Low Power Hardware Synthesis from Concurrent Action-Oriented Specifications

Gaurav Singh · Sandeep K. Shukla

Low Power Hardware Synthesis from Concurrent Action-Oriented Specifications

Springer

Gaurav Singh
Intel Corporation
S. Mopac Expressway 1501
78746 Austin
TX, USA
gasingh@vt.edu

Sandeep K. Shukla
Virginia Tech
Bradley Department of Electrical &
Computer Engineering
Whittemore Hall 302
24061 Blacksburg
VA, USA
shukla@vt.edu

ISBN 978-1-4419-6480-9 e-ISBN 978-1-4419-6481-6
DOI 10.1007/978-1-4419-6481-6
Springer New York Dordrecht Heidelberg London

Library of Congress Control Number: 2010930047

To the friends and families,
and all our fellow Hokies.

Preface

Human lives are getting increasingly entangled with technology, especially computing and electronics. At each step we take, especially in a developing world, we are dependent on various gadgets such as cell phones, handheld PDAs, netbooks, medical prosthetic devices, and medical measurement devices (e.g., blood pressure monitors, glucometers). Two important design constraints for such consumer electronics are their form factor and battery life. This translates to the requirements of reduction in the die area and reduced power consumption for the semiconductor chips that go inside these gadgets. Performance is also important, as increasingly sophisticated applications run on these devices, and many of them require fast response time.

The form factor of such electronics goods depends not only on the overall area of the chips inside them but also on the packaging, which depends on thermal characteristics. Thermal characteristics in turn depend on peak power signature of the chips. As a result, while the overall energy usage reduction increases battery life, peak power reduction influences the form factor.

One more important aspect of these electronic equipments is that every 6 months or so, a newer feature needs to be added to keep ahead of the market competition, and hence new designs have to be completed with these new features, better form factor, battery life, and performance every few months. This extreme pressure on the time to market is another force that drives the innovations in design automation of semiconductor chips.

If one considers high-end servers, workstations, and other performance-hungry systems, their corresponding semiconductor chips also need to resolve many of these market and technological forces. However, the time scale and the scale of betterment from generation to generation are quite different. In this book, our focus is not on these high-performance computing systems. Our focus is rather on the specific, battery-driven consumer devices, in particular the semiconductor ICs or chips inside them.

However, we are not concerned here with the architecture or the design specifics of these chips, but rather with the process of designing them. These are mostly implemented as ASICs (Application-Specific Integrated Circuits) or sometimes on FPGAs (Field Programmable Logic Arrays) for computing specific functions or algorithms. For example, one could design a chip for encryption or decryption of bit streams or one could design a chip for matrix multiplication or Fast Fourier

Transform. One could also have complex interfaces in these chips, such as an AMBA bus interface, or a DMA interface, or some other kind of communication or I/O interface. These would mean that not only such chips do specific computation, they also have multiple threads of control. The computation itself may also be designed with multiple threads of control, because hardware does allow us to overlap computations that do not have to be sequenced in a specific order. However, such concurrent threads of computation require synchronization and communication with each other. Most computation and communication functions have such concurrency and synchronization between concurrent threads of computation.

The question is how to design such a highly concurrent hardware system and fabricate it on a chip, while resolving various constraints on performance (e.g., latency), power consumption (e.g., peak power, leakage, overall energy), area, etc. This is a question of innovative design methodology, design environment, language for design entry, transformation, abstraction, and many other issues. Electronic Design Automation (EDA) community has been debating these questions for many years, but in the last 10 years or so, this question has become one of the prime movers of the industry. Designing complex chips that satisfy requirements and design constraints and delivering them fast enough to meet the time-to-market goals is the central issue of design automation these days.

Semiconductor IC design process has gone through a long history in the last 40 or so years, starting from SSI (Small-Scale Integration), MSI (Medium-Scale Integration), LSI (Large-Scale Integration), VLSI (Very Large-Scale Integration), ULSI (Ultra Large-Scale Integration), etc. Moore's law as predicted by Gordon Moore is still going on with full steam, and we are seeing a doubling of the number of transistors per unit area of the chip every 2 years approximately. As a result, more complex functionalities are being implemented on chip, and thus the need for ultra large-scale integration – UVLSI. (The term UVLSI did not catch up, and the term VLSI is still pervasive in the literature, even though since the time the term VLSI was introduced, the scale has improved by orders of magnitude.)

In the beginning, the designers used to design each transistor by hand, tweaking their parameters to make sure that required characteristics are implemented correctly. Then came gate-level design era, where schematics were initially hand drawn, and later various automated tools arrived. However, as the industry moved from hundreds of gates to thousands of gates on the die, the gate-level design techniques and corresponding automation processes also ran out of steam. The advent of register transfer level (RTL) languages in the early 1990s ushered a new era of productivity in silicon chip design. Capturing the state of the system with registers and state transition conditions with combinational logic facilitated the designers' ability to make much faster design entry into the automation flow. Initially, this abstraction from gate-level design to register transfer level was meant to provide better simulation abilities. A design described only with gates would take much more time to simulate for a few million clock cycles than if it were described with RTL. VHDL was conceived as a simulation language, and so was Verilog. Pretty soon various logic synthesis algorithms started to show up in the literature, and eventually logic synthesis allowed the large-scale proliferation of the use of RTL

as a design entry language for most semiconductor IC companies by the late 1990s. The use of RTL thus reduced validation time by speeding up simulation and reduced the design time by virtue of optimized logic synthesis algorithms and tools.

However, the designers started to demand more out of the automation tools and tried to introduce higher level constructs into RTL, and *behavioral RTL* constructs found ample designer attention. The original variant of RTL called the *structural RTL* described the architecture of the design in terms of modules, the combinational logics with interconnected gates, and the state machines in terms of registers and their update logic. Behavioral RTL aimed at easing the burden on the designers by allowing them to describe the designs in terms of their behaviors rather than the architectural structures (e.g., a state machine described as a state transition system, and not as an interconnection of registers and gates). Algorithms for logic synthesis from behavioral RTL became a popular topic of research. Given a behavioral state machine description, the synthesis algorithm is required to choose among various possible structural implementations of the state machine. For example, one could vary the state encoding (e.g., binary vs. one-hot vs. Huffman encoded) and obtain various area, power, latency characteristics of the design. Instead of the designer having to make the choice, the synthesis algorithm is supposed to do a design space exploration and choose the best implementation based on the design constraints. This means that the synthesis algorithm had to be made aware of constraints such as area, power, latency. The exploration, however, is non-trivial because depending on such design constraints, one encoding vs. another would be more appropriate. This meant that the synthesis algorithms have to solve multi-objective optimization problems before making decisions on what to synthesize. A number of tools such as *Behavioral Compiler* attempted this, but did not gain popularity because expert designers thought that they could do a better job through their experience and intuition on what makes the best hardware for the given set of constraints.

One of the most important reasons for promoting the behavioral style of RTL came from the fact that the chips being designed got increasingly complex, requiring various parts executing sophisticated protocols for communicating with the outside world as well as among its own components. Protocols are best described behaviorally than structurally. Thus *protocol compilation* was another name for behavioral synthesis. There has been mixed reaction to such tools and methodologies, and it was suspected by expert designers to be producing non-optimal implementations.

In the late 1990s and early part of 2000, a push for higher abstraction level than RTL came about. A number of activities related to C/C++-based design entry languages for hardware were announced, including SpecC from the University of California – Irvine, Synopsys' Scenic, later named SystemC, Cynapp's Cynlib, IMEC's OCAPI, etc. Also, enhancing Verilog into Superlog and VHDL into Object-Oriented VHDL were announced around the same time. It was clear that without raising the abstraction level for hardware design entry, it is hard to keep up with the increasing productivity requirements in the semiconductor industry.

The introduction of high-level software languages as carriers for RTL description does not, however, enhance the abstraction level. It only allows one to compile the hardware description with traditional software compilers and, therefore, provides

a free simulation environment. But the cost of RTL simulator is not the biggest concern for the industry, and hence a need for higher abstraction level was given serious thoughts. More academic languages such as SpecC already had introduced a number of conceptual abstractions such as behaviors as processes, channels for communication, channel refinement for communication protocol elaboration, events, and various synchronization between behaviors and events. A stepwise refinement strategy and taxonomy formed the core of the SpecC methodology. Some of these abstractions found their way into the SystemC-2.0 specification and ushered the era of transaction-oriented design entry. The terminology of transaction level model or TLM came about very soon after that, and various levels of transaction-level descriptions were prescribed and adopted into the SystemC language.

The transaction-oriented description of functionalities of a hardware system (or hardware/software system) allows one to abstract away from descripting the bit-level details of the design, especially the communications between modules within a design. Transactions not only allow abstracting away the data-type representation from bits to high-level data types but also allow temporal abstraction. For example, a data transfer over a bus between a module and memory could take a number of clock cycles, and the protocol may be described by a cycle-by-cycle description of what bits get set and what bits get reset at each cycle. This entire process can be replaced by a simple transaction. As a result, transaction-level models can be simulated much faster (orders of magnitude) compared to simulating RTL models.

Since transactions abstract away precise cycle-by-cycle behavior, synthesizing code from transactional model into real hardware brought the old pain back. Now, the synthesizer has to solve multi-objective optimization problem to select from various possible elaboration of transactions into bit level, cycle-by-cycle behavior. If such optimization problem is not solved fast and accurately, the synthesized hardware will be possibly suboptimal. This became a cause of pain for a while, and even today, this problem is not entirely solved. However, progress has been made in two directions. Synthesis of acceptable-quality RTL from TLM models has been done by a number of companies – Forte Design Systems' (now erstwhile Cynapps) Cynthesizer has much of such capabilities. Mentor's Catapult-C, initially started as an algorithmic C to RTL synthesizer, also has adapted itself so it can handle a lot of transactional constructs and synthesize quality RTL. The second technological innovation that allowed this trend of TLM-based design entry to proliferate further was Calypto System's sequential equivalence verification engine that can compare a TLM model and the generated RTL to verify their sequential equivalence.

Having reached this status has enabled the design industry, especially ASIC industry, to progress toward bridging the notorious productivity gap in the industry. However, a lot more is desirable, and TLM with higher abstraction is still out of the scope for much of the automated synthesis tools. TLM synthesis and verification and transference of the verification assurance at the TLM level to the resulting RTL level are still topics of research. Such research papers are in abundance at most design automation-related conferences today.

Recall that the ability to synthesize is not the only criteria for success here, nor is the ability to carry out automated sequential equivalence verification. The quality

of the generated RTL in terms of latency, power, area, etc., is very important, and active research is being carried out along those lines. Many of the tools mentioned above also allow the users to specify timing, power, and area budget constraints and accordingly try to synthesize RTL that meets those bounds. However, estimating power, area, or timing from a TLM-level model is another hard problem. For example, power consumption of a hardware design is highly dependent on the technology being used, details of the physical structure of the design, clock frequency, voltage levels, interconnects and clock tree, etc. Therefore, optimizing the result of synthesis from TLM models for power has to be based on a number of assumptions, profiling of past designs, and various intuitions formulated as numeric guesses. Many of these have been formalized in terms of statistical regressions over other parameters such as toggle counts, state transitions, transaction duration, and width, and research in this field is being carried out vigorously as we write this preface.

As this push toward abstraction was warming up, an alternative approach started taking shape at the Massachusetts Institute of Technology (MIT). James Hoe and Arvind started looking at the specification of hardware in terms of *term rewriting systems*. Term rewriting is a topic extensively studied by the automated theorem-proving community. For example, the steps to proving an algebraic identity can be captured by rewriting rules. Starting from the left-hand side of the identity to be proven, one applies appropriate rewrite rules to arrive at the right-hand side term. Computer algorithms that efficiently figure out which rewrite rules to apply in what order are at the core of term rewriting systems. It seems that the intuition behind Hoe and Arvind's work was as follows. One could conceive of the current state of a hardware system as a term over an appropriate term algebra and the transitions of the system as rewrite rules. Therefore, the evolution of a hardware system in its state space can be looked upon as applications of rewrite rules from the term denoting the initial state. This approach to hardware specification turned out to be abstracting the system in a direction that transactions usually cannot. The concurrency in hardware systems (i.e., one set of registers changing state independent of another set of registers changing state, at the same clock cycle) gets captured by the different rewrite rules. One could actually apply multiple of these rules to the same term (state) to express concurrent evolution in the state space. More importantly, the parts of the state space (registers) on which the multiple rules are applied must be disjoint; otherwise, there will be attempts to change the state of the system by two distinct transitions at the same time, leading to *race conditions*. This *non-interference* property is essential for representing the correct transitions in the state space. In the concurrency literature this is known as *atomicity* of the transactions, or the rules. Finding which rules can be applied concurrently without violating atomicity is not difficult, if one knows exactly which parts of the state these rules try to modify. One can declare those rules which modify overlapping parts of the state space as "conflicting" and create a conflict graph. The conflict graph will have nodes denoting rules and edges denoting conflict. Thus, finding the rules which can be concurrently applied in the same clock cycle is equivalent to finding the *maximal independent set* in the conflict graph. Finding the maximum size-independent set is an NP-complete problem, but finding maximal such set can

be heuristically done efficiently. This led to the scheduling algorithms required to synthesize RTL from such kind of hardware specification. Hoe and Arvind created a new paradigm of hardware specification and synthesis of RTL from such specifications by computing schedules of rules that apply in each clock cycle. The RTL they created had this scheduler inbuilt into the hardware. Later this concept evolved into the language *Bluespec*. The Bluespec language was first used in a network processor design company and was later developed into a product by an EDA company, also named Bluespec Inc. Both of these companies were started by Arvind and his colleagues.

It turns out that the idea germane to the term rewriting view of the state transitions had other variants in the literature. Dijkstra had introduced the notion of a guarded command language, where a system state is spread across multiple variables. A guarded command is a rule that updates those variables based on certain conditions. Given a set of guarded commands, the update process is based on rounds of non-deterministic choices. The choice at each round is among the set of guarded commands whose guard evaluates to true over the current state. This is a simple model to describe interleaving concurrent evolution of the state space. Even if one command is executed per round, it could model concurrent execution of commands. For two successive commands that do not overlap in the set of variables they modify, the effect of executing them sequentially is equivalent to that of executing them concurrently. Chandi and Mishra later extended this notion and defined the UNITY language based on guarded commands for parallel program specification. Any possible execution sequence of commands allowed by the program is a behavior of the specified program. An implemented program, however, may not have all these behaviors but some of the possible ones. Thus, an implementation satisfies such a specification if the set of possible behaviors of the implementation is a subset of the set of possible behaviors of the specification.

A number of computer scientists including Leslie Lamport suggested that interleaving semantics of parallel programs is the appropriate semantics. This means that if you have two guarded commands a and b which can execute at the same time in parallel, then the effect of that is equivalent to first executing a and then b or vice versa. Diagrammatically, this leads to a diamond representation, often called the diamond rule. If a and b are non-interfering (i.e., they do not modify the same variables or state elements), it makes sense to view their execution this way. However, if execution of a modifies a variable that is being used in the guard of b, then after execution of a, the guard of b may not hold true any more. Hence, it would not be possible to have the same effect as a and b executing in parallel. In the interleaving semantics, even parallel execution of a and b would not be allowed in such a case, because there is no equivalent interleaving of such guarded commands.

The other school of thought about parallel program specification at that time was championed by Vaugh Pratt at Stanford and his colleagues. This was termed "partial order" semantics of concurrency. According to this semantics, interleaving semantics misses out many possible interactions between concurrent executions and is not sufficient to capture all possible behaviors one could see in concurrent

systems. Also, they came up with examples of scenarios where two concurrent systems that are indistinguishable in terms of interleaving semantics are distinguishable by partial order semantics, based on partially ordered multiset (POMSET) models of behaviors.

There was a raging debate between these two schools in the early 1990s over an Internet-based mailing list called the "concurrency mailing list." The often heated exchange of messages on the mailing list between Pratt and Lamport is interesting to read and is available in the appendix of a book on partial order models of concurrency edited by Holzmann and Peled, published in 1996. However, here we need not concern ourselves with POMSETs and partial order semantics. Hardware system behaviors are usually observed as traces of inputs and outputs as the state space evolves in reaction to the inputs that are provided to it by the environment. As a result, a trace-based semantics is sufficient for our purposes.

A lot of the work described in this book is based on our experience in working with the Bluespec engineers and researchers. However, in order to make the model of concurrent atomic rules (also called actions) independent of a specific company, we adopted the idea of CAOS (Concurrent Action-Oriented Specifications) which is a Bluespec-like guarded atomic action-based language. In this language, the actions have guards like guarded command languages and Bluespec. The guards are predicates evaluated over the state of the system. At every round of execution, guards are evaluated, and the rules or actions whose guards evaluate to true are called *enabled actions*. The execution of the rules can be carried out in many different ways. The simplest one is to choose one of the enabled rules non-deterministically, execute it, and record the change in the state. In the next round, guards are reevaluated and a rule is selected among enabled ones. This goes on ad infinitum or until no guard is evaluated to true in a round. This is considered the *reference semantics* of any CAOS model. This describes all possible allowable behaviors of the specification. If a certain behavior is undesirable, the actions and their guards may be suitably modified, and possibly more state elements are added to modify the model such that undesirable behaviors are eliminated.

If one's intention is to create a hardware system implementing the specification, it is not a good idea to execute one rule at a time. This is because one would map each round of the model execution to one clock cycle of the actual hardware. Therefore, executing one rule at every cycle will provide an unacceptable latency of the system. To obtain better latency, one should attempt to execute as many rules as possible per round. This reduces to determining which rules are non-conflicting and then selecting a maximal set of such rules per round (clock cycle). But one has to be careful at this point. We have declared the behaviors generated by executing one rule per round as the reference semantics of our model. Thus, by executing multiple non-conflicting rules in the same round, if we create a behavior which has no equivalent behavior in the reference semantics, we will be violating the specification. As a result one has to worry about an extra constraint while scheduling – even when two rules a and b are non-conflicting, one is allowed to execute them concurrently in the same round, if and only if the resulting change in state can also be effected by either executing a first, and then b, or vice versa. This is important for correctness.

Thus, the scheduling process must know which pair of non-conflicting rules would violate this particular constraint and never schedule them together in a single round. A simple example would be two rules written as follows: R_1 : true $\rightarrow$ $x := y$ | R_2 : true $\rightarrow$ $y := x$. These two rules R_1 and R_2 are non-conflicting and when executed concurrently, they swap the values of x and y. However, if you execute R_1 followed by R_2, then x and y both end up having the last value of y. If you do the opposite, both x and y end up having the last value of x. Hence, the rules R_1 and R_2, albeit non-conflicting, cannot be executed in the same round, concurrently.

Once this restriction is understood, creating a scheduler and embedding it into the generated RTL is not very difficult. Thus, synthesizing RTL from CAOS or Bluespec is not a problem at all. In fact, in a 2004 ICCAD paper, Arvind et al. showed that the generated RTL using Bluespec is often competitive in terms of area and latency against the handwritten Verilog RTL for the same functionality. More interestingly, the generated RTL's structure is quite easy to understand.

As mentioned before, the computation of the schedule that executes maximum possible number of rules in every cycle is computationally hard but the heuristics perform well in most cases. One could, however, show contrived examples where even the heuristics can produce results with latencies arbitrarily far from the optimal. But such contrived examples do not happen in practice, and the Bluespec synthesis does produce hardware that has required area and latency. One of the chapters in this book will look deeper into the algorithmic complexity of the scheduling problems related to such CAOS-based synthesis and their approximability with heuristics. Such analyses are important for the sake of understanding the heuristics and their corner cases, but in practice they do not make much difference.

The 2004 ICCAD paper claimed nothing about the power consumption or peak power of the designs generated using Bluespec. That is where this book contributes the most to the CAOS-based synthesis processes. Since executing all non-conflicting rules in the same cycle might reduce latency, there may be a tendency to do so. But that may have several implications – the peak power of the hardware will rise, leading to thermal issues, degradation of performance, and stronger cooling and packaging requirements. Thus, it is not inconceivable to slow down the system a bit by not scheduling all the rules that could be scheduled in the same cycle, but rather stagger them a bit. To do this, one has to have an estimate of the power consumption of each hardware resource that is engaged in the execution of a rule, create a metric of peak power per rule, and accordingly solve an optimization problem. More interestingly, selecting some rules for execution and leaving the rest for later cycles might mean that the behavior of the design has changed. Nevertheless, such selective execution of the rules in a clock cycle can be done as long as the resultant behavior is still included in the set of possible behaviors under the reference semantics.

This led us to another interesting work described in this book. How do you know that when you change the behavior of a design by disallowing some enabled rules from executing in a clock cycle, you are still adhering to the original specification? Neither Bluespec nor CAOS had a formal verification methodology at that point. In fact, the question of formal verification is a bit tricky in this context. One

possibility is to formally verify the generated RTL using standard model checking techniques. But this does not provide any advantage over standard RTL verification techniques used in other methodologies for hardware design. Since the one rule per round semantics embodies the reference semantics for the specification, it would be best to prove that the set of behaviors produced by the synthesized hardware is a subset of the behaviors allowed by the reference semantics. This means that standard model checking is not what we want, but rather automata theoretic language inclusion-based formal verification is more appropriate. There are not many tools out there which (i) allow automata theoretic formal verification and (ii) have the facility to model concurrent atomic actions in a straight-forward manner. The one freely available tool which has a great reputation in software verification is SPIN from Bell Labs. However, SPIN does not have a notion of clock cycle, and hence requires addition of a clock process if clocked hardware needs to be modeled. Having managed these modeling issues, we were able to model the reference behavior and the behavior of the generated hardware into SPIN and carry out automata theoretic verification. The interesting results we obtained were as follows: if a maximal set of non-conflicting rules are scheduled per cycle then the set of behaviors we obtain is correct with respect to the reference model. So is the peak power saving-based scheduling model which selectively executes rules based on a peak power constraint. But the peak power saving model and the standard RTL model (which executes maximal set of non-conflicting rules) actually may not have the same set of behaviors. This is a bit of a conundrum, because one would think that peak power saving model would have behaviors that are special cases of the standard RTL behavior, but this is not the case always. This is due to the unique nature of the CAOS-type modeling paradigm. The model is non-deterministic in its reference semantics, and hence a large number of different behaviors are allowed, and any implementation could actually pick to implement a subset of these behaviors. So two distinct implementation strategies might produce very different behaviors, both of which are correct with respect to the reference specification.

One way out of this dilemma would be to always use the same synthesis strategy, so that behaviors of the synthesized RTL are always the same. But that means any new optimization trick would be difficult to incorporate into the synthesis process. Another way is to suitably restrain the CAOS model with lots of constraints in the guards, such that the model has deterministic behavior. In other words, the reference semantics has only a very restricted set of behaviors. But this means constraining the CAOS model very tightly and burdening the designer with all the worries of handling synchronizations to determinise the model. We suspect that such an approach would make hardware specification with CAOS as error prone and hard as with traditional RTL specification.

As we alluded to earlier, peak power reduction is an important aspect of chip design due to thermal and packaging considerations. But overall energy reduction for increasing battery life is also another aspect of power optimization. This field is well studied for over a decade, especially with clock-gating and operand isolation techniques. Clock-gating essentially helps in reducing register power, by gating the

clocks to the registers that do not change their values in a particular clock cycle. The gating logic can be inserted automatically during the synthesis process. Operand isolation is a process of isolating combinational logic from its inputs when the corresponding output is not being used in the current clock cycle. Both of these techniques have shown to save average dynamic power. For CAOS-based synthesis, one could essentially implement these techniques by analyzing the operations involved in each action or rule. Specifically, when the guard of an action is evaluated to false in a clock cycle, we know that the corresponding state updates will not take effect in that cycle. However, if there are other rules which might also update the same state element (register) and if one of those rules is enabled and selected for execution by the scheduler, then one has to make sure that the clock-gating of that register does not get turned on in that particular cycle. This requires more involved analysis of data dependency between various rules. In order to take some of these decisions efficiently during the execution of a design, extra logic needs to be added, which in turn is a potential source of additional power consumption. Thus, one has to figure out if the additional circuitry added to enable clock-gating and/or operand isolation is not offsetting the gains in overall power savings for the design. These techniques were implemented in the Bluespec synthesis tool called *Bluespec Compiler*. With specific options enabled in the compiler, one could effectively reduce the power consumption of the generated design. Power estimation techniques applied on various industrial benchmark designs have shown power gains. This book has a chapter on this particular topic reporting the techniques and detailed experimental results.

To sum up, in our experience, every time a new paradigm of modeling is introduced for hardware design entry, one has to create a design flow starting at specification of designs in that paradigm. But a design flow is not possible without automated synthesis techniques to generate the implementation at gate level or RTL. Once the synthesis algorithms are proven to be correct, one has to concentrate on resource optimizing synthesis techniques, especially targeting reduction in area, latency, power, etc. One might have to consider various trade-off points and accordingly create the appropriate tools and methodologies. The work described in this book is on CAOS-based hardware specification and was inspired by our association with Bluespec language and the company. Therefore, this work is focussed on various power reduction issues related to CAOS-based synthesis and the corresponding formalism, algorithms, complexity analysis, and verification problems.

We believe that it will provide the readers, especially research students who are entering the field of resource-constrained synthesis of hardware from high-level specifications, with a perspective and guide them into creating their own research agenda in related fields.

We are particularly indebted to Bluespec Inc. for their financial support in carrying out this research work [105–111] between 2005 and 2007, including multiple summer internships for the first author. A part of this research was also supported by a National Science Foundation PECASE award and a CRCD grant. We also received personal attention and help from Arvind, Rishiyur Nikhil, Joe Stoy, Jacob Schwartz, and others at Bluespec Inc. Sumit Ahuja at the FERMAT lab of Virginia Tech has been particularly helpful by providing many suggestions in power estimation and

power reduction techniques. Finally, this work was the basis of the Ph.D. dissertation [103] of the first author, and this Ph.D. work was carried out at Virginia Tech's Electrical and Computer Engineering Department. We are indebted to Virginia Tech for all the facilities.

Austin, TX
Blacksburg, VA
March 2010

Gaurav Singh
Sandeep K. Shukla

Acknowledgments

We acknowledge the support received from Bluespec Inc., NSF PECASE, and NSF-CRCD grants, which provided funding for the work reported in this book.

Contents

List of Figures

List of Tables

Acronyms

RTL	Register Transfer Level
HLS	High Level Synthesis
EDA	Electronic Design Automation
HDL	Hardware Description Language
CAOS	Concurrent Action Oriented Specifications
CDFG	Control Data-Flow Graph
HTG	Hierarchical Task Graph
GCD	Greatest Common Divisor
BSC	Bluespec Compiler
BSV	Bluespec System Verilog
LPM	Longest Prefix Match
VM	Vending Machine
LTL	Linear-time Temporal Logic
TLA	Temporal Logic of Actions
MCS	Maximal Concurrent Schedule
ACS	Alternative Concurrent Schedule
MNS	Maximum Non-conflicting Subset
MIS	Maximum Independent Set
MLS	Minimum Length Schedule
FFD	First Fit Decreasing
PTAS	Polynomial Time Approximation Scheme
AES	Advanced Encryption Standard
UC	Upsize Converter