaluated. The output of every gate is looked up in a truth table, such as that of Figure 2.2b, based on the values available at the gate's inputs. These values may be produced by similar look-ups for predecessor gates. The result of such functional simulation on all possible input combinations is presented in the form of a truth table in Figure 2.2c, although simulation of larger circuits is often restricted to a subset of the entries in such tables.

The gains in simulation speed that result from the use of truth tables facilitate simulation algorithms whose runtime scales linearly with circuit size [39], allowing them to deal with circuits of any practical size. The same algorithmic template (topological traversal with table look-up) can be extended to timing simulation that estimates the duration of each signal transition [72]. A key idea is to approximately capture the waveforms from Figures 2.1a-b by a small set of parameters, for example, the logic-level transition, the timing of the 50%-point, or the slope (angle) of the transition. The look-up tables used in static analysis tools capture gate delay for all possible functional transitions and several values of continuous parameters, so that intermediate values can be handled by linear interpolation. It is interesting to note that the slope parameter is difficult to observe directly, but often affects the timing of downstream transitions.

Functional simulation can be repeated for different input combinations, and timing analysis can be performed for different input transitions. Since such

invocations of the simulator are independent, they can easily be parallelized when the same circuit must be simulated many times.



(a)

input		*output*
x	y	NAND
0	0	1
0	1	1
1	0	1
1	1	0

(b)

input			*output*
x	y	z	$F(x,y,z)$
0	0	0	0
0	0	1	1
0	1	0	0
0	1	1	1
1	0	0	0
1	0	1	1
1	1	0	1
1	1	1	1

(c)

Fig. 2.2. Simulating a small logic circuit: (a) graphical depiction, (b) truth table of a NAND gate, (c) simulated values at the circuit's output in truth-table form.

Simulating Multiple Input Combinations

Linear-time algorithms for logic and timing simulation achieve their efficiency by abstracting away unnecessary details. Further abstraction helps in simulating multiple input combinations or input transitions at once, which can be useful to compute the worst-case (critical) delays through the circuit, in order to determine the highest possible clock frequency. Logic simulation for multiple input combinations can determine if the circuit can ever produce incorrect results or enter a prohibited state. Extending timing analysis to estimate worst-case delays is relatively easy if the delay of every gate is represented by the worst possible case, regardless of the functional values at the gate's inputs. This pessimistic assumption facilitates a linear-time algorithm which slightly over-estimates the worst possible delay between any input transition and the stabilization of the circuit outputs [39].

Extending logic-level simulation to handle many input combinations at once is rather difficult. *Exhaustive simulation*, that is, trying all possible input combinations one by one, requires prohibitive amounts of time for circuits with more than about 30 inputs. *Symbolic simulation*, a technique developed in the 1980s, attempts to reuse and share computations normally performed for different input combinations. The term "symbolic" refers here to the fact that the information propagated through the circuit during a topological traversal does not consist of individual numeric parameters, but rather of graph-based data structures (symbols) that capture multiple parameter values through an abstract transformation. Such data structures are introduced in Section 2.2

and implicitly capture large sets of value combinations by means of specially-designed directed acyclic graphs [39].

Symbolic simulation starts by representing all input combinations by a rather straightforward graph data structure. It proceeds by (symbolically) applying individual gate operations to this data structure, which results in implicit representations of intermediate value combinations. The algorithms involved traverse and modify the graphs that encode these combinations. As a result, the data structures typically grow in size. After symbolically simulating all gates, one obtains a representation of all possible output value combinations. Dedicated algorithms can then verify if prohibited combinations appear on the outputs. This technique can be quite helpful for small- and medium-sized circuits in that it saves a great deal of runtime compared to exhaustive simulation. However, it often requires much more memory, and on large circuits gives rise to the so-called *memory explosion* problem, when memory requirements skyrocket after the data structure reaches a certain size. Symbolic simulation is also notoriously difficult to parallelize.

In one technique that is fundamental to both single-input and symbolic simulation, gate operations are not applied one by one, but are first conglomerated into clusters that are modeled by larger gates (this can be done once, but will speed up simulation of many inputs). This technique is illustrated in Figure 2.2 and removes the need to simulate signals within the clusters. It may, however, be limited by the size of look-up tables that represent these larger gates. To circumvent this limitation, symbolic simulation can represent the function of a larger gate implicitly (symbolically) by a specially-designed graph, as shown in later sections. Simulation algorithms are then extended to work with value-combination graphs and function graphs, rather than explicit look-up tables. Other possible extensions involve partitioning large graphs and performing symbolic simulation on one partition at a time.

Circuit Verification

Simulation can be used to verify the functional correctness of a given circuit. For example, if the circuit in question is an optimized variant of another circuit that is known to be correct, one can formulate an equivalence-checking problem to confirm that the two circuits always produce the same outputs when given identical inputs. If the two circuits are not equivalent, one seeks an input combination for which the outputs differ. Exhaustive simulation of all possible input combinations suffices when the number of inputs is small. To check the equivalence of two larger circuits, it is common to use a "miter" circuit, which connects the target circuits to the same input source and produces 0 when their outputs are identical and 1 when their outputs differ. The miter is formed by connecting the corresponding outputs of the circuits being compared to XOR gates; the outputs of the XORs are then combined by an OR gate. This construction reduces equivalence checking to the task of

checking if the miter always produces 0, which can be addressed efficiently by symbolic simulation.

In later chapters, we will draw some parallels between the simulation of quantum and conventional logic circuits—simulating quantum superposition is similar to simulating multiple input combinations at the same time. To this end, we will extend symbolic simulation to the quantum case by viewing (square) matrices that capture quantum operators as analogous to the truth tables used to simulate conventional gates. On the other hand, important differences exist, which include the necessary use of complex-valued arithmetic when modeling quantum circuits, as well as several different notions of equivalence. Another major difference is that quantum circuits often exhibit probabilistic behavior, while conventional logic circuits are deterministic. As is the case with conventional electronic circuits, physically-accurate modeling is possible for quantum circuits based on partial differential equations (such as the Schrödinger equation), but such detailed simulation is beyond the scope of this book.

2.2 Simulation with Binary Decision Diagrams

This section describes an important data structure that has been found extremely useful for symbolically simulating classical logic circuits. The key property of this data structure is that it can represent many signal combinations in a compressed form which can be manipulated directly without decompression. Such compression makes it possible to represent exponentially many signal combinations in a format whose size is linear in the number of inputs. As we will see later, the ability to exponentially compress many data states in this way is especially important for the simulation and verification of quantum circuits. First, we need to understand the original data structure from which the quantum techniques are derived, namely the *binary decision diagram* (BDD).

Binary Decision Diagrams

The BDD was introduced by Lee in 1959 [51] in the context of classical logic circuit design. It represents a Boolean function $f(x_1, x_2, ..., x_n)$ by means of a directed acyclic graph (DAG); see Figure 2.3. By convention, the top node of a BDD is labeled with the name of the function f represented by the BDD. Each variable x_i of f is associated with one or more nodes that have two outgoing edges labeled *then* (solid line) and *else* (dashed line). The *then* edge of node x_i denotes an assignment of logic 1 to x_i, while the *else* edge represents an assignment of logic 0. These nodes are called *internal nodes* and are labeled by the corresponding variable x_i. The edges of the BDD point downward, implying a top-down assignment of values to the Boolean variables depicted by the internal nodes.

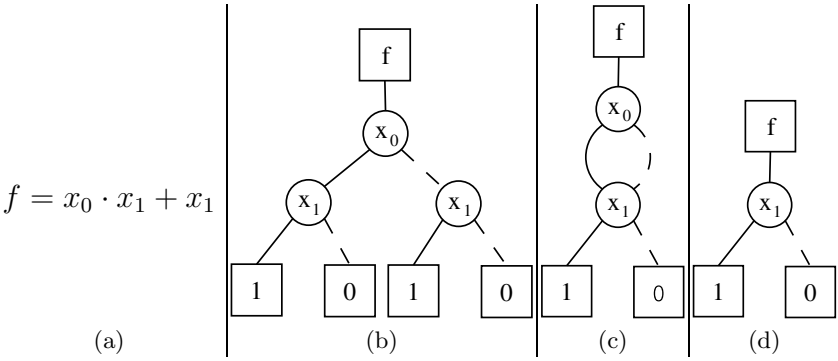


Fig. 2.3. (a) A logic function, (b) its BDD representation, (c) its BDD representation after applying the first reduction rule, and (d) its ROBDD representation.

At the bottom of the BDD are box-shaped *terminal nodes* containing the logic values 1 or 0. They denote the output value of the function f for a given assignment of its variables. Each path through the BDD from top to bottom represents a specific assignment of 0-1 values to the variables $x_1, x_2, ..., x_n$ of f, and ends with the corresponding output value $f(x_1, x_2, ..., x_n)$. For example, Figure 2.3b shows a BDD representing the two-valued Boolean function $f(x_0, x_1) = x_0 \cdot x_1 + x_1$. To determine, the value of $f(0, 1)$, one proceeds downward from the top node f to the internal node x_0. Then the outgoing *else*(dashed) branch from node x_0, corresponding to $x_0 = 0$ is taken to internal node x_1. Finally, the outgoing *then* (solid) branch from x_0 is traversed leading to a terminal node marked 1. One therefore concludes that $f(0, 1) = 1$.

The original BDD data structure conceived by Lee has exponential memory complexity $\Theta(2^n)$, where n is the number of Boolean variables in a given logic function. The reason for this complexity bound is that in Lee's initial design, the paths corresponding to all possible 2^n combinations of variable assignments are explicitly represented, as in Figure 2.3b. Moreover, exponential memory and runtime are required in many practical cases, making this data structure impractical for simulation of large logic circuits. To address this limitation, Bryant developed the *reduced ordered BDD* (ROBDD) [20], where all variables are ordered, and decisions are made in that order. A key advantage of the ROBDD is that variable ordering facilitates an efficient implementation of rules that automatically eliminate redundancy from the basic BDD representation. These *reduction rules* may be summarized as follows:

Reduction Rule 1. There are no nodes v and v' such that the subgraphs rooted at v and v' are isomorphic
Reduction Rule 2. There are no internal nodes with *then* and *else* edges that both point to the same node

An example of how the rules transform a BDD into an ROBDD is shown in Figure 2.3. The subgraphs rooted at the x_1 nodes in Figure 2.3b are isomorphic. By applying the first reduction rule, the BDD in Figure 2.3b is converted into the BDD in Figure 2.3c. Notice that in this new BDD, the *then* and *else* edges of the x_0 node now point to the same node. Applying the second reduction rule eliminates the x_0 node, producing the ROBDD in Figure 2.3d. Intuitively, it makes sense to eliminate the x_0 node since the output of the original function is determined solely by the value of x_1. An important aspect of redundancy elimination is the sensitivity of ROBDD size to the variable ordering. Finding the optimal variable ordering is an NP-hard problem, but efficient ordering heuristics have been developed for specific applications. Moreover, it turns out that many practical logic functions have ROBDD representations that are polynomial (or even linear) in the number of input variables [20]. In addition, the reduction rules make ROBDDs *canonical*, which means that no two different ROBDDs represent equivalent Boolean functions. Thus, equivalence of ROBDDs can be checked in $O(1)$ time by simply comparing the root nodes. Consequently, ROBDDs have become indispensable tools in the design, simulation, and synthesis of classical logic circuits.

Associated Algorithms

Even though the ROBDD is often quite compact, efficient algorithms are necessary to make it practical for logic circuit simulation. Thus, in addition to the foregoing reduction rules, Bryant introduced a variety of ROBDD operations whose complexities are bounded by the size of the ROBDDs being manipulated [20]. Of central importance is the **Apply** operation, which performs a binary operation with two ROBDDs, producing a third ROBDD as the result. It can be used, for example, to compute the logical AND of two logic functions. **Apply** is implemented by a recursive traversal of the two ROBDD operands. For each pair of nodes visited during the traversal, an internal node is added to the resultant ROBDD using the three **Apply** rules defined graphically in Figure 2.4. To understand these rules, some notation must be introduced. Let v_f denote an arbitrary node in an ROBDD f. If v_f is an internal node, $Var(v_f)$ is the Boolean variable represented by v_f, $T(v_f)$ is the node reached when traversing the *then* edge of v_f, and $E(v_f)$ is the node reached when traversing the *else* edge of v_f.

Clearly the **Apply** rules depend on the variable ordering. To illustrate, consider performing **Apply** using a binary operation *op* and two ROBDDs f and g. **Apply** takes as arguments two nodes, one from f and one from g, and the operation *op*. This is denoted as $\mathbf{Apply}(v_f, v_g, op)$. **Apply** compares $Var(v_f)$ and $Var(v_g)$ and adds a new internal node to the ROBDD result using its three rules. These rules also guide **Apply**'s traversal of the *then* and *else* edges (this is the recursive step). For example, suppose $\mathbf{Apply}(v_f, v_g, op)$ is called and $Var(v_f) \prec Var(v_g)$. Rule 1 is invoked, causing an internal node containing $Var(v_f)$ to be added to the resulting ROBDD.