



Achieving Power Efficiency in Hardware Circuits with Symbolic Discrete Control

Thesis submitted in accordance with the requirements of the University of Liverpool for the
degree of Doctor in Philosophy by

Mete Özbaltan

January 2020

Abstract

The power efficiency of hardware circuits is of paramount importance for constructing embedded electronic devices, as it is one of the major design constraints in today's embedded systems, limiting performance, battery life, etc. This thesis targets the power efficiency in hardware circuits with symbolic discrete control. In the research proposed in this thesis, we consider hardware circuits, described using the popular Hardware-Description-Language (HDL), Verilog, at the Register-transfer Level (RTL) abstraction, as hierarchical compositions of sub-circuits. We achieve power-efficiency by switching-off the clock of each sub-circuit according to some clock-gating logic, where the technique applied is known as RTL clock-gating, which is one of the best low-power technique applied on synchronous hardware circuits.

We advance the following approaches in order to produce a clock-gating logic: to switch-off the clock signal of a sub-circuit in the idle status, which is a set of values of the circuit signals when the values of the memory components do not change; to apply power-aware scheduling policies for data-flow hardware circuits implemented as Kahn-Process-Networks (KPNs), using the clock-gating logic as used to selectively filter the clocks of the sub-circuits involved; and to employ an energy-efficient configuration manager for choosing the optimal configuration, by means of the clock-gating logic, among the alternatives on data-flow hardware circuits implemented as KPN, with parallel synchronous processes. We devise a tool-supported framework for achieving power-efficiency of hardware circuits for each approach. Our approaches rely on formal control techniques, where the goal is to compute a strategy that can be used to drive a given model so that it satisfies a set of control objectives. More specifically, we give an algorithm that derives abstract behavioral models directly in a symbolic form from original designs, and for formulating suitable constraints and objectives. We encode the computation of the latter as several small symbolic Discrete-Controller-Synthesis (DCS) problems, and use the resulting controllers to derive power-efficient versions from original circuit designs.

Finally, we show how a resulting strategy can be translated into a piece of synchronous circuit that, when paired with the original design, ensures the aforementioned objectives. We detail and illustrate our approaches using various hardware designs and objectives, and validate them experimentally by deriving a low-power version of the original hardware designs.

Acknowledgements

First of all, I would like to thank my supervisors, Sven Schewe, Dominik Wojtczak, and Nicolas Berthier, for their support, advice, and guidance throughout my research and the writing up of my thesis. Their extensive experience and knowledge provided me with valuable advice and comments, during my PhD.

I would also like to thank Eric Rutten and Alexei Lisitsa for agreeing to be my examiners on my thesis committee.

I also extend my gratitude to people in the Department of Computer Science at the University of Liverpool for their assistance and encouragement.

I wish to express my sincere gratitude to The Republic of Turkey Ministry of National Education for supporting me as my sponsor.

Last, but not least, I would like to thank my family for their support.



Contents

Abstract	i
Acknowledgements	iii
Contents	vii
List of Figures	ix
List of Listings	xi
List of Tables	xii
1 Introduction	1
1.1 Problem Statement	3
1.2 Contributions	5
1.3 Thesis Outline	6
2 State of The Art on Low Power Synchronous Circuits Design	8
2.1 Integrated Circuits (ICs)	9
2.1.1 Classification of ICs	9
2.1.2 Architectures of ICs	11
2.2 Chip Design Process	13
2.2.1 Evolution of Chip Design	13
2.2.2 Abstraction Levels	14
2.2.3 Chip Design Methodologies	19
2.3 Hardware Description Languages (HDLs)	20
2.3.1 Data Types	23
2.3.2 Functional Blocks and Ports	24
2.3.3 Instantiations	24
2.3.4 Statements	25
2.3.5 Structural Representation	25

2.3.6	Data-Flow Representation	25
2.3.7	Behavioral Representation	26
2.4	Low Power Synchronous Circuits Design Techniques	27
2.4.1	Clock Gating Technique	28
2.4.2	Existing Approaches	30
2.5	Existing Tools Used on Chip Design	32
2.6	Summary and Discussion	34
3	Modeling Formalisms on Hardware Circuits Designs, & Symbolic Tools, Languages, and Techniques for Discrete Control	35
3.1	Modeling Formalisms Applied on Chip Design	36
3.1.1	Automata-Based Modeling Formalisms	36
3.1.2	Data-Flow Based Modeling Formalisms	37
3.2	Discrete Controller Synthesis (DCS)	39
3.3	BZR/Heptagon Synchronous Language	42
3.4	ReaX Synchronous Language	44
3.5	Existing Approaches	46
3.6	Summary and Discussion	48
4	Exercising Symbolic Discrete Control for Designing Low-Power Hardware Circuits: an Application to Clock-Gating	49
4.1	Introduction	49
4.1.1	Motivation	50
4.1.2	Contributions	51
4.1.3	Overview	51
4.2	Running Example	52
4.2.1	The Verilog Hardware Description Language	52
4.2.2	A Fragment of Heptagon	54
4.2.3	Variables & Further Notations	55
4.3	Computing CGLs using Symbolic DCS	56
4.3.1	Overview of the Modeling Technique	56
4.3.2	Building Sub-module Models & Idleness Predicates	59
4.3.3	Building Composed Module Models	61
4.3.4	Computing & Integrating the CGLs	62
4.4	Application on a Case Study	63
4.5	Summary and Discussion	65
5	Power-Aware Scheduling of Data-Flow Hardware Circuits with Symbolic Control	68
5.1	Introduction	69
5.2	Discrete Control with Symbolic Systems	70

5.2.1	Symbolic Notations	70
5.2.2	Symbolic Systems	71
5.2.3	Symbolic Control	72
5.2.4	Tooling and Related Works on Optimal Control	73
5.3	Models and Objectives for Power-aware Scheduling	74
5.3.1	Overview of the Approach and Contributions	74
5.3.2	Abstracting Process Implementation Behaviors	75
5.3.3	Abstract Process Observers & Control Means	78
5.3.4	Achieving Power-efficiency by Control	81
5.3.5	Enabling Dynamic CGL Reconfiguration	81
5.4	Experimental Evaluation	82
5.5	Summary and Discussion	85
6	A Case for Symbolic Limited Optimal Control: Energy Minimization in Data-flow Circuits	87
6.1	Introduction	88
6.2	An Algorithm for Symbolic Limited Optimal Control	88
6.2.1	Computing Expected Outcomes	89
6.2.2	Computing the Refined Strategy	90
6.3	Use-case: Energy-aware Configuration Management	91
6.3.1	Management of Configurable Designs	91
6.3.2	Overview of the Approach for Computing Managers	92
6.3.3	Abstract Channel Model	92
6.3.4	Abstract Process Model	94
6.3.5	Control Objectives	95
6.4	Experimental Evaluations	96
6.4.1	Constructing Simulators of Managed Designs	96
6.4.2	Simulation Bench	97
6.4.3	Simulation Results	101
6.5	Summary and Discussion	103
7	Conclusions and Future Recommendations	105
7.1	Conclusions	106
7.2	Open Issues and Future Recommendations	108
	References	111

List of Figures

1.1	Observation of the Feedback Control Problem	4
1.2	Design Flow of the Research Methodology Proposed in this Thesis	5
2.1	Structure of a Synchronous Circuit	10
2.2	Tile Based MPSoC Block Diagram	12
2.3	Y-Chart Design Model	15
2.4	Generic Chip Design Flow	21
2.5	CMOS logic/gate inverters invert the given value V_{in} , on the output value V_{out} , by using PMOS  and NMOS  transistors.	27
2.6	Basic Principle of The Clock Gating Technique	29
3.1	Representation of The Control Theory of Discrete Event Systems	40
3.2	Principle of The Discrete Controller Synthesis	41
3.3	Task $t(r, c, data_in) = a, data_out$, faking computation, in the form of a Mealy machine, where: r (request), c (control), and $data_in$ (input data) are input variables; a (active—keeps a memorized value, initially false) and $data_out$ (output data) are output variables; e (executed), cnt (counter—keeps a memorized value, initially 0), $comp$ (computation—keeps a memorized value, initially 0), and $S = \{Idle, Busy\}$ (state—keeps a memorized value, initially Idle) are local variables.	42
3.4	Computational Steps on The State Space X	46
4.1	Mealy machine symbolically encoded by register state in Verilog module m of List. 4.1, decorated with operations on cnt and m . The initial value of cnt is 0.	53
4.2	Example Verilog module instantiation graph, associated models, and resulting CIPs. Arrows \rightarrow (resp. $\cdots\rightarrow$) represents Verilog sub-module (resp. Heptagon node) instantiation relations. In turn, $- \rightarrow$ (resp. \rightsquigarrow) denotes modeling (resp. symbolic DCS) steps.	56
4.3	Interface of a Sub-module Model SM_M	59
4.4	Interface of a Composed Model CM_M	61

5.1	Overview of the possible work-flows for computing the power-aware CGL. . .	74
5.2	Extracts of symbolic model built from the example process of List. 5.1. . . .	77
6.1	Overview of our suggested work-flows for computing energy-aware configura- tion managers.	91
6.2	Automaton representation for 3-state channel model encoded as $q_{i,j}$	93
6.3	Graphical representation of a 2-process 2-channel configurable design; the $p_{i,1}, \dots, p_{i,n}$'s represent the distinct configurations of process i —similarly for j . Notice that r, cfg_i, cfg_j, e_i , and e_j are inputs of the system model, whereas $cons_i, prod_i$ and $cons_j$ denote expressions (predicates).	95
6.4	Synthesis & simulation tool-chain for assessing global design objectives. . .	96
6.5	Example of configurable design with 6 processes.	101
6.6	Gain in simulated energy per completely processed data for the design with 6 processes illustrated in Figure 6.5. For each size of sliding window (x-axis), we give the minimum, average, and maximum gain in percentage, as well as the low and high quartiles. A window of size 0 indicates that the manager is produced without any enforcement of optimal control objective. All gains (y-axis) are given in percentage of the average energy consumption obtained for the non-optimized design (represented with the dashed horizontal line). .	102

List of Listings

2.1	Verilog module <code>m</code> , faking computations on data given on input wires “ <code>data_in</code> ” upon request “ <code>start</code> ”. Output wires “ <code>data_out</code> ” carry its result, available when “ <code>done</code> ” becomes “1”.	22
2.2	Verilog module <code>M</code> , instantiating <code>m</code> of List. 2.1 twice, as <code>m1</code> and <code>m2</code> , in a pipelining mode where <code>m1</code> feeds <code>m2</code>	23
3.1	Heptagon node <code>t</code> , symbolic encoding of the Mealy machine of Figure 3.3. . .	43
3.2	Heptagon node <code>twotasks</code> , parallel composition of two instantiations of the node (task) <code>t</code> of List. 3.1.	43
3.3	Heptagon node <code>controlledTasks</code> , symbolic encoding of the desired property of the system, where the system, plant, is the parallel composition of two instantiations of the node (task) <code>t</code> of List. 3.1.	44
4.1	Verilog module <code>m</code> , faking computations on data given on input wires <code>i</code> upon request <code>r</code> . Output wires <code>o</code> carry its result, available when <code>e</code> becomes 1. . . .	52
4.2	Verilog module <code>main</code> , instantiating <code>m</code> twice.	54
4.3	Heptagon node encoding the machine of Figure 4.1.	55
4.4	Heptagon node SM_m , obtained from the Verilog module <code>m</code> of List. 4.1 using marked variables $S_m = \{\text{state}, r, e\}$	60
4.5	Excerpts of the CGL-enabled Verilog module <code>m'</code> obtained from the module <code>m</code> of List. 4.1.	60
4.6	Heptagon node CM_{main} obtained from the Verilog module <code>main</code> of List. 4.2 using marked variables $S_{\text{main}} = \{\text{cfg}, \text{wait}_m1, \text{wait}_m2\}$ and S_m as used in List. 4.4.	62
4.7	Excerpts of resulting Verilog module obtained from module <code>main</code> of List. 4.2.	63
5.1	A simple process in Verilog.	77
6.1	Verilog module for simulating a process’s behaviors.	97
6.2	Verilog module for FIFO’s simulated behaviors	97
6.3	Verilog module for a managed design with 5 processes; the <code>manager</code> module is the result of the control algorithms (<i>cf.</i> Figure 6.4).	99

6.4 Excerpts of stochastic simulation bench in Verilog for a design with 5 processes; this particular bench simulates the arrival of a new piece of data once every 100 clock cycles (cc), and reports every 1000 clock cycles the total energy consumed by the simulated design per piece of data completely processed. The quantitative values (power consumption and execution times of each process configuration) are drawn within 70% and 100% of their respective specifications. The code that stops the simulation is not shown. 100

List of Tables

4.1	Estimated mean power dissipation (in mW) of original and resulting RS decoders.	64
5.1	Evaluation results for the original and power-aware designs; the “Cycles” column denotes the total number of clock cycles required for processing the considered test-bench (Device: Cyclone IV, Frequency: 100Mhz).	83
5.2	Performance of the strategy computation tool for N RLE instances.	84
6.1	Specification values for our example design with 6 processes; the Worst-case Execution Times are given in number of clock cycles.	101
6.2	Synthesis time and memory footprint of ReaX for designs involving either 5 or 6 processes (N), <i>w.r.t.</i> size of sliding window selected for the limited optimal control algorithm (k); we also report the size of the resulting manager circuit for the design with 6 processes.	102

Chapter 1

Introduction

The influence of embedded systems on our life has been rapidly increasing for several decades, and it seems that they are becoming more and more important every day [4, 38, 95]. They are used almost in every area of life and try to satisfy different kinds of demands ranging from simple to very sophisticated systems. Although each system is constructed in accordance with different kinds of requirement, some design goals of these systems are common, such as low power and cost, high performance, and small size. However, the system construction (*i.e.*, the combination of hardware and software design) with such performance properties comes with rather contradicting requirements. As an example, on the one hand, consumers are expecting high performance from them. On the other hand, they should still be energy efficient. To meet such performance goals and overcome the design constraints should require efficient analysis and management techniques [105]. The objective of the research proposed in this thesis is to achieve power efficiency in hardware circuits with symbolic control.

The hardware circuit design has evolved rapidly over the last four decades [44, 45, 81]. The greatest impact triggering the development of the hardware circuits design is the decreasing of the transistor size, where transistors are the basic building elements in hardware circuit design. As the technology advanced and the transistor size has decreased, designers were able to place a higher amount of transistors on a circuit, and it seems that the trend of the increase in the transistor amount will continue; however, the design of the huge number of transistors on a circuit is becoming more sophisticated. The design complexity of circuits has been dealt with using abstraction methodology, which advocates that design consists of a hierarchical structure, where larger blocks (higher levels) include abstracted

smaller blocks (lower levels). A resulting hardware design consists of the combination of these abstracted designs, and each design is considered at each abstraction level and/or in a structure that is a mixture of some abstraction levels (an abstracted hardware design generally includes this kind of a mixed structure today). Therefore, the optimization of desired performance properties in a resulting hardware design has become a challenging task that requires to be solved for each design block. Among these abstraction levels, the Register Transfer-Level (RTL) is the most effective level for the optimization of hardware designs, in terms of the trade-off between efficiency and complexity. In the literature, several kinds of researches have been conducted to meet various performance goals at this abstraction level. At the register-transfer level, especially the power/energy efficiency is one of the essential performance goals that allows making significant optimizations on a circuit, where the power is one of the major design constraints in today's embedded systems, limiting performance, battery life, and reliability [9, 58, 94]. Various mechanisms can be used to reduce power consumption in hardware chips at the register transfer level, which includes clock-gating, multi-supply voltage, and power-gating. Achieving power/energy efficiency in hardware circuits by applying RTL clock gating technique is one of the most emerged topics in the literature, and the research proposed in this thesis addresses this issue as well. In synchronous circuits in particular, RTL clock-gating, which promises maximum power saving among the others, is used to selectively cut off the clock of components with the aim of reducing the power dissipation induced by the switching activity it incurs; however, the fact remains that efficient analysis and management are required to decide when to switch-off clock signals.

The increasing complexity of contemporary embedded systems complicates the analysis and management of hardware designs, where the design consists of some piece of code [4, 105]. To manage the increasing design complexity and power consumption of hardware circuits as well as to optimize various performance properties, modeling formalisms are becoming common practice. The models abstract away the irrelevant details, so that they allow designers to focus only on the essential properties of the system under design. For this reason, hardware designers often use modeling formalisms, such as automata, synchronous data flow, Kahn-Process-Networks (KPNs), and data-flow synchronous languages based models. Among these models, the main advantage of data-flow synchronous languages is that they do not include the well-known state explosion problem, and allow for building the model by effectively using conditional expressions. The use of such models (*i.e.*, abstracted symbolic models, which are built in the form of data-flow synchronous languages aspect

from hardware designs) significantly reduces the effort of analysis and management, to meet performance goals in hardware designs. Thus, this effective structure (*i.e.*, symbolic modeling) is utilized in order to analyze the given designs in the research proposed in this thesis.

Various self-management approaches, such as control, model checking, heuristics, and machine learning techniques, can be used to analyze hardware design models and apply an optimization technique [4]. Compared to other existing techniques for achieving various performance goals, the main advantage of the Discrete Controller Synthesis (DCS) technique is that it provides a synthesisable controller with formal correctness. Thus, the research proposed in this thesis advocates safe design methodologies based on the DCS technique, in order to achieve power efficiency in hardware circuits. Towards this goal, the objective of the research is to develop a systematic framework that includes: deriving an abstract symbolic model from hardware circuit designs and some kinds of control objectives; and then synthesizing a controller that ensures the given property (*i.e.*, inhibiting clock signals when required without restricting the behavior of the design) from the abstract symbolic model by means of DCS technique (*i.e.*, symbolic control) in order to achieve power efficiency in hardware circuits; and then integrating the controller into the original designs. The framework can also be automatically employed to derive power-efficient versions from original circuit designs. The proposed approach can assist designers in tackling average and/or instantaneous power consumption after designed their behavioral models.

1.1 Problem Statement

The trends outlined above show that with increasing complexity in hardware circuits, power is one of the major design constraints in today hardware circuits, and RTL clock gating, which is one of best low power techniques, has become one of the most effective solutions to achieve power efficiency and also to optimize various performance properties in hardware circuits. Furthermore, the trends outlined above also show that modeling and analysis are often required to reason on hardware circuit designs in order to meet various performance goals with formal correctness by using a controller (*i.e.*, discrete controller synthesis technique), and they are becoming the main solution of today for such systems; where the family of data-flow synchronous languages seems to be one of the best solutions, as it significantly reduces the effort of analysis and management. Given these realities, we observe that RTL Clock-Gating-Logic (CGL) computation (the principle is illustrated in

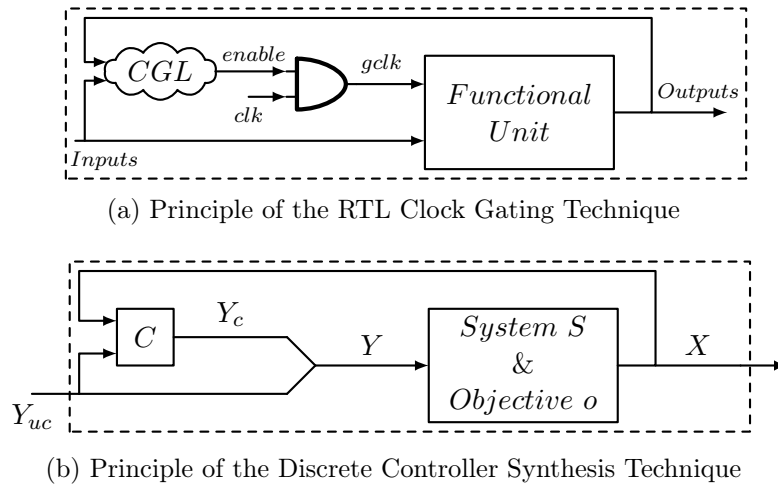


Figure 1.1: Observation of the Feedback Control Problem

Figure 1.1a—for details see Section 2.4.1) is actually a Discrete-Controller-Synthesis (DCS) problem (the principle is illustrated in Figure 1.1b—for details see Section 3.2), where the similarity of the schematic diagrams between them is illustrated in Figure 1.1 (*i.e.*, the observation of the feedback control problem). With respect to this, the main research question of this thesis is:

How to achieve power efficiency in hardware circuits with symbolic discrete control?

The objective of the research proposed in this thesis is to derive power-efficient versions of original hardware designs at Register-transfer Level with symbolic discrete control. In this direction, the research covers the following challenges.

- **Modeling Formalism:** aims to construct abstract models (*i.e.*, symbolic data-flow synchronous language models) from original hardware designs at the RTL, in order to analyze them efficiently by focusing only on the essential properties of the system under design.
- **Design Objective:** aims to define a strategy that controls clock signals in hardware circuits to achieve power efficiency.
- **Self Management Strategy:** aims to apply a technique that provides a synthesisable controller with formal correctness (*i.e.*, DCS technique).

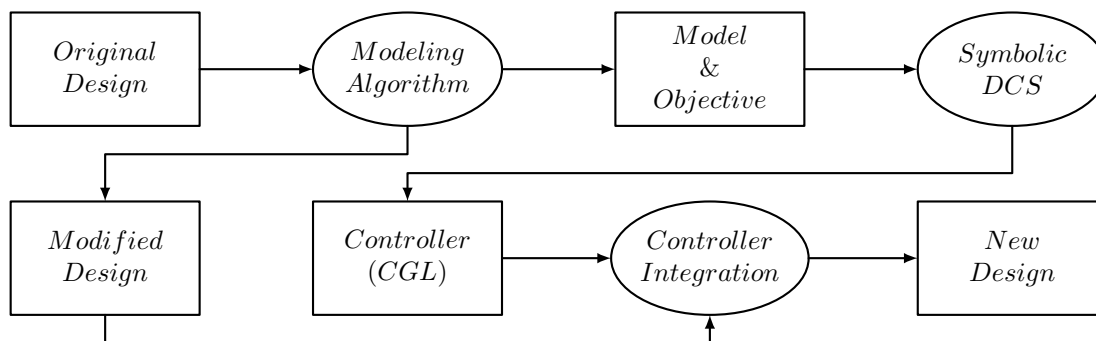


Figure 1.2: Design Flow of the Research Methodology Proposed in this Thesis

- **Code Generation & Integration:** aims to generate a new hardware design from the original design by adding necessary components and integrating the resulting controllers.
- **Validation:** aims to guarantee that the behavior of a resulting design is strictly equivalent to the original, and to ensure that it consumes less power.

To the solution of the research question covering the challenges above, we encode the RTL clock gating computation as several symbolic discrete synthesis problems by using the data-flow synchronous language aspect, and use the resulting controllers (*i.e.*, CGL) to derive power-efficient versions from original circuit designs. The design flow of the research methodology proposed in this thesis is illustrated in Figure 1.2.

1.2 Contributions

In this research, we identified three hardware design problems, where each consists of some design objectives, in order to achieve power efficiency of synchronous circuits. We consider that synchronous circuits are implemented as KPNs, and hierarchical compositions of sub-circuits, described using the popular Hardware-Description-Language (HDL) Verilog at RTL. We propose a tool-supported framework, which addresses the aforementioned challenges, for each hardware design problem. The main contributions of this thesis are introduced below.

In Chapter 4, we describe a systematic approach that computes the CGL of synchronous circuits in order to switch off the clock of each sub-circuit (to save power) when they are in an idle status. More specifically, we encode the computation as several small symbolic

discrete controller synthesis problems (*i.e.*, an abstract symbolic synchronous model by using Heptagon language for each individual Verilog modules) by means of the BZR tool, and use the resulting controllers (where controllable variables represent output wires of CGLs) to derive power efficient versions from original circuit designs. We demonstrate the principles using an example, report on its manual application on a realistic case study, and validate our approach experimentally.

In Chapter 5, we advance our framework introduced in Chapter 4. Our approach relies on the automatic construction of abstract symbolic models of the design, as well as associated control objectives (one of them supports the solution of the hardware design problem introduced in Chapter 4), and employs discrete control techniques to compute a piece of hardware circuit that implements some power-aware scheduling policies specified in a declarative way. This piece of circuit can eventually be used to selectively filter the clocks of the processes involved. The resulted designs are automatically produced by means of our tool `dcs4cgl`, where the DCS technique is performed by the tool `ReaX`, and the abstract models are encoded on the `ReaX` environment as well. We illustrate and validate our approach and the strategies it provides, experimentally using various RTL designs described using Verilog.

In Chapter 6, we re-specify the objectives placed in Chapter 5 in order to employ an energy-efficient configuration manager for choosing the optimal configuration, by means of the clock-gating logic, among the alternatives on data-flow hardware circuits. Our technique permits the systematic construction of abstract symbolic models of such designs, as well as associated control objectives, so it is easily implementable. Then, we employ the DCS technique along with the associated control objectives on the models by means of the `ReaX` tool, in order to produce a piece of hardware circuits that implements a configuration manager with energy reduction guarantees, where the manager behaves as a clock-gating constraint. We apply and validate our approach on a series of various design flows built from RTL implementations (described using the HDL Verilog), where we use artificial processing modules and memory components.

1.3 Thesis Outline

The main goal of this thesis is to achieve power efficiency in hardware circuits with symbolic discrete control. This thesis is divided into seven chapters and the organization of the thesis is as follows.

First three chapters include an introduction, some technical background information, and the literature review. Chapter 1 makes an introduction, identifies the problem statement, introduces our contributions, and gives the outline of this thesis. Chapter 2 provides the necessary overview of the current state-of-the-art in the context of integrated circuits, their design methodologies and existing low-power design techniques, and some CAD tools. Chapter 3 presents the existing modeling formalisms on hardware circuits designs emerged in the literature and also provides the necessary overview of the DCS technique and its implementations.

The next three chapters exhibit our contributions, to achieve power efficiency in hardware circuits with symbolic discrete control. In Chapter 4, we present a framework for the construction of energy-efficient synchronous circuits with symbolic discrete control based on the idleness conditions. In Chapter 5, we advance a framework for the construction of power-aware hardware circuits with symbolic discrete control based on the scheduling constraints and power-efficiency objectives. In Chapter 6, we describe a framework for the construction of an energy-aware configuration manager with symbolic discrete control, based on the energy optimization objective and configuration constraints by supporting with parallel synchronous processes.

Finally, Chapter 7 provides concluding remarks on the work presented and summarizes the main conclusions and future recommendations of this thesis.

Some material used in this thesis has already been published/reviewed/submitted, where the contribution chapters introduce the identification tags.

Chapter 2

State of The Art on Low Power Synchronous Circuits Design

Nowadays, the power efficiency of integrated circuits is one of the essential factors in modern embedded systems, and various techniques are used to reduce the power consumption in hardware chips. However, the clock gating technique, which is implemented only on synchronous integrated circuits, is one of the most common techniques used for the energy reduction because the clock power consumes the majority of the total chip power and synchronous-circuits/clock-signals are used in almost all embedded systems. Thus, low power synchronous integrated circuits design is of significant importance for constructing embedded devices, as the circuits have a wide range of usage area [9, 43, 55, 70, 94, 97, 99].

This chapter gives some background information about chip design and presents existing low power synchronous integrated circuits design techniques. This chapter is organized as follows: Section 2.1 gives a brief overview of integrated circuits; Section 2.2 presents the chip design process, and Section 2.3 introduces how chips are described during this process; Section 2.4 starts with the background information of the clock gating technique, and it continues with existing low power synchronous integrated circuits design techniques; Section 2.5 presents some existing design tools used during the chip design process; finally, Section 2.6 concludes with the summary and discussion.

2.1 Integrated Circuits (ICs)

This section gives some background information that includes definitions, classifications, architectures, and application areas about *Integrated-Circuits (ICs)*.

Integrated circuits (monolithic integrated circuits) are a group of electronic circuits, where the whole circuit (up to several billion transistors and other electronic components) is designed in a single piece of semiconductor material [37].

They are used in almost all electronic devices (*e.g.*, computers, mobile phones, and any digital devices used in our daily life). The wide range of usage area requires that they are designed in different types. Furthermore, various architectures have developed for meeting desired features (*e.g.*, low power, high performance, and flexibility). These types and architectures are introduced below.

2.1.1 Classification of ICs

The structure of ICs shows the material and technology used during the construction or fabrication process. ICs can be constructed based on the structure as *monolithic*, *thin/thick film*, and *hybrid*; however, in common usage, *Integrated-Circuit (IC)/chip* is referred to as a *monolithic integrated circuit* [101]. In the thesis, the terms *Integrated-Circuit (IC)/chip* are used in this way.

A well-known phrase is that the number of transistors on ICs is almost doubled every year with increasing technology [78]. Indeed, the number of transistors on a chip is over a billion now. According to the amount, new technical specifications occur and classifications are reshaping. As an example, *Large-Scale-Integrations (LSIs)* have up to twenty thousands transistors. Then, as the number of transistors increased, the new definitions took place in terminology as *Very-Large-Scale-Integration (VLSI)* and *Ultra-Large-Scale-Integration (ULSI)*.

The classification of ICs can be done in various forms (*e.g.*, scale, package type, and transistor type); however, they have some main characteristics that are introduced below.

Classification of ICs by Function

The function indicates the continuity of amplitude values (*analogue*, *digital*, and *mixed*) by referring to signals of a chip [101]. Signals have a continuous range of amplitude values in *analogue ICs* (*e.g.*, sensors, power management circuits, and operational amplifiers). On the

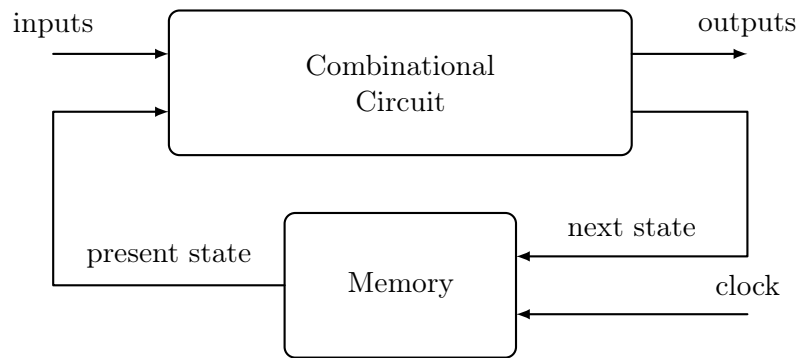


Figure 2.1: Structure of a Synchronous Circuit

contrary, *digital ICs* (e.g., microprocessors, digital signal processors, and micro-controllers) process by using a finite set of discrete values of the signals.

Classification of Digital ICs

Digital ICs can be classified as *combinational* and *sequential* [52]. *Combinational ICs* (e.g., adder, code converter, decoder, and multiplexer) consist of logic gates, where the outputs at any instant depend on the current inputs at that instant only. *Sequential ICs* (e.g., state machine, counter, and shift register) also consist of logic gates; however, the outputs at any instant depend not only on the current inputs but also on previous inputs. Therefore, *combinational ICs* do not have any memory, whereas *sequential ICs* have memory (e.g., register, flip-flop, latch), where previous values are stored in.

Classification of Sequential ICs

Sequential ICs can also be divided into *synchronous* and *asynchronous* within itself based on trigger signals of the circuit [41]. *Synchronous ICs* are triggered with a global clock signal to control the operation of the circuits, and the generic structure of a *synchronous IC* is illustrated in Figure 2.1. On the other hand, *asynchronous ICs* do not have a clock signal; instead, they use signals that show the completion of tasks to control the operation of the circuits.

2.1.2 Architectures of ICs

Chips are used almost in every area of life and they try to satisfy different kinds of demands. However, modern embedded systems have some design constraints [38]. On the one hand, designers are expecting high performance from them. On the other hand, they should still be energy efficient and flexible to be used for some kinds of tasks. Thus, new architectures are developed in order to overcome these constraints. Architectural development is not restricted just for a single component (*e.g.*, processor). It also includes more complex design, such as that some components piece together in a single chip in order to get high performance.

The main architectural concepts are flowing as *processor*, *memory*, *interconnect*, and *system-on-chip*.

Processor

Processors are a group of electronic circuits, where the whole circuit performs operational tasks by using data from the environment. A *processor* can be an IC itself or a part of it.

Processors are designed for intended usage (*i.e.*, to always perform the same task or for alterable tasks). There are several architectures available for different usage areas. According to [95], these architectures can be classified into two main domains: general-purpose computing and application-specific computing.

The general-purpose computing domain supports different kinds of tasks. These the *processors* are programmable and they are called *General-Purpose-Processors (GPPs)*. On the other hand, *processors* that focus only on a specific task in the application-specific computing domain are called *Application-Specific-Integrated-Circuits (ASICs)*. *ASICs* have better performance than *GPPs*, whereas *GPPs* offer higher flexibility.

Apart from these domains, there is also a domain that provides both high performance and flexibility, and it is called reconfigurable computing [95]. These *processors*, *e.g.*, *Field-Programmable-Gate-Arrays (FPGAs)*, can be described many times in order to process any desired task (*i.e.*, as if it is redesigned in a factory each time).

Memory

A *memory* is a component on a chip for data storage, and *memory* architectures have an enormous impact in order to overcome design constraints, so they should allow quick access to data. Basically, *memory* architectures can be divided into *distributed* and *shared memory*

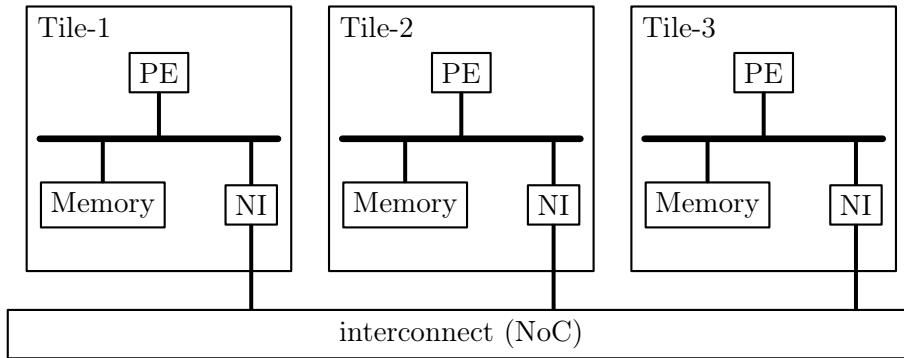


Figure 2.2: Tile Based MPSoC Block Diagram

systems based on data accessibility [27, 104]. In a *shared memory* system, all processors communicate with a single (global) storage. On the other hand, each processor has a local storage in a *distributed memory* system; however, they can still communicate with each other via interconnect.

Interconnect

Interconnect provides connections between components on a chip for carrying data. *Interconnect* can be provided using traditional *bus* techniques, which have a restricted bandwidth, or *interconnect* can be provided with a *Network-on-Chip (NoC)*, which is a system, composed of some components *e.g.*, Network-Interface (NI), that only deals with communication [4, 104].

System on Chip (SoC)

A *System-on-Chip (SoC)* is a complex IC that involves all necessary components (*e.g.*, processor, memory, and interconnect) on a single chip [57]. A circuit that includes more than one processor can also be called *Multi-Processor-System-on-Chip (MPSoC)*. The types of processors can be the same or different, and the respective *MPSoCs* are called homogeneous and heterogeneous, respectively.

Figure 2.2 shows a tile based *MPSoC* block diagram, where each tile is a largely independent sub-system in a *MPSoC* and is generally composed of Processing-Element (PE), memory, and interconnect. In the diagram, the NIs, the sub-components of the NoC, provide communication both locally (between a processor and memory, in each tile) and globally (between the tiles).

2.2 Chip Design Process

The previous section gives some information about the main types and architectural concepts of ICs, and this section presents their *design process* that consists of some stages/levels/layers. Furthermore, this section introduces *chip design methodologies* and *the evolution of chip design*.

2.2.1 Evolution of Chip Design

Chip design has evolved rapidly over the last four decades [44, 45, 81]. The first ICs called Small-Scale-Integrations (SSIs) involved a very small number of transistors, and their construction was easy because their design models consisted of a very small number of connections between these very small number of transistors. These the construction, made with transistors, is called transistor-level (the lowest level) design; the main concern of low-level design was the physical layout plan (*e.g.*, placement and routing).

As the technology advanced and the transistor size decreased, designers were able to place a higher amount of transistors on a circuit called Medium-Scale-Integration (MSI) or Large-Scale-Integration (LSI); however, design became more complex, and the layout plan became non-manageable. In order to overcome design complexity, designers began to use the abstraction methodology, which advocates that design consists of a hierarchical structure that larger blocks (higher levels) include abstracted smaller blocks (lower levels). The aim of the abstraction methodology is always the next higher level of abstraction. In this case, the respective levels were transistor and gate/logic, from low-level to high-level respectively. Thus, designers started to make a design at the new level called Register-Transfer-Level (RTL).

In order to support the abstraction methodology: first, designers created libraries of lower level components in order to use them on the next high-level; then, some design tools called Computer-Aided-Design (CAD) tools were developed to make a design with higher level components, and to decompose these components towards lower level components, and to simulate the design by using lower level components. This decomposition, called hardware synthesis, supports to generate lower level descriptions of a design automatically from the higher level descriptions, so the design can be constructed with fewer components and less error-prone. Thus, CAD tools (*e.g.*, hardware synthesis) have managed low-level problems, such as the layout plan, because the models of low-level design are generated automatically (*i.e.*, CAD tools replaced human designers). As a result, the advent of logic

synthesis, where logic synthesis is a special name of hardware synthesis at the relevant level, has reduced design time dramatically and has made a design to be more reliable.

When chips, *e.g.*, Very-Large-Scale-Integration (VLSI) and Ultra-Large-Scale-Integration (ULSI), have continued to contain a higher amount of transistors, researchers and industry began to focus higher levels of abstraction, and they abstracted the Register-Transfer-Level (RTL) and algorithmic-level, respectively. The aim of all levels of abstraction is to use fewer components in order to make a manageable, rapid, and reliable design; details of abstraction levels are introduced in the next section. Now, the system-level is the highest level of abstraction and promises to reduce problems between hardware and software by supporting to develop hardware design simultaneously with software design.

As a result that higher levels of abstraction let design to be automatic, to complete quickly, to be more reliable and less error-prone, by getting help from CAD tools. Although CAD tools are available to automate the design process, designers are still responsible to control how the tools are working at each level. Today, design can be done at any level that is mentioned above; however, the design process generally continues fully automated after the RTL. In addition, although the target of the abstraction methodology is a higher level of abstraction, the most popular implementation area is the RTL, and research is still continued for all levels of abstraction.

2.2.2 Abstraction Levels

As the number of transistors on a chip increased, chip design became more complex [44, 45, 81]. Thus, designers began to use the *abstraction methodology* in order to overcome design complexity. The *abstraction methodology* advocates that design consists of a hierarchical structure that larger blocks include smaller blocks, and both the bottom-up (*i.e.*, larger blocks that are constructed by using smaller blocks) and top-down (*i.e.*, smaller blocks that are decomposed from larger blocks) approaches can be used in order to make a chip design by using these blocks. The aim of the methodology is an *abstraction* between larger and smaller blocks/components, so designers focus just their own design level without considering design details of other abstraction levels. Thus, a chip design can be constructed more efficiently.

Various *abstraction levels* are shown in Figure 2.3, which is called *Y-chart*, and they are identified by concentric circles. The *Y-chart* was developed in order to explain differences between different *abstraction levels* and claims that any design can be modeled in three ways without considering whether it is complex [44, 45, 46]. These three models, *behavioral*,

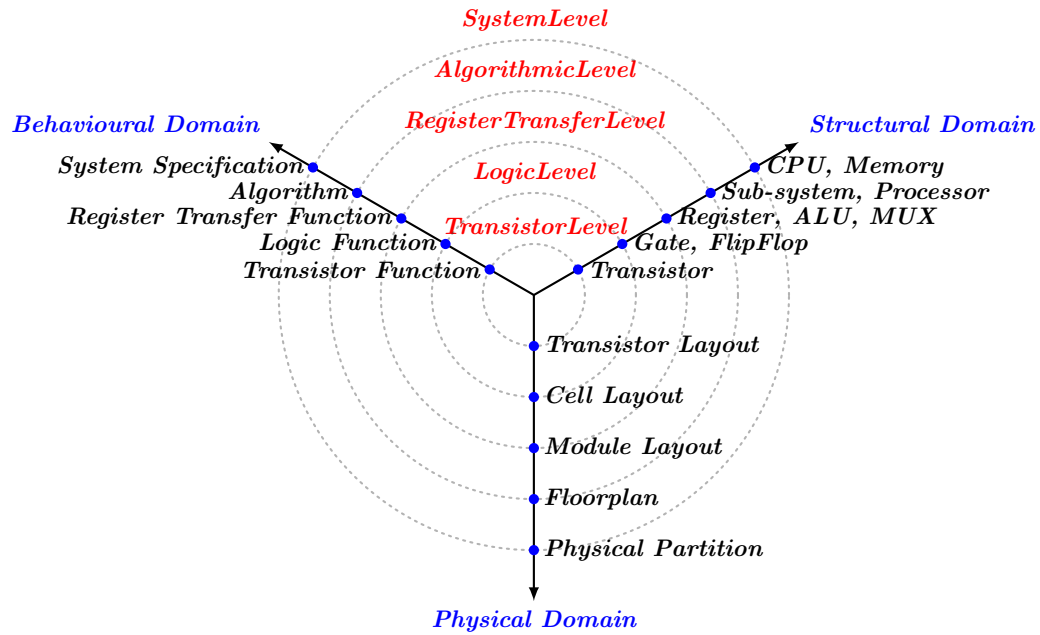


Figure 2.3: Y-Chart Design Model

structural, and *physical*, are represented in the *Y-chart* by three axis.

- **Behavioral Model:** Design specifications and functionalities are described. Models at different levels include transistor function, logic function, register transfer function, algorithm, and system specification.
- **Structural Model:** Block diagrams and netlists are designed generally by using a behavioral model. Models at different levels include Central-Processing-Unit (CPU), memory, processor, Arithmetic-Logic-Unit (ALU), register, Multiplexer (MUX), gate, and transistor.
- **Physical Model:** Layout and board design are arranged generally by using a structural model. Models at different levels include transistor layout, cell layout, module layout, floor plan, and physical partition.

The *Y-chart* allows that design can be easily manageable at all levels because these three models show different characteristics of the same design. These three representations can be modeled separately from each other with their concepts along *abstraction levels*. However,

almost all designers use the top-down methodology in the literature, so the generic design flow follows that: first, designers implement the *behavioral model* of design at the desired level; latter, the *structural model* (which represents block diagrams of the components, where the components consist of next low-level components) and *physical model* (which adds geometric dimension to the structural model) are automatically generated from the *behavioral model* by the help of CAD tools (*i.e.*, hardware synthesis); then, the design continues with the next low-level *behavioral model* (which is automatically obtained from the structural model); and then, the design follows the design flow until reaching to the lowest level *physical model* [4, 44, 45]. The *abstraction levels* available in the *Y-chart* are introduced below.

Transistor Level

The *transistor-level*, where a design is constructed by using transistors, resistors, capacitors, and etc. or contains them, is the lowest level of abstraction [25, 44, 45, 102].

The structural models, which represent connections between *transistor-level* components, are generally synthesized automatically from the behavioral models, which describe functionalities between transistors by using Hardware-Description-Languages (HDLs) 2.3. Then, the physical models, which show the placement of transistors on board, are created from the structural models by using CAD tools (*e.g.*, placement and routing).

However, the manual implementations are very difficult while considering huge design and complexity. The behavioral descriptions are generally obtained automatically from the logic-level (next higher level). Therefore, logic-level components (*e.g.*, gate and flipflop), which are stored in *transistor-level* libraries, are constructed by using *transistor-level* components, at this level in order to use them in the logic-level.

Once a physical model is produced, the fabrication process starts with the transistor layout description. In addition, there are also some stages (*e.g.*, device-level and technology-level) which consider the transistor parameters such as size and temperature, at the fabrication process.

Logic Level

The *logic-level* is an abstraction level where the building blocks are standard cells (*e.g.*, gate and flipflop) in a design [25, 44, 45, 102].

The implementations can be modeled in a similar way as in the transistor-level by using

standard cells instead of transistors. However, the manual implementations take a very long time like as in the transistor-level, because there are too much design details. The behavioral descriptions are generally synthesized automatically from RTL (next higher level) components (*e.g.*, register, alu, and mux), which are constructed by using *logic-level* components and are stored in *logic-level* libraries. Thus, designers generally prefer to do optimization at the RTL by using simulation/feedback results of *gate-level* netlists (where the netlists are automatically generated from a behavioral RTL code), instead of doing *logic-level* implementations. Once the optimization of a design is completed, the design continues at the transistor-level (next lower level) by using the *logic-level* structural model, which can be transformed into a transistor-level behavioral model.

Register Transfer Level (RTL)

The *Register-Transfer-Level (RTL)* is a micro-architectural level, where a design is constructed as a hierarchical structure by using functional blocks called modules, which are built by using the components (register, alu, and mux) available in logic-level (next lower level) libraries [25, 44, 45, 81, 102].

The most popular implementation area on chip design is the *RTL*, so the studies are intensified in this level. Thus, various techniques are developed for design optimization (*e.g.*, high performance, low power, and less area usage), where the optimization is on the behavioral models, which are some piece of code written by using HDLs 2.3.

The structural models show block diagrams of interconnected *RTL* components, and they are generally synthesized from the behavioral models, which supply the functionality of the interconnected components. Thus, the physical models show the module layout composed of the *RTL* components.

As mentioned above, designers generally implement the behavioral models and synthesize them in order to generate the structural models and low-level models. Then, they apply optimization on the behavioral codes by using simulation results of gate-level netlists, where the results include some design information (*e.g.*, functional correctness). Once a design completed at this level, then the design process continues as a fully automated until produced a hard device.

Algorithmic Level

The *algorithmic-level* is a layer of abstraction where a design consists of functional blocks and/or subsystems (*e.g.*, hardware module, processor, custom hardware component, and intellectual property), which are the components constructed at the RTL (next lower level) [25, 44, 45, 102].

The aim of this level is: to describe the behavior of a design by using algorithms and mathematical expressions, as a set of concurrent programs; and/or to create the libraries that include system-level (next higher level) components (*e.g.*, cpu and memory), by using functional blocks, in order to use them in the system-level.

The structural models, which represent interconnected functional blocks, can be manually implemented by using *algorithmic-level* components or can be automatically synthesized from the behavioral descriptions, which can be specified by using HDLs 2.3. Although the physical models can be obtained from the structural models by using CAD tools (if the RTL library used stores the layout of components), it is not generally preferred at this level. Thus, design continues at the RTL by converting the structural models to RTL behavioral descriptions.

System Level

The *system-level*, where a design is simultaneously constructed with hardware and software, is the highest level of abstraction and describes connections between main/system components/blocks (*e.g.*, cpu, memory, and processor) [44, 45].

The manual implementations can be modeled easily because there are few components used in a design. The behavioral models represent functionalities of components' communication, such as that a model can include pipelining or parallel process between processors. Functionalities can be described by using HDLs 2.3 as a behavioral code that consists of a Finite-State-Machine (FSM). The structural models, which represent block diagrams of system components, can be obtained manually by using system components or automatically from the behavioral descriptions. Then, if the algorithmic-level (next lower level) libraries used in design support to access layout models of components, the physical models can be generated from the structural models. However, the generation of the physical models is not generally preferred at this level by designers, so design continues at the algorithmic-level by using the structural models, which involve algorithmic-level components.

2.2.3 Chip Design Methodologies

As the complexity of chip design increased, designers started to make a design at higher levels of abstraction. In order to make that, they created higher level components by using lower level components, in order to use them at higher levels. Thus, the components used at higher levels are available in libraries by storing design information (*e.g.*, behavioral, structural, and physical models) of lower level components. However, it is not required that all design information are always available in libraries for a chip construction. As an example, a higher level component can be constructed as a whole component that holds all design parameters/information, or it can be constructed with some missing parameters/information (*e.g.*, physical model), which will be included to design at further design steps. Thus, final design can be obtained by using various methodologies. Which methodology is used is generally depending on manufacturers and products. However, design methodologies can be broadly divided into three approaches as *bottom-up*, *top-down*, and *meet-in-the-middle* [25, 44, 45, 102].

The *bottom-up* approach is the methodology that higher level components are constructed by using lower level components with their all design parameters/information. Design constructions start at the lowest level (transistor-level), and once design gets the desired level, constructions are completed, because all design information (*i.e.*, physical model) are available for the fabrication process.

In the *top-down* approach, a design is constructed in the opposite direction of the bottom-up approach. The construction starts at the desired level, and the components used in the design are decomposed towards to lowest level components. Higher level components include only behavioral and structural models of lower-level components, so there are some missing parameters/information (*e.g.*, the physical model of lower level components) in higher level design. Once the structural model of the transistor-level (lowest level) is obtained, the physical model is generated from the structural model, and the design is completed.

A design has a long process at the *bottom-up* approach because all design parameters/information of all abstraction levels must be available in libraries. Thus, any changes in lower levels may affect the whole design. However, designers have full knowledge of all design at any time. Although the *top-down* approach offers a very quick design, designers can not have full knowledge of a design before the design is completed. Thus, the approach does not allow optimization about design metrics unless a design is completed.

The *meet-in-the-middle* approach is located between the bottom-up and top-down

approaches, in terms of advantages and disadvantages. A design construction starts together at the desired level and the lowest level. A design coming from the lowest level with all design details, and a design coming from the highest level with missing design information, come together in the middle. Thus, designers have full knowledge about the design at a level which is located in the middle.

In the literature, almost all designers use the *top-down* approach, because it offers a shorter time-to-market. However, one trend in embedded systems is that the size of products is decreasing, so it pushes manufacturers to use the *meet-in-the-middle* approach in order to have knowledge about the physical model of design at higher levels. As a result, manufacturers determine the methodologies used in a design by depending on products.

2.3 Hardware Description Languages (HDLs)

In the previous section: the chip design process and abstraction levels are introduced by using the Y-chart, which claims that a design can be modeled in three ways; and it is also mentioned that designers generally use the generic design flow, where the generation of a behavioral model is enough for the chip construction, in design. In this section, *Hardware-Description-Languages (HDLs)*, which are used for the implementation of a behavioral model, are introduced and are detailed through an example language, *Verilog*.

Figure 2.4 shows the generic chip design flow. As it has shown in the flow, hardware synthesis is one of the design tasks during the chip design process in order to automatically generate the low-level description of a design from the high-level behavioral model; and *HDLs* are a language that is used to describe these behavioral models as a set of concurrent programs by using the components available in libraries. However, designers have different requirements to implement a behavioral model for each level of abstraction because the components used in each level of design are also different, as well as the functionality required between components. Thus, there are several *HDLs*, where each supports to make a design at different level/s, and they are sometimes categorized based on the level of abstraction [93]. Some common languages, *SystemC*, *VHDL*, and *Verilog*, are briefly introduced below, and then, *Verilog*, the most popular *HDL*, is detailed.

SystemC is a C++ class library that creates a model of hardware architecture at high-levels (*e.g.*, system-level) [21, 48]. It was developed in order to make all system design tasks together, such as software and hardware synthesis, because before the invention of *SystemC*, designers were creating the high-level model of a design by using high-level

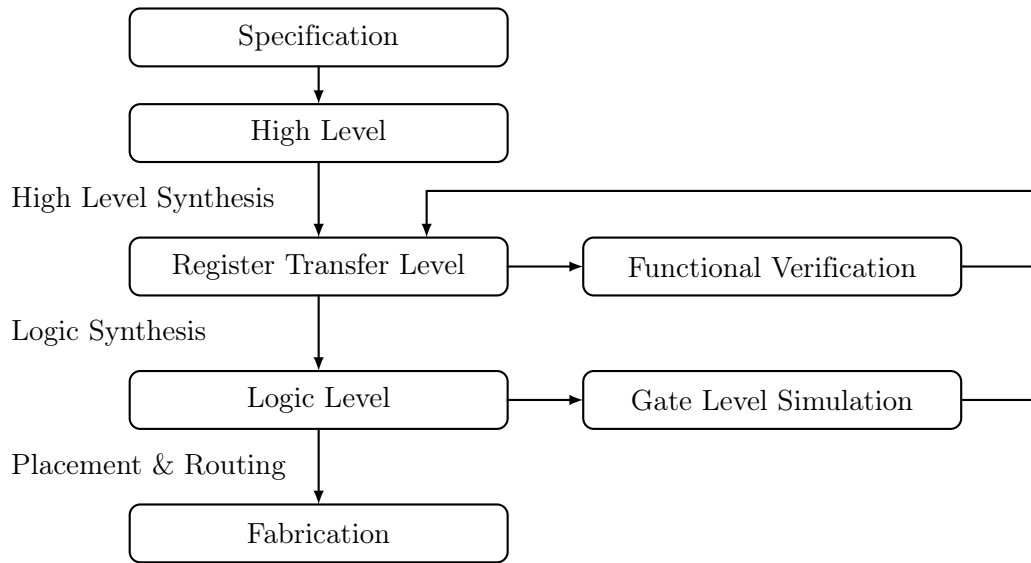


Figure 2.4: Generic Chip Design Flow

languages (*e.g.*, C/C++); and lower level (*e.g.*, generally RTL) designers were continuing the design with the lower level model that is transformed from the high-level model, which could cause incompatibility between high-level and low-level designs; and then, the software was being built on top of the hardware design, which could also cause incompatibility between the hardware and software design. Thus, *SystemC* has managed these problems by simulating and synthesizing all design tasks together. Furthermore, *SystemC* has provided compatibility between high-level and low-level designs because it supports not only features of the object-oriented approach but also features of the hardware-oriented approach, such as low-level timed, cycle-accurate, and component models. By the way, the behavioral models and/or hardware components are constructed by using modules, which are the basic building blocks in a design, as a set of concurrent programs as in hardware-oriented languages; however, modules, which contain input/output ports, are a functional block constructed by using standard C language concepts.

VHDL, which is influenced from Pascal and Ada languages, is a HDL that can be used from the logic-level to the algorithmic-level; however, it is generally preferred at the RTL and also supports to use different levels of abstraction to be mixed in the same model [86, 93]. The models are implemented in a design without depending on technology, so a hard chip can be constructed in a factory by using only the model, without requiring any extra

```

1 module m (input clk, input start, input [7:0] data_in, output [7:0] data_out, output reg done);
  // Constants Declarations
  parameter initialization=0; parameter calculation=1;
  // Data-Types Declarations
5  reg state; reg [7:0] memory;
  // Data-Flow Assignments
  assign data_out=memory;
  // Initial Statement
  initial begin state=initialization; memory=0; end
10 // Always Statement
  always @(posedge clk) begin
    case (state)
      initialization: begin done<=0; if (start) begin state<=calculation; end end
      calculation:  begin memory<=data_in; done<=1; state<=initialization; end
15  endcase
  end
endmodule

```

List. 2.1: Verilog module *m*, faking computations on data given on input wires “*data_in*” upon request “*start*”. Output wires “*data_out*” carry its result, available when “*done*” becomes “1”.

manual implementation. The design concept is to create functional blocks called entities or components as a set of concurrent programs in a hierarchical structure, where a block has input/output ports, hardware data-types, and memory; and functionalities can be easily supplied by composing Finite-State-Machines (FSMs) and/or using statements of C-like language concepts such as “if”, “case”, and “for”.

Before the invention of HDLs, hardware descriptions were being made by using programming languages, such as Fortran, Pascal, and C; however, designers were needing a standard language to describe digital circuits because programming languages were not being satisfied designers’ requirements, such as the concurrency [81]. Thus, HDLs were developed, and *Verilog*, one of the first HDLs, is the most commonly used HDL now. Furthermore, *Verilog* offered tremendous improvements, such as to support logic synthesis and schematic diagrams, for designers; because until the invention of *Verilog*, HDLs had been used only for a logic verification/simulation and had not been supported to show the schematic diagram of a behavioral model. Thus, *Verilog* is both a behavioral and structural language. Also, *Verilog* supports to make a design at from the transistor-level to the algorithmic-level as a hierarchy of functional blocks/modules, by supporting to use different levels of abstraction to be mixed in the same model; however, in the digital design community, the term RTL is generally used for a *Verilog* description that uses a combination of the algorithmic, RTL, and logic levels. Some main concepts of *Verilog* are given below.

The behavioral model of a design/chip is constructed by using a HDL with its concepts,

```

1 module M (input clk, input start, input [7:0] data_in, output [7:0] data_out, output done);
  // Data-Types Declarations
  wire start_m1, done_m1, start_m2, done_m2;
  wire [7:0] data_in_m1, data_out_m1, data_in_m2, data_out_m2;
5 // Data-Flow Assignments
  assign start_m1=start; assign start_m2=done_m1; assign done=done_m1 & done_m2;
  assign data_in_m1=data_in; assign data_in_m2=data_out_m1; assign data_out=data_out_m2;
  // Module Instantiations
10 m m1 (.clk(clk), .start(start_m1), .data_in(data_in_m1), .data_out(data_out_m1), .done(done_m1));
  m m2 (.clk(clk), .start(start_m2), .data_in(data_in_m2), .data_out(data_out_m2), .done(done_m2));
endmodule

```

List. 2.2: Verilog module M, instantiating m of List. 2.1 twice, as m1 and m2, in a pipelining mode where m1 feeds m2.

where concepts of different HDLs show similarities. As an example, behavioral models are a hierarchy of functional blocks as a set of concurrent programs, where a block has input/output ports, data-types, and memory. Thus, some main concepts of HDLs are introduced with an example of a Verilog behavioral model, where the behavioral model consists of two source files illustrated in List. 2.1 and List. 2.2. The main concepts introduced below are *data-types, functional blocks and ports, instantiations, and representations of different levels of abstraction* [81].

2.3.1 Data Types

Basically, there are two data-types available as *net* and *register* in order to declare a variable in Verilog; and variables can take some types of values, such as boolean, binary, and decimal. A variable of net type takes values continuously and only shows the current value of the assignment, so the variables do not have memory, and they are generally declared by the keyword *wire* as at line 3 in List. 2.2. On the other hand, a variable of register type takes values on discrete time and shows not only the current value but also the previous value, which is stored in itself, so the variables have memory, and they are generally declared by the keyword *reg* as at line 5 in List. 2.1.

The variables declared by only keywords (*e.g.*, “**reg state;**” at line 5 in List. 2.1 and “**wire start_m1;**” at line 3 in List. 2.2) are in the boolean domain (called bit in the chip design community) and only take values of a set of {0,1}, so the variables have one bit length. On the other hand, variables can also be declared as a vector that has multiple bit widths (*e.g.*, “**memory**” and “**data_in_m1**” have 8 bit lengths by declaring as “**reg [7:0] memory;**” at line 5 in List. 2.1 and “**wire [7:0] data_in_m1;**” at line 4 in List. 2.2).

Verilog also supports some facilities, such as array and constant declarations. As an example, constants can be declared by some keywords, generally *parameter*, in order to use names of parameter-type instead of constants (*e.g.*, “`parameter calculation=1;`” at line 3 in List. 2.1).

2.3.2 Functional Blocks and Ports

Functional blocks are a circuit where the outputs are expressed as a function of the inputs, and are the basic building block in design, and are called *modules* in Verilog. List. 2.1 and List. 2.2 represent the modules, `m` and `M`, respectively. Modules are declared by the keyword *module*, a specific name, and a list of input/output ports, as at first lines in List. 2.1 and List. 2.2; and the keyword *endmodule* indicates the end of a module as at the last lines. The body of a module describes how inputs are transformed into outputs, in the form of a set of equations. Furthermore, Verilog supports to use some operators (*e.g.*, mathematical, logical, and bit-wise) and statements (*e.g.*, “if”, “case”, and “for”), in order to supply the functionality of a module.

Ports provide communication with the environment by carrying data values. Input and output ports are declared by the keywords *input* and *output*, as at the first lines, and their data-types are net unless specified as register (*e.g.*, “`output reg done`” at first line in List. 2.1).

2.3.3 Instantiations

Verilog supports a hierarchical design by using modules, where a design has only one root/main/top module. A module is a template that provides the functionality specified; and *instances* are an object created from a template/module, where each object has its own name, inputs, outputs, variables, and etc. A module can use features of another module by creating an object/instance from it, and the process is called *instantiation*, and a module, except a top module, can be available in a design only by instantiating it.

The instance declaration is similar to the module declaration. Instances are declared in a module by a source module name, a specific name, and a list of input/output ports of the source module, where each port also includes a specific name in parentheses. As an example, module `M` of List. 2.2 instantiates module `m` of List. 2.1 two times as `m1` at line 9 and `m2` at line 10.

2.3.4 Statements

In Verilog, the functionality of models is provided with a set of concurrent statements, where the statements can be divided into two categories: *assignment* and *conditional*. Assignment statements provide to update a value on a variable by the signs “=” and “<=”. On the other hand, conditional statements are used to decide whether or not an assignment statement should be executed (*e.g.*, by using C-like language concepts, such as “if”, “case”, and “for”). In addition, conditional statements can include a group of statements, where each member of statements places between the keyword pairs such as *begin* and *end*.

2.3.5 Structural Representation

Verilog supports to make a design at the logic-level by using standard cells, such as “and”, “or”, and “xor”. The models made with standard cells are called *structural model/representation*; however, structural representations are in the behavioral domain of the Y-chart, so the term *structural representation* is a somewhat confusing term. To clarify, Verilog supports to make a design in the structural domain of the Y-chart, by using schematic diagrams, and schematic diagrams can also be exactly specified by using notations, in the behavioral domain of the Y-chart; so the representations are called *structural*. As an example, the relation between three variables of net-type, “x”, “y”, and “z”, are constructed by using logic gate “and” below.

The notation of “`and (z,x,y)`” represents the schematic of  .

2.3.6 Data-Flow Representation

Verilog supports to make a design at the RTL in terms of the data flow between variables (*e.g.*, registers and nets), and this kind of a model is called *data-flow model/representation*. Data-flow representations provide the means of describing combinational circuits by their function. The functionality of the models is provided with a set of concurrent *continues assignments*, so only net-type variables have an assignment in data-flow representations; however, their assignments can include any type of variables (*e.g.*, register and net) by means of logical expressions. Thus, continuous assignments describe the logical values taken by the corresponding variables at any instant by means of logical expressions. The assignments

are declared by the keyword *assign* and corresponding expressions, as at line 6 in List. 2.2. Also, a continuous assignment can be declared implicitly, such as “`wire w=exp;`”.

2.3.7 Behavioral Representation

The highest level supported by Verilog is the algorithmic-level, and a model constructed at this level of abstraction is called *behavioral model/representation*, and the representations are mostly used to describe sequential circuits. The functionality of the models is provided with a set of concurrent *procedural assignments*, where each assignment only appears inside the structured procedure statements, *initial* and *always*.

Procedural assignments only update values of register-type variables, where each update depends on a conditional statement; and a register can have multiple assignments. There are two types of procedural assignments, *blocking* and *nonblocking*. Blocking statements provide a priority that the first written register is updated first, between assignments where the conditions are provided (*e.g.*, the sign “=” provides that “`state`” is updated earlier than “`memory`”, at line 9 in List. 2.1). On the other hand, nonblocking statements provide the concurrency between assignments where the conditions are provided (*e.g.*, the sign “<=” provides that all the registers at line 14 in List. 2.1 are updated concurrently).

Initial statements are a conditional statement, where the condition is whether the time is at the beginning. Initial statements include only procedural assignments and load the initial value of a register, and they are declared by the keyword *initial* as at line 9 in List. 2.1. Always statements are also a conditional statement; however, the conditional statement has a sensitivity list that is used to decide whether or not a statement should be executed. The system is called triggering mechanism, and always statements include only procedural assignments. A sensitivity list consists of variable/s available in a model, and the triggering mechanism is always listening to the sensitivity list in order to trigger the execution of the procedural assignments in the always statement. There are two types of sensitivity list: event-based triggering for asynchronous circuits, by using any variables; and clock-based triggering for synchronous circuits, by using only clock signals (*e.g.*, “`posedge clock`”, positive edge of “`clock`” signal). Always statements are declared by the keyword *always* and a sensitivity list, as at line 11 in List. 2.1.

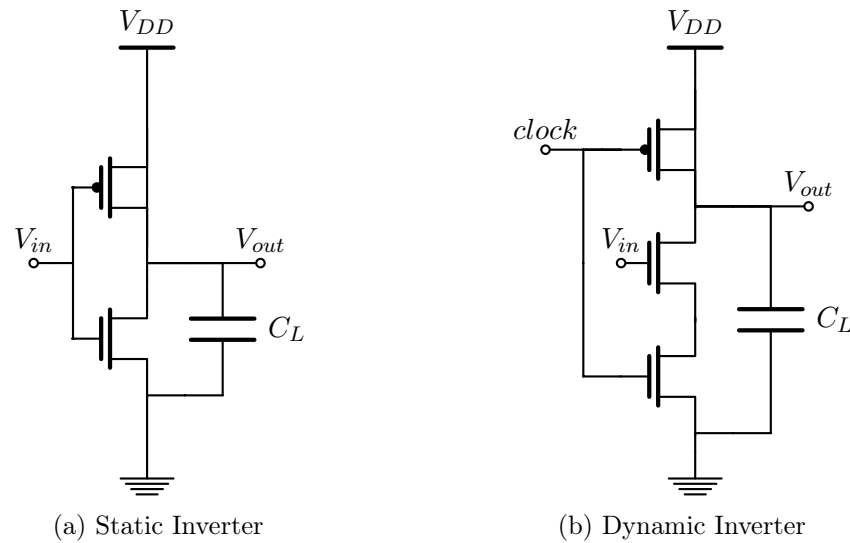




Figure 2.5: CMOS logic/gate inverters invert the given value V_{in} , on the output value V_{out} , by using PMOS  and NMOS  transistors.

2.4 Low Power Synchronous Circuits Design Techniques

The sections above in this chapter give general information on how to design any circuit. This section presents *existing low power approaches on synchronous circuits design* and introduces *the clock gating technique*, one of the most commonly used low power technique on synchronous circuits.

Power consumption P_{Total} is one of the important design constraints in chip design and generally consists of static and dynamic [59, 65]. Static power P_{Static} is consumed by a circuit during its sleep mode (*i.e.*, when the signals do not alter), whereas dynamic power $P_{Dynamic}$ is consumed by the circuit during its active mode (*i.e.*, while the signals are altering). P_{Static} is expressed as:

$$P_{Static} = I_{Static} * V_{DD},$$

where I_{Static} is the static current (*i.e.*, when the signals do not alter), and V_{DD} is the supply voltage. The $P_{Dynamic}$ equation is given below, through an example in Figure 2.5. Figure 2.5a shows a Complementary-Metal-Oxide-Semiconductor (CMOS) static logic gate

inverter, where a CMOS circuit consists of two transistors, PMOS (P-type MOS) and NMOS (N-type MOS), which are the most commonly used transistor types on chip design.

When the input V_{in} is low, the PMOS is on and the NMOS is off; and if the capacitor C_L is empty, C_L , which represents the load capacitance of this circuit, becomes charged. The energy provided from the power supply V_{DD} is then expressed as:

$$\begin{aligned} E_{V_{DD}} &= \int_0^{\infty} I_{DD}(t) * V_{DD} * dt \\ &= C_L * V_{DD}^2, \end{aligned}$$

and the energy stored in C_L is expressed as:

$$\begin{aligned} E_{C_L} &= \int_0^{\infty} I_{C_L}(t) * V_{out}(t) * dt \\ &= 0.5 * C_L * V_{DD}^2. \end{aligned}$$

As shown in the equations, half of the energy is stored in C_L while the other half is consumed by the PMOS. Then, when V_{in} switches are high, the PMOS switches off and the NMOS switches on; and the stored energy on C_L is consumed by the NMOS. Thus, if the switching activity on V_{in} continues periodically, switching power, consumed by either the PMOS or NMOS, $P_{Switching}$ is expressed as:

$$P_{Switching} = 0.5 * C_L * V_{DD}^2 * f_{sw},$$

where f_{sw} is the switching frequency of V_{out} ; and $P_{Dynamic}$ generally refers to $P_{Switching}$.

2.4.1 Clock Gating Technique

Clock signals are the circuit signals that ensure that the other signals switch periodically (with clock frequency f_{clk}). As an example, Figure 2.5b shows a dynamic inverter (*i.e.*, has clock signal). If it is assumed that V_{in} switches every clock cycle, $P_{Dynamic}$ is expressed as:

$$P_{Dynamic} = 0.5 * C_L * V_{DD}^2 * f_{clk}.$$

However, signals generally do not switch every clock cycle, so the equation can be rewritten as:

$$P_{Dynamic} = \alpha * 0.5 * C_L * V_{DD}^2 * f_{clk},$$

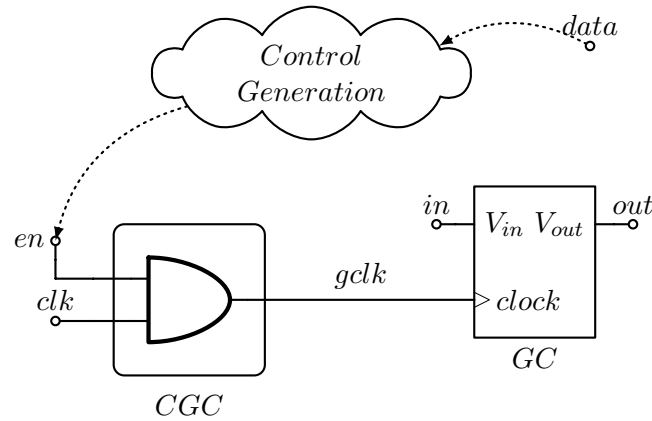


Figure 2.6: Basic Principle of The Clock Gating Technique

where α is the switching activity factor (*i.e.*, it is a value between 0 to 1 giving the probability of the switching of V_{out} per clock cycle). Thus, a circuit can consume power even when the input signals do not change. As a result, the total power of a synchronous circuit is expressed as:

$$P_{Total} = \underbrace{I_{Static} * V_{DD}}_{P_{Static}} + \underbrace{\alpha * 0.5 * C_L * V_{DD}^2 * f_{clk}}_{P_{Dynamic}}.$$

The clock gating technique is a low power technique on synchronous circuits, applied by decreasing α , where all the parameters in the equation except α depend on the technology and/or performance; it is used to selectively cut off the clock of components with the aim of reducing the power dissipation induced by the switching activity it incurs; however, it is generally applied when outputs do not change [65, 92, 94, 96, 97, 98, 100]. Figure 2.6 shows a basic principle of the clock gating technique. *GC* (Gated-Component) represents the desired component clock-gated in a circuit, where *in* and *out* are a set of input and output signals of *GC*, respectively. *GC* can be put in sleep mode by switching-off the clock signal (*clk*) by means of a Clock-Gating-Cell (represented with *CGC*), which consists of some logic element/s (in this case an “and” gate) that produce gated-clock (represented with *gclk*) by using the original clock source of a circuit and an enable-signal (represented with *en*) produced, for reducing the power consumption. The mechanism applied is the clock gating. It saves power by switching-off the clock of a component whenever desired by means of an enable signal. Various approaches (some of them are introduced in the next section) have then developed in order to generate enable signals. However, the basis of the generation of

an enable signal is a logic computation (represented with *Control Generation*) that uses a subset of variables in a circuit (represented with *data*).

2.4.2 Existing Approaches

Power efficiency is nowadays of paramount importance for constructing synchronous circuits, and various techniques can be used to reduce power consumption by decreasing each parameter on the P_{Total} equation above. Implementations start at the technological level by using various device-level techniques, such as the transistor sizing technique [47], which is the operation of sizing the width of the channel of a transistor. They continue up to the system-level by using system-level techniques, such as scheduling [32], which schedules the execution time of hardware blocks by the help of software blocks. Some techniques, such as clock-gating [87] and power-gating [22], can also be used at multiple levels, where the power-gating technique uses a gating mechanism for switching-off the supply voltage when outputs do not change. Furthermore, voltage scaling techniques are highly preferred on chip design, as they reduce the power exponentially. The multi-supply-voltage technique [61] and dynamic-voltage-scaling technique [56] separate the supply voltage to some voltage domains, where more time-critical blocks run at higher power supply voltage domains; and in the multi-threshold-voltage technique [83], lower threshold-voltage (the minimum gate-to-source voltage) transistors are used in more critical blocks in order to reduce the sub-threshold leakage, which is the leakage current between the source and drain pins of a MOS transistor when the gate voltage is less than the threshold voltage. In addition, it is possible to use some techniques together. As an example, [66] uses the clock-gating and power-gating techniques together. An optimized bus-specific-clock-gating scheme is proposed to improve the traditional XOR-based clock-gating, and the scheme chooses only a subset of FlipFlops to be gated selectively, which converts the gated FlipFlops selection from exponential to linear; then, the combinational logic elements, which completely depend on outputs of clock-gated FlipFlops, are power-gated in order to switch-off the supply voltage if the FlipFlops are performing redundant operations. As a result, various techniques can be used in order to reduce power consumption on synchronous circuits design. However, maximum power saving potential is on the clock gating, so it is the most common technique used by designers.

The clock gating technique is generally grouped into three broad categories [51]: (i) system/block-level [19], the method of switching-off the clock signal of a circuit block

as a whole by identifying idle periods (*i.e.*, enable signal) of the block for the duration of the idle periods; (ii) combinational [100], the method of switching-off the clock signal of a flip-flop/register when the output does not change; (iii) sequential [67], the method of switching-off the clock signal of a flip-flop when the output data is not used by a subsequent stage of the pipeline. Furthermore, the clock gating technique can be applied at different hierarchical levels from the system-level to gate-level [98], such as architectural-level [68], RTL [33], and circuit-level [8]. However, the RTL clock gating technique is the most commonly used technique for reducing dynamic power on synchronous circuits because most of the designers make a design at the RTL, which is the best level during the design process to reduce dynamic power [58, 94].

The target of the clock gating is the gated-clock signal generation by using a CGC (Clock-Gating-Cell) and an enable signal, where the enable signal can be implicitly part of the CGC. Various CGC topologies are reviewed, proposed, and compared in [35, 58, 92, 94, 96, 97, 98, 100]. The CGC types have advantages and disadvantages compared to each other in terms of power, area, time, glitch, and hazard, so the type selected depends on a designer/design; however, a CGC comes with extra logic units (*i.e.*, power and area), so some CGCs support the clock-gating for a group of flip-flops instead of a single flip-flop. Furthermore, various approaches are developed for the enable signal generation, grouping flip-flops, and automation. Most of the commercial tools apply the fine/coarse grain clock gating technique by replacing the muxes with CGCs, by changing the RTL code or during the synthesis, so it also leads to less die area [87, 94, 97, 100]. However, [1, 2] offer a high-level modeling environment where the gated components can be identified with enable signals during the modeling by designers; then a RTL description with CGCs is automatically generated from the high-level model. On the other hand, some approaches use idleness conditions in order to derive an enable signal. [89] exploited conditional statements and case structures, within blocks of clock-triggered assignments in HDLs to determine such conditions. Furthermore, several strategies are also available in the literature for automatic clock-gating/idle-condition extraction. [12] proposed a scheme for the automatic synthesis of gated clocks to detect idleness conditions by using explicit Finite-State-Machines (FSMs). Later, [11] used the symbolic approach in order to simplify the representation of clock-gating conditions by using Binary-Decision-Diagrams (BDD), to overcome scalability issues in the FSM approach for large FSMs. [7] also suggests an algorithm that automatically tries to approximate idleness conditions by using the ODC (Observability-Don't-Care) based approach, to overcome these issues. In addition, the clock gating technique is also used for

scheduling policies by using enable signals to reduce both step and peak power, where an enable signal can be generated by means of hardware circuits or software tools in terms of the scheduling [68, 79]. By the way, [24], who introduced a control-based adaptive clock-gating algorithm to shut down IP (Intellectual-Property) cores based on given explicit finite-state models, reviewed the hardware and software approaches in the clock-gating in terms of the enable signal generation.

2.5 Existing Tools Used on Chip Design

The production of a hardware chip consists of some steps as mentioned. The process generally starts with the description of a behavioral model and continues then by synthesis tools until the generation of a physical model at the transistor-level and is completed in a factory by using the physical model. However, the final design must meet the requirements of designers, such as functionality, performance, and power consumption, so there are several kinds of tools applied during the design process, in order to meet the requirements [81, 85]. Some tools according to the tool-type are the synthesis, functional verification, simulation, and power estimation tools. Further tools that apply a technique, such as clock gating [87], on design are also used during the process. This section introduces some design tools used in chip design.

Chip manufacturing is a highly capital intensive business, so hard chips are constructed only by chip manufacturing companies. However, the companies offer reconfigurable devices in order to let designers implement chip design by using the same models, on the devices, where a chip on a reconfigurable device is called soft chip [76, 85, 103]. Thus, some tools on chip design are offered by chip manufacturing companies as part of a vendor's design suite, and some others are provided by specialized Electronic-Design-Automation (EDA)/CAD companies; these tools are generally an Integrated-Development-Environment (IDE), which supports several kinds of features, such as synthesis and implementation environment. Some IDEs by some semiconductor market leaders and EDA companies are *Quartus* by Altera/Intel, *Xilinx ISE* by Xilinx, *Design Compiler* by Synopsys, and *RTL Compiler* by Cadence, and these IDEs can include and/or import several kinds of design tools. As an example, the simulation tool, *ModelSim*, and power estimation tool, *PowerPlay Analyzer*, by Altera have compatibility with *Quartus*.

Synthesis tools are used to generate lower level descriptions of a design automatically from the higher level behavioral models, and each tool supports different HDL/s and

abstraction level/s. Thus, IDEs include a synthesis tool, and the IDEs introduced above are generally used at the RTL, with Verilog and VHDL. Furthermore, IDEs provide an implementation environment, such as behavioral code writing, syntactic correctness, and source file creation, and support various kinds of semiconductor devices, where models are implemented on by means of synthesis tools. As an example, *Quartus* provides several kinds of FPGA and ASIC families, such as FPGA: Stratix, Arria, Cyclone, and Max, and ASIC: HardCopy, for different kinds of requirements (*e.g.*, performance, power, and cost).

Simulation tools provide the reflection of the values taken by signals/variables of a circuit, over time according to the given input values, immediately after the synthesis. Thus, the model can be tested, in terms of the functionality, by controlling the value of each variable in the model, at each instant. The given input values over time are provided by some piece of code written by using a HDL, where the code is called test-bench; and simulation results, all values over time, are saved in a file called Variable-Change-Dump (VCD).

Power estimation tools give the estimated power consumption of a chip, such as combinational, sequential, dynamic, static, average, instantaneous, and hierarchical, by using the design information. As an example, *PowerPlay Analyzer* uses the VCD file of a design and the target device technology and gives a wide range of power information.

As mentioned, the clock gating technique is one of the most common techniques used on chip design, and several commercial and academic tools that apply the clock gating technique are available to reduce dynamic power consumption. Commercial tools that support the clock gating are generally offered at the RTL and as a plug-in feature called Integrated/Intelligent-Clock-Gating (ICG) [60, 87]. These tools generally use a vendor's library where memory elements have a clock enable pin, to reduce power consumption, and add clock gating cell/s into a design by means of some logic units, for register/s; and the cell/s disable clock signal/s of the register/s in the next clock cycle if the enable pin/s are not active. In addition, clock gating cells are automatically integrated into a design during the synthesis; or designers have to specify the registers clock-gated, manually during the design, but clock gating cells are still automatically integrated into the design (*i.e.*, semi-automated). There are also a few semi-automated approaches are available to generate a RTL code with clock gating cells automatically. Among them, [1] developed an environment for high-level design with their own procedural language; [2] also provide a solution to design circuits directly using the C language. In these approaches, designers are responsible for the selection of gated components. However, [89] developed an algorithm that automatically inserts clock gating cell into RTL descriptions of circuits. In the algorithm, conditional behaviors

are first detected by parsing a RTL code; and then, idle conditions are identified by means of conditional behaviors; and clock gating cells, for individual registers, are produced by computing of idleness conditions, in order to switch-off clock signals of redundant registers.

2.6 Summary and Discussion

Low power synchronous circuits design is becoming the main solution in modern embedded systems, as the circuits are used in almost all the systems and the power consumption is one of the major design constraints in today embedded systems, limiting performance, battery life, and reliability [9, 43, 55, 70, 94, 97, 99]. In this chapter, the classification of integrated circuits, the chip design process, and chip description were introduced. It is observed that a single integrated circuit can include different kinds of sub-circuits and architectures by using a HDL, which can support that a circuit is modeled by using a mixture of concepts of different design steps; many such circuits are available in today embedded systems. Thus, it is important to have a general knowledge about integrated circuits for constructing a low power synchronous circuit.

Furthermore, the chapter also presented some existing low power synchronous circuits design techniques. As mentioned, various low power techniques are applied at different levels of abstraction; however, the RTL clock gating technique is the most commonly used technique for reducing dynamic power because most of the designers make a design at the RTL and the clock power consumes the majority of the total chip power; hence, the basic principle of the clock gating technique is also given. Additionally, the usage of EDA tools is also mentioned in detail due to their contribution to the design process, and some commercial and academic tools are introduced in terms of the role they play during the design process. As a result, low power implementations on a chip design, especially the RTL clock gating technique, and their automation are of paramount importance for constructing embedded devices.

Chapter 3

Modeling Formalisms on Hardware Circuits Designs, & Symbolic Tools, Languages, and Techniques for Discrete Control

Modeling and analysis are often required to reason on hardware circuit designs, and apply scalable techniques (*e.g.*, to compute a controller, to synthesize parts of a circuit, and to translate original designs into new designs) in order to meet various performance goals (*e.g.*, power-efficiency). Among these modeling formalisms, designers generally use the symbolic approach, which offers more abstraction, to overcome scalability issues, and need to control the system behaviors for managing performance properties, where the models in the symbolic approach consist of discrete events [4, 17, 105]. This chapter presents modeling formalisms on hardware circuits designs, the control theory of discrete event systems (*i.e.*, the discrete controller synthesis technique), and some symbolic tools, languages, and techniques for discrete control.

This chapter is organized as follows: Section 3.1 gives some common modeling formalisms used in hardware circuit design; Section 3.2 presents the principle of the discrete controller synthesis technique; Section 3.3 and Section 3.4 respectively introduce the symbolic tools/languages, BZR/Heptagon and ReaX, which apply the Discrete-Controller-Synthesis (DCS) technique; Section 3.5 presents some existing approaches that apply the DCS technique

and/or modeling formalisms; finally, Section 3.6 concludes with the summary and discussion.

3.1 Modeling Formalisms Applied on Chip Design

The chip design process today generally includes modeling formalisms, where the models abstract away the irrelevant details so that they allow designers to focus only on the essential properties of the system under design [105]. There are several types of modeling formalisms available that are used during the chip design process to manage performance properties. In this section, two common modeling formalisms applied for chip design, automata-based and data-flow-based, are introduced.

3.1.1 Automata-Based Modeling Formalisms

Automata-based models are a model of computation, where the model consists of automata/s or Finite-State-Machine/s (FSM). As an example, the formal definition of a deterministic finite automaton with a Labeled-Transition-System (*i.e.*, input/output automaton, Mealy machine) is given below [3, 4, 62, 105].

Definition (Automaton). An automaton is a tuple $A = (Q, q_0, I, O, T)$, where:

- Q is a finite set of states;
- $q_0 \in Q$ is the initial state of A ;
- I is a finite set of input events;
- O is a finite set of output events;
- $T \subseteq (Q \times Bool(I) \times 2^O \times Q)$ is a set of transition functions, where $Bool(I)$ is the set of Boolean expressions of I and 2^O is the power set of O ; and each transition denoted by $q \xrightarrow{g/a} q'$, where the source state $q \in Q$, destination state $q' \in Q$, guard $g \in Bool(I)$, and action $a \in 2^O$, is only taken when g is true.

The generic models consist of a set of states, a set of initial states, a set of transitions, and a set of guarded actions, where: each transition function determines the next state from the current state and the inputs; and each action is also a function that determines the outputs from the current state and the inputs. Thus, FSMs are a natural model of

high-level (*e.g.*, algorithmic-level) behavioral models of circuits because they are semantically very close to each other. As an example, module `m` of List. 2.1 can be easily modeled as the form of a Mealy or Moore machine, by hiding the irrelevant details (*e.g.*, it can be “`memory<=data_in;`”), where the register `state` is already in the form of a FSM. Thus, the models only describe the main behavior of a design by abstracting away the irrelevant details, so designers can focus only on the essential properties of the design.

Several kinds of tools and languages are available based on the automata-based modeling formalism and support several kinds of features, such as modular, parallel, and hierarchical descriptions of system behaviors, as well as the concurrency and/or synchronization; some languages are StateCharts, Argos, SyncCharts, Mode-Automata, and Heptagon.

3.1.2 Data-Flow Based Modeling Formalisms

Data-Flow-based (process network-)models are a model of computation, where the model is specified by a directed graph whose: nodes/vertices represent computational functions (actors) that each actor maps input data into output data during a sequence of firings¹; and edges/arcs represent communication channels (un/bounded buffers) that each carry data tokens from one node to another when fired [4, 31, 62, 63, 82, 105].

Hardware designers generally prefer to use data-flow models (symbolic approach) instead of automata-based models, in order to overcome scalability issues (*e.g.*, state space explosion) because the symbolic approach is more abstract than FSMs. In other words, FSMs/automatons are cycle-accurate models (*i.e.*, they have a possible state space explosion problem for infinite or too large FSMs/automatas) and are more detailed than data-flow models; however, data-flow models emphasize the relationship of the inputs and outputs.

Three main distinct variants of the dataflow model of computation have emerged in the literature: Synchronous-Data-Flow (SDF), Kahn-Process-Network (KPN), and Synchronous-Languages. These are also the most frequently used classes of the data-flow family applied for synchronous hardware circuits designs, which are used in almost all embedded systems today as mentioned in Chapter 2, and they are introduced below.

¹Firing rule: is a set of prefixes with certain technical conditions to ensure determinacy; an actor is fired, if the condition is provided, by consuming data tokens from input streams (links/edges) of the actor and producing data tokens on the output streams.

Synchronous Data Flow (SDF)

The Synchronous-Data-Flow (SDF) is a special case of the data-flow family, where the models include a fixed number of data tokens (*i.e.*, specified a priori) produced or consumed on each link during at each firing by each actor [4, 31, 62, 82]. Actors in a SDF graph can be fired periodically and at different iterations (*i.e.*, no notion of time), and together. Thus, the models have the concept of an abstract clock, which makes the models a natural model for synchronous circuit design. The approach is generally used for infinite streaming and signal processing (*e.g.*, audio and video) applications.

Kahn Process Network (KPN)

The Kahn-Process-Network (KPN), which is a data-flow-based modeling formalism, is a concurrent continuous function on infinite streams and generally can be seen as a generalization of the SDF, where the notion of firings has been absent from the models (*i.e.*, computation times and communication times may vary) [4, 64, 75]. Actors represent computational functions like as in the SDF; and edges represent unidirectional un/bounded communication channels based on FIFO (First-In-First-Out) principles, where the writing policies are non-blocking (*i.e.*, queues are a priori un/bounded as in the SDF) and the reading policies are blocking (*i.e.*, if a queue is empty, the process blocks until the queue becomes non-empty). This means that a process is stalled if the process attempts to read from an empty input stream. Thus, the KPN can be used to describe systems where the amount of data produced and consumed by a process is not statically determined, and is generally applied to support scheduling policies on hardware circuits.

Synchronous Languages

SDF models are a natural model of synchronous circuits as mentioned; however, they have one main limitation that conditional expressions could not be represented on the graphs [4, 18, 71]. On the other hand, synchronous languages apply the SDF approach by abstracting the clock and also supporting to use formal mathematical semantics, so the modeling formalism with synchronous languages is pretty suitable to model hardware circuits designs.

The main concept of synchronous languages is building a program where the basic building blocks are called data-flow nodes, which consist of instants and reactions. Instants can be accepted as a firing (atomic action), or discrete time which behavioral activities

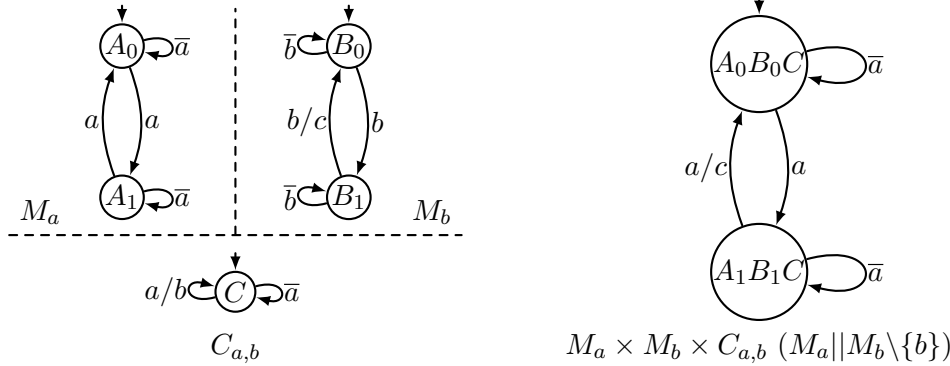
executed depend on. At each instant, input data maps to output data by means of some mathematical computation that includes conditional statements. This execution cycle is called the reaction.

The behavior of synchronous nodes can be described with Mealy machines; communication and/or association of nodes/Mealy machines can be supported/notified by composition operators. Synchronous composition (sometimes called completely synchronous composition) operator is denoted by “ \times ”, which supports to show both communication and association of Mealy machines. As an example, synchronous product of two Mealy machines $M_a \times M_b$ is a tuple $(Q_a \times Q_b, (q_{0a}, q_{0b}), I_a \cup I_b, O_a \cup O_b, T')$, where: $Q_a \times Q_b$ is the combination of all states (*i.e.*, $q_a \cdot q_b \in Q_a \times Q_b$); T' is defined by $(q_a \xrightarrow{g_a/a_a} q'_a \in T_a) \wedge (q_b \xrightarrow{g_b/a_b} q'_b \in T_b) \Rightarrow q_a \cdot q_b \xrightarrow{g_a \cap g_b / a_a \cup a_b} q'_a \cdot q'_b \in T'$. On the other hand, parallel composition (sometimes called synchronous composition) operator denoted by “ \parallel ” supports the association of Mealy Machines (*i.e.*, it is suitable to show independent systems) and does not make any synchronization between them; however, synchronization can be supported together with encapsulation operation, which is used to express a limitation of the range of a given signal for enforcing the synchronization between nodes. As an example, parallel composition of two Mealy machines $(S = M_a \parallel M_b = (Q, q_0, I, O, T))$ through (encapsulation *w.r.t.*) a set of inputs and outputs of S , $\Gamma \subseteq I \cup O$, allows the construction of a new Mealy machine $(S \setminus \Gamma = (Q, q_0, I \setminus \Gamma, O \setminus \Gamma, T'))$, whose behavior is no longer influenced by a possible emission of Γ by another part of the system itself, where: T' is defined by $(q \xrightarrow{g/a} q' \in T) \wedge (g^+ \cap \Gamma \subseteq O) \wedge (g^- \cap \Gamma \cap O = \emptyset) \Rightarrow q \xrightarrow{\exists \Gamma. g/a \setminus \Gamma} q' \in T'$, where $g^+ = \{x \in g \mid (x \wedge g) = g\}$ and $g^- = \{x \in g \mid \neg(x \wedge g) = g\}$.

Several languages (*e.g.*, Esterel, Lustre, Signal, Argos, Heptagon, and ReaX) that apply the SDF (symbolic approach) are available, where the concept of synchronous languages is introduced through example languages, Heptagon and ReaX, below.

3.2 Discrete Controller Synthesis (DCS)

Discrete Controller Synthesis (DCS), the control theory of Discrete-Event-Systems (DES—[23, 90]), allows the use of constructive methods ensuring, *a priori* and by means of control, required properties on a system’s behavior. The control theory of DES (Supervisory Control Theory) advocates that a system can be built by discrete events (*i.e.*, a system that has not been controlled yet, called plant-P in terms of DCS community—*e.g.*, invoke or completion



(a) Two isolated Mealy machines (i.e., which do not communicate), M_a and M_b , and the controller $C_{a,b}$, which guarantees the desired property after encapsulation by the controllable signal b .

(b) Representation of The Behavior of The Controlled System

Figure 3.1: Representation of The Control Theory of Discrete Event Systems

of a task and the failure or frequency switch of a processor) with a controller (supervisor-S) that guarantees the desired property (specification-Sp) by using the controllable and uncontrollable signals, without modifying the behavior of the discrete events. The principle of the theory is illustrated in Figure 3.1 [13]. Figure 3.1a shows system properties, where: the discrete event systems are represented with Mealy machines, M_a and M_b ; the controller $C_{a,b}$ guarantees (after the composition) the desired property that if M_a is in the state A_0 (respectively A_1), then M_b is in the state B_0 (respectively B_1); and the signal b is assumed to be a controllable signal. Figure 3.1b describes the desired property of the system (i.e., synchronous parallel composition of M_a , M_b , and $C_{a,b}$, by encapsulating the signal b), where the given property is verified.

Usually, the starting point of these theories is: given a model for the system and control objectives, a *controller* must be derived by various means such that the resulting behavior of the closed-loop system meets the *control objectives*. The principle of the DCS is represented in Figure 3.2 [4]. In the figure, the given system S (consists of discrete events/states/FSMs with input/output/local signals) fulfills/satisfies the desired objective o (expressed in terms of the input/output/local signals and/or states) by means of the controller C , where: X is a set of output(-event)s of S ; Y is a set of input(-event)s of S that consists of a set of controllable and uncontrollable inputs Y_c and Y_{uc} , respectively; and C is automatically

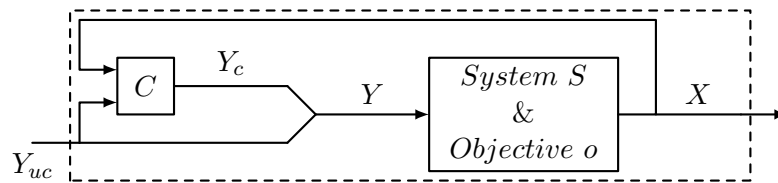


Figure 3.2: Principle of The Discrete Controller Synthesis

obtained from S and o (both given properties) via appropriate synthesis algorithms that automatically compute (by exploring the system state space and Y_{uc}) a constraint on Y_c (*i.e.*, guarantee the desired property/objective o that has to be enforced by control).

Note that synthesis algorithms work like model checking to guarantee the desired property/objective; however, there is not always any controller available that guarantees the control objective, with given properties for a system (*i.e.*, discrete controller synthesis may fail). In addition, there can be several controllers that provide the same desired objective (*e.g.*, a controller maybe enforce the system stays stable, *i.e.*, no state transition, in order to meet the desired property); however, maximally permissive (minimally restrictive) controllers are considered in the thesis, where the controllers provide the largest possible set of correct behaviors of the system (*i.e.*, only enforce unreachability of states violating the desired property) [4, 18, 54].

In the thesis, the specification of systems (*i.e.*, plant) is considered using the synchronous data-flow languages, such as ReaX and Heptagon; hence, the synthesis algorithms are able to synthesize controllers satisfying various kinds of control objectives, such as safety, reachability, attractivity, liveness, and optimal; the typical examples, safety/invariant and optimal control objectives, are introduced below, and how the controller that satisfies them is (automatically) generated is explained through example synchronous data-flow languages, Heptagon and ReaX [4, 18, 72].

Safety/Invariant Control Objective: is a subset of the state space and is described using state and input variables with associated dynamics. The desired objective is the enforcement of some invariant a priori not satisfied by the system: a controller is to be computed that restricts the admissible values for a subset of the input variables (referred to as controllable variables) so that the resulting controlled system satisfies the invariant (*e.g.*, typically, forcing a predicate over a subset of the state space of the system to be always true).

Optimal Control Objective: is to optimize a cost function (summed), which is typically

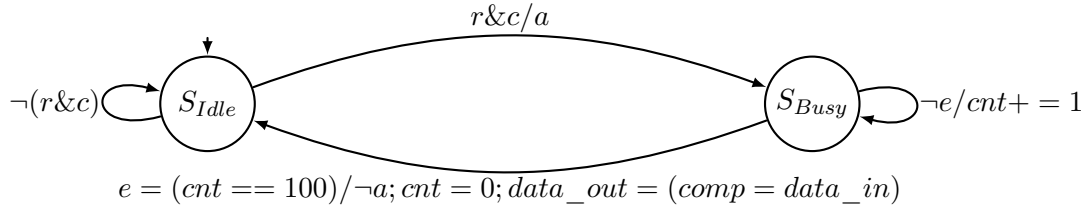


Figure 3.3: Task $\tau(r, c, data_in)=a, data_out$, faking computation, in the form of a Mealy machine, where: r (request), c (control), and $data_in$ (input data) are input variables; a (active—keeps a memorized value, initially `false`) and $data_out$ (output data) are output variables; e (executed), cnt (counter—keeps a memorized value, initially 0), $comp$ (computation—keeps a memorized value, initially 0), and $S=\{Idle, Busy\}$ (state—keeps a memorized value, initially `Idle`) are local variables.

a total mapping from state and input valuations into some (partially-)ordered set, such as the Rationals, over a sliding window of a given number of ticks (*i.e.*, along the specified paths of state transitions), so an optimal control algorithm works, from current states to target states, to produce a controller at the best cost of controllable events against the worst moves of uncontrollable events.

3.3 BZR/Heptagon Synchronous Language

Heptagon is a reactive synchronous data-flow language, where programs are built as parallel and hierarchical compositions of data-flow *nodes*, each having *input*, *local*, and *output flows*. The body of a node describes how input flows are transformed into output flows, in the form of a set of *equations*. These equations define the values of outputs (and possible local flows), using the current values of inputs, and the current *state* of the node: the latter is made of memorized values expressed by “`last`” values of flows. New values for input flows are given at each *execution step*, where equations are then evaluated all together, and values of output flows are updated accordingly [4, 29, 30].

As an example, List. 3.1 (**Heptagon node τ**) symbolically encodes the Mealy machine (task τ) given in Figure 3.3, which represents the behavior of a model (*e.g.*, hardware design). A **Heptagon** node is described by the keyword “`node`” with a name (in this case τ), a set of input variables (in parentheses), and a set of output variables by the keyword “`returns`”, as at line 2; the body of the node, between the keyword pair “`let`” and “`tel`”, describes the behavior of the node (in the form of mathematical equations/expressions that can include


```

1 type S_t = Idle | Busy
node t (r,c: bool; data_in: int) returns (last a: bool = false; data_out: int)
var last S: S_t = Idle; last comp: int = 0; last cnt: int = 0; e: bool;
let
5  S = if last S = Idle & r & c then Busy else
      if last S = Busy & e then Idle else
        last S;
  a = if last S = Idle & r & c then true else
      if last S = Busy & e then false else
        last a;
10 comp = if last S = Busy & e then data_in else
         last comp;
  cnt = if last S = Busy & not e then last cnt + 1 else
        if last S = Busy & e then 0 else
          last cnt;
15 e = cnt==100;
   data_out = comp;
tel

```

List. 3.1: Heptagon node `t`, symbolic encoding of the Mealy machine of Figure 3.3.

```

1 node twotasks (r1,r2,c1,c2: bool; data_in1,data_in2: int)
  returns (last a1,a2: bool = false; data_out1,data_out2: int)
let
  (a1,data_out1) = inlined t(r1,c1,data_in1);
5  (a2,data_out2) = inlined t(r2,c2,data_in2);
tel

```

List. 3.2: Heptagon node `twotasks`, parallel composition of two instantiations of the node (task) `t` of List. 3.1.

conditional statements). Furthermore, Heptagon allows to define local variables by the keyword “var”, and memory elements also need the keyword “last” and an initial value, as at line 3. Any variable defined in a Heptagon node should take a value only in a specified (a priori) domain, such as “int” and “bool”; further, Heptagon allows to specify a custom enumerated type as at line 1.

Heptagon nodes can be reused by instantiations, and one can compose the nodes using instantiations in a parallel or hierarchical (modular) way. As an example, the node `twotasks` of List. 3.2 shows the parallel and hierarchical/modular composition of two instantiations of the node (task) `t` of List. 3.1, by the keyword “inlined” as at lines 4 and 5.

BZR [30] extends Heptagon with a behavioral contract mechanism, and encapsulates a DCS tool (*e.g.*, Sigali and ReaX) in its compilation process. An *invariant* and *controllable flows* (each taking its value in the Boolean domain `bool = {false, true}`) can be specified for Heptagon nodes using *contracts*. When it encounters a node featuring a contract, the BZR compiler involves a symbolic DCS algorithm to automatically produce a *controller*

```

1 node controlledTasks (r1,r2: bool; data_in1,data_in2: int)
    returns (last a1,a2: bool = false; data_out1,data_out2: int)
    contract enforce not (a1 & a2) with c1,c2: bool;
    let
5   (a1,data_out1) = inlined t(r1,c1,data_in1);
   (a2,data_out2) = inlined t(r2,c2,data_in2);
    tel

```

List. 3.3: Heptagon node `controlledTasks`, symbolic encoding of the desired property of the system, where the system, plant, is the parallel composition of two instantiations of the node (task) `t` of List. 3.1.

constraining the values of the controllable flows so as to guarantee that the resulting *controlled node* satisfies the invariant. The controllers produced take the form of as many predicates as controllable flows, that implement the following behavior: considering every controllable flow c in turn (according to their order of declaration), the controller tries to assign c to `true` unless this could lead to a potential violation of the desired invariant in subsequent execution steps.

As an example, the Heptagon node `controlledTasks` of List. 3.3 (which includes the contract mechanism) produces (by applying the DCS technique) a controller enforcing that the expression “`not (a1 & a2)`” (given desired property) holds using the controllable variables, `c1` and `c2`, (*i.e.*, the controller ensures that the two tasks running in parallel is not both active, `Busy` state, at the same time) as at line 3. Note that `c1` and `c2` are no longer placed in (uncontrollable-)input flows, as they are controllable signals (*e.g.*, as at lines 1 and 3).

Programmatically, how the contract mechanism (or a DCS implementation) synthesizes a controller that guarantees the desired property is explained on the tool `ReaX`, below.

3.4 ReaX Synchronous Language

`ReaX`: is a reactive synchronous data-flow language, where reactive systems (*i.e.*, plants) are modeled by Arithmetic-Symbolic-Transition-Systems (ASTSs), which are a finite set of transition systems with state and/or input variables in a finite and/or infinite domain; and is a (DCS-)compiler that synthesizes controllers ensuring given properties (*i.e.*, specification), where the systems are modeled by ASTSs [15]. `ReaX` allows controlling of infinite reactive synchronous systems modeled by ASTSs for several kinds of control objectives (*e.g.*, safety, reachability, and optimal). As an example, the invariance/safety control problem

of the model of ASTSs (*i.e.*, the DCS paradigm applied by ReaX) is formally explained below.

Definition (ASTS). An ASTS is a tuple $S = (X, I, T, A, \Theta_0)$, where:

- $X = \{x_1, \dots, x_n\}$ is a finite set of vector state variables ranging over in/finite domains;
- $I = \{i_1, \dots, i_m\}$ is a finite set of vector input variables ranging over in/finite domains;
- T is a finite set of guarded state transition functions using variables in $X \cup I$;
- $A \in X \cup I$ is a predicate for expressing an assertion about the possible values of the inputs depending on the current state;
- Θ_0 is a predicate expressing the initial values of variables in X .

To each transition function in an ASTS model S , one can make correspond an infinite sequence of $x \xrightarrow{i} x'$, where x and x' are a value of the domain of $x_j \in X$ and i is a value of the domain of $i_j \in I$. This kind of system is called an Infinite-Transition-System (ITS) (denoted by $[S]$), and the set of transitions of ITS $[S]$ is denoted by T_S , where $x \xrightarrow{i} x' \in T_S$.

Given an ASTS S , its associated ITS $[S]$, and a predicate (*i.e.*, desired property or specification) Φ over X on S , S should satisfy/fulfill Φ (noted by $S \models \Phi$) by some restriction (*i.e.*, control by using controllable variables); S can be rewritten as $S = (X, I_{uc} \cup I_c, T, A, \Theta_0)$, where I divided into controllable (I_c) and uncontrollable (I_{uc}) input variables. Thus, solving the discrete controller synthesis problem is to compute a controller (predicate A_Φ) on $S' = (X, I_{uc} \cup I_c, T, A_\Phi, \Theta_0) \models \Phi$ and $\forall v \in X \cup I_{uc} \cup I_c, A_\Phi(v) \Rightarrow A(v)$, where v is a variable on S .

The synthesized controllers in ReaX are maximally permissive, and the computational steps rely on some fix-point computation of the predicate. Given an ASTS S , its associated ITS $[S]$, and a predicate Φ , the approach is illustrated in Figure 3.4. The computational steps are: (i) first, bad states Bad , which is the set of states that do not satisfy Φ , are computed as $Bad = \neg\Phi$; (ii) then, the set of forbidden states I_{Bad} , which ensures the unreachability of Bad with uncontrollably, is computed based on least fix-point calculation by using T_S ; (iii) last, the controller $A_\Phi \subseteq D_X \times U \times C$ (where D_X , U , and C are the domain of X , I_{uc} , and I_c , respectively) is computed by using $I_{Bad}^c = \neg I_{Bad}$, which gives the set of non-forbidden states represented with the gray color in Figure 3.4.

The resulting controllers are correct and maximally permissive but non-deterministic (*i.e.*, controllable variables may not be a singleton). To obtain an executable deterministic

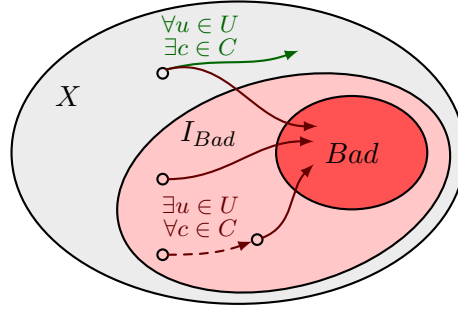


Figure 3.4: Computational Steps on The State Space X

controller, the triangulation function, which requires prioritizing the controllable input variables, provides a deterministic (*i.e.*, singleton) set of the predicate of controllable variables.

3.5 Existing Approaches

As mentioned in Chapter 2, power efficiency is nowadays of paramount importance for constructing circuits and various techniques can be used to reduce power consumption. One of the significant technique, which can be used with the techniques mentioned in Chapter 2, is modeling formalisms. Designers generally apply several types of modeling and analysis, to reduce the power as well as to optimize various performance properties [105]. [1, 2] provide a high-level automata-based modeling environment, which generates a low-level description that includes some clock gating logic from the given model, for power efficiency. [6] proposes a systematic modeling framework by applying automata-based modeling formalism in order to manage system behaviors of dynamically reconfigurable hardware architectures. Automata-based models are a natural model of hardware circuits, as they are semantically very close to each other; however, designers generally use the family of data-flow models, because of the well-known state explosion problem [105]. [42] proposes the reconfigurable data-flow graph (RDF), which is a variant of the SDF, and extends the SDF with some rules in order to overcome the lack of the capability to express the dynamism on a SDF graph. [95] employs a run-time scheduling algorithm by capturing system behaviors of dynamically reconfigurable architectures based on the KPN. [40] introduces the light-weight data-flow (LWDF) methodology, adapts it to the hardware description languages (*e.g.*, Verilog and VHDL), and applies some low-power techniques. [14] considers the management

of dynamically partially reconfigurable hardware architectures based on the synchronous languages aspect. On the other hand, [105] develops a formal conformance relation between the heterogeneous formalisms (*i.e.*, the automata/FSM and SDF based approaches) for hardware circuits; and [29] develops a tool that lets the mixed usage of the synchronous language data-flow and automata based approaches.

The techniques applied on hardware models for meeting various performance properties can be provided by means of several kinds of self-management approaches, such as model checking, heuristics, and machine learning techniques. [39, 95] apply some self-management strategies, and [69] discusses some existing standard control techniques applied to some hardware systems; the main feature of discrete control techniques provides a synthesizable controller with formal correctness [4]. DCS, the control theory of discrete event systems, allows the use of constructive methods ensuring, a priori and by means of control, required properties on a system's behavior [23, 90]. Finding DCS algorithms computing such controllers in the case of finite-state symbolic systems (*i.e.*, where the state and input variables are Booleans only) has been the objective of several studies, and led to several implementations, *e.g.*, by [72, 74]. [15] later extended these studies for infinite-state systems, featuring numerical state and input variables. Meanwhile, other studies considered optimization objectives, given partial order relations [73, 74], or cost functions [34], that essentially consist in symbolic adaptations of Bellman's algorithm for the computation of optimal strategies using dynamic programming [10]. Several hardware design techniques rely on DCS to provide several kinds of performance properties. [88] presents a high-level design flow for reconfigurable FPGA-based System-on-Chip (SoC); they model potential reconfiguration behaviors and manually derive a controller that automatically takes reconfiguration decisions. Later, [50] and [5, 6] were among the firsts to apply DCS algorithms for reconfiguration management in SoC design. Doing so, they could automate the generation of controllers, thereby exploiting the formal correctness and guarantees that DCS techniques provide. In particular, [5, 6] model the applications' behaviors and the needed resources (area in hardware—*i.e.*, regions of the FPGA) using explicit automata; they then use a symbolic DCS algorithm to automatically compute a reconfiguration manager for the system. [14, 15] introduce the recent advances in control algorithms for symbolic infinite-state systems with applications to quantitative models. Few works have addressed the problems of enforcing the optimization and various kinds of control objectives on this kind of symbolic models and tools; they mostly derive from the seminal work of [90]. [36, 72, 29, 77, 15] implemented tools that are suitable for enforcing several objectives, such as optimization, safety, and

reachability properties.

3.6 Summary and Discussion

This chapter presented the existing modeling formalisms on hardware circuits designs, & symbolic tools, languages, and techniques for discrete control. The automata based and data-flow based models are first introduced; then, three distinct variants of the data-flow family of computation that emerged in the literature, SDF, KPN, and synchronous languages, are detailed. KPN seems as a generic model of the data-flow family and can be used to describe systems where the amount of data produced and consumed by a process is not statically determined. However, synchronous data-flow languages (*e.g.*, Heptagon and ReaX), where the models have the concept of an abstract clock and formal mathematic semantics, are a very suitable model for hardware circuits (especially synchronous circuits). Synchronous data-flow languages distinguish among the modeling formalisms, as they do not have a state explosion problem as in FSM (automata) models and they support conditional expressions unlike SDF [4, 105].

As mentioned, modeling and analysis are often required to reason on hardware circuit designs in order to meet various performance goals; such goals are generally achieved by computing a controller [4, 105]. Compared to other existing techniques for achieving various performance goals, the main advantage of DCS (the control theory of discrete event systems) provides a synthesisable controller with formal correctness. This chapter also presents the supervisory control theory (*i.e.*, DCS). Furthermore, ReaX and BZR (synchronous data-flow languages), which are encapsulated with the DCS operation, are introduced and how they provide formal correctness are explained. Additionally, several existing approaches in the literature that apply the DCS technique and/or modeling formalisms are presented. As a result, it is observed that hardware circuits can be modeled to meet scalability issues by synchronous languages, and various performance goals can be achieved by such languages (*e.g.*, ReaX and BZR) encapsulated with the DCS technique.

Chapter 4

Exercising Symbolic Discrete Control for Designing Low-Power Hardware Circuits: an Application to Clock-Gating

In this chapter, we devise a tool-supported framework for achieving power-efficiency of hardware chips from high-level designs described using the popular hardware description language Verilog. We consider digital circuits as hierarchical compositions of sub-circuits, and achieve power-efficiency by switching-off the clock of each sub-circuit according to some clock-gating logic. We encode the computation of the latter as several small symbolic discrete controller synthesis problems, and use the resulting controllers to derive power-efficient versions from original circuit designs. We detail and illustrate our approach using a running example, and validate it experimentally by deriving a low-power version of an actual Reed-Solomon decoder. The contents and results of this chapter have been presented in [80].

4.1 Introduction

This section presents our motivation, and our contribution to the design of low-power synchronous circuits, where our approach applies the clock gating technique at the register

transfer level by exploiting the symbolic discrete controller synthesis technique by means of the tool, BZR; and the section also gives an overview of the chapter.

4.1.1 Motivation

Power efficiency of digital circuits is nowadays of paramount importance for constructing embedded electronic devices, and various mechanisms can be used to reduce power consumption in hardware chips. At the technological level, these include clock-gating, multi-supply voltage, and power-gating for instance [51]. In synchronous circuits in particular, clock-gating is used to selectively cut off the clock of components with the aim of reducing the power dissipation induced by the switching activity it incurs. This technique mostly consists in computing the Clock-Gating Logic (CGL) for sub-circuits, and then translating each CGL itself into a piece of circuit whose output wires can be used to switch-off (to gate) the clocks that drive the sub-circuits.

In this work, we observe that CGL computation is actually a feedback control problem, where the sub-circuit constitutes the system to control, and the objective is to switch-off its clock whenever possible. This new perspective constitutes a first step towards employing other control techniques for producing self-adaptive power efficient digital circuits, *e.g.*, that would automatically adapt to the remaining capacity of some battery according to an objective power/performance trade-off.

Discrete Controller Synthesis (DCS), the control theory of Discrete Event Systems (DES — [23, 90]), allows the use of constructive methods ensuring, *a priori* and by means of control, required properties on a system's behavior. Usually, the starting point of these theories is: given a model for the system and control objectives, a *controller* must be derived by various means such that the resulting behavior of the closed-loop system meets the *control objectives*. A typical example is the *safety control problem* for *symbolic* systems (*i.e.*, described using state and input variables with associated dynamics), where the desired objective is the enforcement of some *invariant a priori* not satisfied by the system: a controller is to be computed that restricts the admissible values for a subset of the input variables (referred to as *controllable variables*) so that the resulting *controlled system* satisfies the invariant. Finding DCS algorithms computing such controllers in the case of finite-state symbolic systems (*i.e.*, where the state and input variables are Booleans only) has been the objective of several studies, and led to several implementations, *e.g.*, by [72, 74]. [15] later extended these studies for infinite-state systems, featuring numerical state and input

variables. Meanwhile, other studies considered *optimization* objectives, given partial order relations [73, 74], or cost functions [34]. Most of these solutions adapt Bellman’s algorithm for the computation of optimal strategies using dynamic programming [10].

In this work, we exploit DCS principles through the use of the BZR environment [30], that integrates symbolic DCS within the reactive data-flow language Heptagon.

4.1.2 Contributions

We present a framework involving symbolic DCS to achieve energy efficiency of hardware chips. We consider circuits described using the Hardware-Description-Languages (HDL—*e.g.*, Verilog) at the Register-Transfer Level (RTL) abstraction. RTL descriptions are high-level hierarchical compositions of components, registers, and logical operators, linked using wires. They can be converted into equivalent digital chip designs for Application-Specific-Integrated-Circuits (ASICs) or Field-Programmable-Gate-Arrays (FPGAs).

The framework broadly comprises the following steps: First, the RTL description of the original circuit is translated into a set of synchronous models where controllable variables represent output wires of CGLs. These models are associated with control objectives whose enforcement guarantee the correct behavior of the CGLs. Second, the latter are obtained using a symbolic DCS algorithm. Last, the CGLs are translated into pieces of circuits that are then integrated into a new, clock-gated circuit design. Our translation algorithm is parametrized with a set of variables to be picked from the HDL description, and that is used to abstract away most of the circuit in order to: (i) focus on the portion of its sequential logic that is relevant for expressing the CGLs; and thus (ii) restrict the size of the DCS problems. Our algorithm automatically generates interpreted non-controllable inputs called *oracles* to model the non-determinism introduced by the abstractions and allow the computation of deterministic, hence implementable, CGLs.

4.1.3 Overview

We give a running example along with a description of the Verilog HDL and BZR in Section 4.2, and use it to describe and illustrate the framework in Section 4.3. We exercise our technique on a realistic case study in Section 4.4, and conclude with the summary and discussion in Section 4.5.

```

1 module m (input clk, input r, input [7:0] i,
           output [7:0] o, output e);
   parameter idle = 0, busy = 1;
   reg state = idle;
5   reg [6:0] cnt = 0; // to fake lengthy computations
   reg [7:0] m; // internal memory
   assign o = m; // always output memorized value
   assign e = (cnt == 100); // raise e when counter reaches 100
   always @(posedge clk)
10    case (state)
       idle: if (r) begin m <= i; state <= busy; end
       busy: if (e) begin
15            cnt <= 0; // reset counter
               if (r) m <= i; // restart immediately
               else state <= idle;
               end else cnt <= cnt + 1;
   endcase
endmodule

```

List. 4.1: Verilog module `m`, faking computations on data given on input wires `i` upon request `r`. Output wires `o` carry its result, available when `e` becomes 1.

4.2 Running Example

We now introduce the Verilog HDL using a running example, and then describe the fragment of the Heptagon language that we use for the modeling and computation of CGL for RTL circuits.

4.2.1 The Verilog Hardware Description Language

Verilog is an HDL dedicated to the design of electronic systems. In particular, it can be used to specify *synchronous circuits*. The description of such a circuit in Verilog consists of a *main module* made of an assemblage of *registers*, *wires*, *gates* and/or *sub-modules*. Each of the latter components features an interface that comprises *input* or *output wires* or *registers*. Verilog provides several constructs to program modules, such as conditional and case statements, wire/register declarations and assignments, and event detection (*e.g.*, positive edge detection, triggered when the value carried by a wire transitions from 0 to 1). One input wire, usually called `clk`, carries a clock that is used to trigger changes in the values of registers.

We give in List. 4.1 an example specification of a module “`m`”. It starts with the declaration of its interface, here comprising basic wires (*e.g.*, `clk`, `r`, and `e`) or wire arrays (*e.g.*, `i` and `o`, here used to carry data). A declaration of constants used internally (`idle` and `busy`) follows, along with internal registers. Assignments at lines 7 and 8 describe the logical values

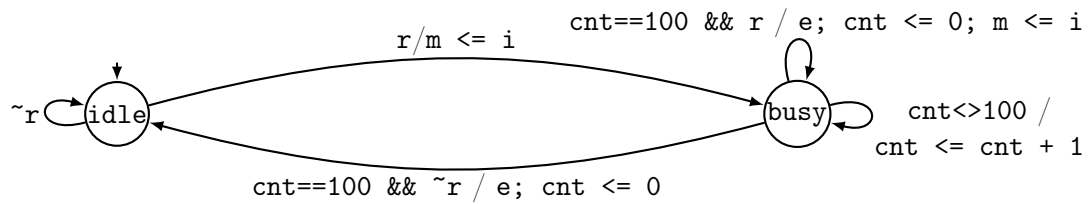


Figure 4.1: Mealy machine symbolically encoded by register `state` in Verilog module `m` of List. 4.1, decorated with operations on `cnt` and `m`. The initial value of `cnt` is 0.

taken by the corresponding wires at any instant by means of logical expressions. The code following “`always @(posedge clk)`” consists in conditional “clock-triggered” assignments to internal or output registers, denoted by “`<=`”. For instance, the statement “`state <= busy`” at line 11 states that, at every instant t where a positive edge of `clk` occurs, the value memorized by the register `state` takes the value `busy` if the condition `r` holds (*i.e.*, `r` carries the value 1 at the very instant where `clk` becomes 1); notice the value of `state` would actually become `busy` at a subsequent instant $t + \varepsilon$. The internal registers in `m` *symbolically* encode a finite-state automaton, that we represent in the form of a Mealy machine with variables in Figure 4.1.

We show in List. 4.2 an example module `main` making use of `m`. Sub-module instantiations `m1` and `m2` are at lines 8 and 10. `main` features an internal working mode memorized using the register `cfg` taking its values in `{LQ,HQ}`: as can be seen on line 18, upon a positive edge of `clk`, if the request signal `r` holds and neither `m1` nor `m2` is currently computing (`r1 & ~error`), `cfg` takes the current value of the input wire `mode`. The selection of values for output data `o` and end signal `e`, along with the triggering of computations by sub-module instance `m2`, depend on `cfg`: in mode `LQ` an input data `i` is only processed by `m1`, whereas in mode `HQ` this data is serially processed by `m1` and then `m2`.

In the remainder of the chapter, we consider Verilog circuits given as directed acyclic graphs whose nodes are modules, where arcs describe the “instantiates” relation, and with a single source node representing the module that describes the whole circuit. In turn, every Verilog module `M` is considered as a tuple $M = (I_M, O_M, R_M, Sub_M)$ where: I_M denotes input wires; O_M denotes output wires; R_M are internal or output registers; Sub_M is a set of sub-module instantiations. A Verilog module with a non-empty set of sub-module instantiations is called a *super-module*.

Clock-gating in Verilog Circuit Specifications: Consider a module instance mi . We say that η_{mi} is a *Clock-inhibition Predicate* (CIP) for mi if, upon an edge of the clock of mi

```

1 module main (input clk, input r, input mode, input [7:0] i,
              output [7:0] o, output e, output error);
   parameter LQ = 1, HQ = 0; // Low/High Quality modes
   reg cfg = LQ;
5   reg wait_m1 = 0, wait_m2 = 0; // sub-module's working statuses
   wire [7:0] o1, o2; // output data of sub-modules
   wire r1 = r & ~error, e1; // request & end wires for m1
   m m1 (.clk(clk), .r(r1), .i(i), .o(o1), .e(e1));
   wire r2 = e1 & (cfg == HQ), e2; // request & end wires for m2
10  m m2 (.clk(clk), .r(r2), .i(o1), .o(o2), .e(e2));
   // raise error upon request before end of m1 or m2
   assign error = r & (wait_m1 | wait_m2);
   // select data output and end signal based on configuration
   assign o = (cfg == LQ) ? o1 : o2;
15  assign e = (cfg == LQ) ? e1 : e2;
   always @(posedge clk) begin // behavioral assignments
       if (r1) begin // accept new request
           cfg <= mode; // change mode
           wait_m1 <= 1; // wait for end of m1
20       end
       if (e1) wait_m1 <= 0;
       if (cfg == HQ) begin // HQ Mode
           if (r2) wait_m2 <= 1; // wait for end of m2
           if (e2) wait_m2 <= 0;
25       end
   end
endmodule

```

List. 4.2: Verilog module `main`, instantiating `m` twice.

(*e.g.*, `clk`), η_{m_i} holds if the values of every registers and output wires of m_i are strictly equivalent before and after the edge of the clock. If translated into a CGL, η_{m_i} can then be used to save dynamic power by gating the clock of m_i by preventing flip-flop switches. Considering our example Verilog module `main` again, a piece of circuit encoding the CGL for sub-module instances `m1` and `m2` would typically output two wires, say η_{m1} and η_{m2} , used to filter each of their respective input clocks `clk`. The extract of Verilog code instantiating the clock-gated instance of `m1` (replacing the beginning of line 8 in List. 4.2) would then be “`m m1 (.clk(clk & ~ η_{m1}),`”.

4.2.2 A Fragment of Heptagon

Heptagon [30] is a reactive data-flow language where programs are built as parallel and hierarchical compositions of data-flow *nodes*, each having *input*, *local*, and *output flows*. The body of a node describes how input flows are transformed into output flows, in the form of a set of *equations*. These equations define the values of outputs (and possible local flows), using the current values of inputs, and the current *state* of the node: the latter is made

```

1 type state_t = Idle | Busy
node m (r: bool; i: int) returns (e: bool; o: int)
var last state: state_t = Idle;
    last cnt: int = 0; last m: int = 0;
5 let
    state = if last state = Idle & r then Busy else
            if last state = Busy & e & not r then Idle else
            last state;
    cnt = if last state = Busy & e then 0 else
          if last state = Busy & not e then last cnt + 1 else
          last cnt;
10 m = if last state = Idle & r or last state = Busy & e & r then i else
      last m;
    o = m;
15 e = (cnt = 100);
tel

```

List. 4.3: Heptagon node encoding the machine of Figure 4.1.

of memorized values expressed by “last” values of flows. New values for input flows are given at each *execution step*, where equations are then evaluated all together, and values of output flows are updated accordingly.

We give an example Heptagon node in List. 4.3; this node symbolically encodes the Mealy machine given in Figure 4.1 by using “last” flows to memorize its state. One can compose Heptagon nodes using instantiations; *e.g.*, like “(e1, o1) = `inlined` m (r1, i1);” for m.

An *invariant* and *controllable flows* (each taking its value in the Boolean domain $\text{bool} = \{\text{false}, \text{true}\}$) can be specified for Heptagon nodes using *contracts*. When it encounters a node featuring a contract, the BZR compiler involves a symbolic DCS algorithm to automatically produce a *controller* constraining the values of the controllable flows so as to guarantee that the resulting *controlled node* satisfies the invariant. The controllers produced take the form of as many predicates as controllable flows, that implement the following behavior: considering every controllable flow c in turn (according to their order of declaration), the controller tries to assign c to `true` unless this could lead to a potential violation of the desired invariant in subsequent execution steps. Given a Boolean output o , a contract enforcing that o holds using controllable flows $c1$ and $c2$ for a node is declared as “`contract enforce o with (c1, c2: bool)`”.

4.2.3 Variables & Further Notations

In Verilog terms, a set of variables V represents wires and outputs of registers; equivalently in Heptagon terms, variables in V represent flows, including state ones (“last” flows). \mathcal{P}_V

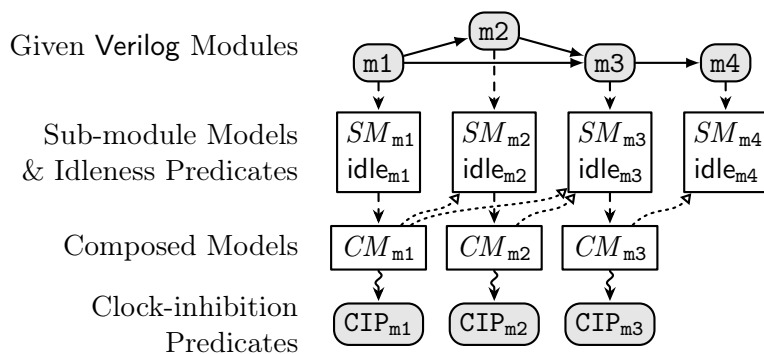


Figure 4.2: Example Verilog module instantiation graph, associated models, and resulting CIPs. Arrows \rightarrow (resp. \dashrightarrow) represents Verilog sub-module (resp. Heptagon node) instantiation relations. In turn, \dashrightarrow (resp. \rightsquigarrow) denotes modeling (resp. symbolic DCS) steps.

is the set of propositional predicates expressed using variables in V . Given an instantiation M_i of a Verilog module M and a set of variables V_M pertained to M , we denote by V_{M_i} the set of variables substituted according to the instantiation M_i . By extension, we write V_{Sub_M} to denote $\uplus_{M_i \in Sub_M} V_{M_i}$, where \uplus is the disjoint union.

4.3 Computing CGLs using Symbolic DCS

We now describe our technique for computing CGLs. We give an overview of the modeling principles and how we eventually integrate the resulting CGLs into the original circuit. We then detail the models and results we obtain from our example.

4.3.1 Overview of the Modeling Technique

Our translation algorithm produces two families of models (represented using Heptagon nodes) that each fit two distinct purposes:

SM the first family of models, called *Sub-module Models*, aims at representing generic sub-modules (*i.e.*, not yet instantiated). They model their behavior using one internal state flow per marked register, and abstract away any sub-module instantiation;

CM the second kind of models, referred to as *Composed Models*, is derived from the Sub-module Models of every Verilog super-module M . CMs instantiate Sub-module Models, and encode the computation of CIPs as symbolic DCS problems.

Our algorithm for computing CIPs works by visiting every Verilog module in the instantiation graph according to an inverse topological order. Every module M is first translated into a Sub-module Model SM_M , accompanied with an *idleness predicate* $idle_M$ that expresses a condition on which the registers' values of any instance of M do not change. Then, every node SM_M of a super-module M is further transformed into a Composed Model CM_M that instantiates Sub-module models. Each node CM_M features a contract that involves control objectives (*i.e.*, at least an invariant involving idleness predicates of sub-module instances) and controllable flows that represent the CIPs of each sub-module instantiated by M : the enforcement of this contract by using a symbolic DCS algorithm results in correct CIPs. We sketch this process using an example circuit specification in Figure 4.2. In this example, three symbolic DCS problems are solved, leading to as many sets of CIPs. Note that SM_{m1} and $idle_{m1}$ are never instantiated: SM_{m1} is only used to derive CM_{m1} .

During the modeling process, one CGL-enabled Verilog module mi' is derived from the each original one mi . Each CIP_{mi} is eventually translated into Verilog code, and then integrated into mi' to derive a clock-gated Verilog design.

Tackling Complexity Issues: Consider a Verilog module M with sub-module instantiations Sub_M , and assume a perfect knowledge of the values of all its input wires I_M , its registers R_M , and the registers of all its direct and indirect sub-module instances. The optimal CIP for each of its sub-module instances $mi \in Sub_M$ is $\eta_{mi}^{optim} \in \mathcal{P}_{I_M \uplus R_M \uplus I_{Sub_M} \uplus R_{Sub_M^*}}$, where Sub_M^* denotes every direct and indirect sub-module instances within M . In principle, one can then build the CGL computing a value for η_{mi}^{optim} at each clock cycle, and use it to inhibit the clock of mi within M . However, the size of today's circuit designs make the exact computation of optimal CIPs generally intractable. To tackle this problem, we compute under-approximations of CIPs by: (i) using a layered approach, where CIPs are computed separately for each super-module and only their direct sub-module instances are taken into account; and (ii) devising a parametrized abstraction technique.

Marking Variables: To drive the abstractions, we parametrize our algorithm with a set of variables to be taken into account when modeling the circuits. This key aspect of our approach allows designers to exploit the knowledge they have on their designs. In particular, the usual distinction between command parts and operational parts of hardware circuits permits a quick identification of registers and wires that are relevant for the computation of CIPs that would otherwise be hard to compute. Referring to our example Verilog module m in List. 4.1, one can observe that the computations on the input data given using wire

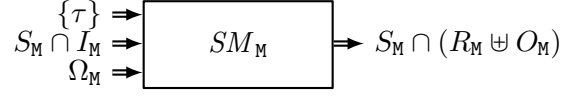
array `i` and output using wire array `o`, are driven by the values held in registers `state` and `cnt`, plus input wire `r`. The output wire `e` is also relevant *w.r.t.* the behaviors of any circuit instantiating `m` as it indicates the termination of its computations. Regarding module `main` of List. 4.2, relevant wires and registers include `r`, `mode`, `e`, `error`, `cfg`, `wait_m1`, and `wait_m2`.

Marked wires and registers shall be specified as a union S of sets S_M of variables pertained to a Verilog module M instantiated in the circuit. Note that our modeling algorithm is sound *w.r.t.* the set of marked variables, meaning that, although some sets S give better results than others (*e.g.*, in terms of dynamic power savings), it always produces functionally equivalent results. In the worst case ($S_M = \emptyset$), the resulting model for M symbolically describes a single-state automaton, and it is only likely to result in a module that is never considered idle.

Abstracting behaviors of the Verilog modules leads to potentially non-deterministic models. Consider for instance an explicit automaton with an input wire i and two transitions from the same source and distinct destinations, respectively guarded with i and $\neg i$; abstracting away i would lead to a non-deterministic automaton. To still construct Heptagon nodes (that are deterministic by definition), we automatically generate *oracles* to replace sub-expressions whose values are abstracted away, thereby explicitly modeling the non-determinism.

Introducing Oracles: Given an expression e on any set of variables, the *oracle* ω_e is an interpreted input that proxies e . In particular, ω_e takes its values in the domain of e (*e.g.*, the Booleans if e is a predicate), and can thus be used to model behaviors where e itself is abstracted away and can non-deterministically take any value in its domain. Every knowledge about the modeled behaviors is not lost however. Indeed, assuming that e and e' admit the same canonical representation e'' , every occurrence of e and e' can be replaced with the same oracle $\omega_{e''}$, and the equality of valuations for e and e' can still be represented. For instance, given the expression $x + y$, where x and y are Integer variables, the oracle ω_{x+y} can non-deterministically take any Integer value: the addition operation, x , and y are abstracted away, replaced by some undetermined Integer. Additionally, if $y + 1 + x - 1$ admits the same canonical representation as $x + y$, then it can also be modeled with ω_{x+y} .

When constructing SM_M or CM_M from a Verilog module M , we introduce a set of oracles Ω_M to handle expressions within M that involve non-marked wires and registers (not belonging to S_M): *i.e.*, the actual values of these expressions are abstracted away in the resulting models. Yet, our goal is to actually generate circuits that encode CGLs, and that can thus


 Figure 4.3: Interface of a Sub-module Model SM_M .

be used to inhibit the clock of instances of M : the actual values of marked registers and abstracted expressions computed within M are hence required when translating the resulting CIPs into Verilog code. As a result, while oracles Ω_M are *inputs* to the models of M , we also build a *CGL-enabled* version M' that features one additional *output wire* per oracle in Ω_M that does not represent expressions only involving inputs of M ; these additional wires carry the actual values of the corresponding expressions, and are thus used to feed the CGL of super-modules. Additional output wires of M' also carry the value of marked registers belonging to S_M .

Bottom-up Clock-inhibition Allowance: Of course, the clock of any instance of M also drives sub-module instances $mi \in Sub_M$. As a result the clock of M should not be inhibited whenever any sub-module instance mi must not be inhibited. However, SM_M does not model the behavior of any of its sub-module instances. Hence, we choose to add a *bottom-up clock-inhibition allowance* output wire $allow\eta_M$ to M' , built as the conjunction of every CIPs of sub-modules instantiated by M , or 1 if there is none.

Resulting CGLs: Eventually, the CGL to be integrated within a Verilog super-module M consists of one CIP $\tilde{\eta}_{mi} \in \mathcal{P}_{S_M \uplus \Omega_M \uplus S_{Sub_M} \uplus \Omega_{Sub_M} \uplus \biguplus_{mi \in Sub_M} allow\eta_{mi}}$ per sub-module instance $mi \in Sub_M$. $\tilde{\eta}_{mi}$ under-approximates the condition upon which the clock of sub-module instance mi can be inhibited: *i.e.*, it is such that $\tilde{\eta}'_{mi} \Rightarrow \eta_{mi}^{optim}$, $\tilde{\eta}'_{mi}$ being $\tilde{\eta}_{mi}$ where every oracle ω_e is substituted by e . We rely on a symbolic DCS algorithm to compute such CIPs.

4.3.2 Building Sub-module Models & Idleness Predicates

We outline in Figure 4.3 the interface of a Sub-module Model SM_M for a Verilog module M . Its inputs include (i) an enable flow τ ; (ii) flows mirroring marked input wires selected for this module ($S_M \cap I_M$); and (iii) a set of input oracles Ω_M that are used to model undetermined behaviors of instances of M . The outputs of SM_M comprise flows mirroring the marked registers and output wires selected for M ($S_M \cap (R_M \uplus O_M)$).

We further associate each Sub-module Model SM_M with an idleness predicate $idle_M \in \mathcal{P}_{S_M \uplus \Omega_M}$, that under-approximates the condition on which the registers' values of any instance

```

1 type state_t = Idle | Busy
node SMm (τ, r, ωcnt==100: bool)
returns (last state: state_t = Idle; e: bool)
let
5  state = if not τ then last state else
           if last state = Idle & r then Busy else
             if last state = Busy & ωcnt==100 & not r then Idle else
               last state;
           e = ωcnt==100;
10 tel

```

List. 4.4: Heptagon node SM_m , obtained from the Verilog module m of List. 4.1 using marked variables $S_m = \{\text{state}, r, e\}$.

```

1 module m' (input clk, input r, input [7:0] i,
            output [7:0] o, output e,
            // additional outputs:
            output state', output ωcnt==100, output allowm);
5  ...
  assign state' = state;
  assign ωcnt==100 = (cnt == 100);
  assign allowm = 1; // no clock-gated sub-module instance
endmodule

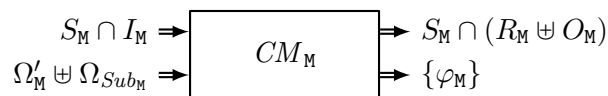
```

List. 4.5: Excerpts of the CGL-enabled Verilog module m' obtained from the module m of List. 4.1.

of M do not change; *i.e.*, given values for marked input wires, marked internal registers, and for the oracles, if idle_M holds then any assignment to any (both marked and non-marked) internal and output registers of M would not change the value it memorizes.

One can easily build the Heptagon node SM_M and the associated condition idle_M from a Verilog module M where every internal wire is substituted with the expression it is assigned to. A traversal of clock-triggered assignments to marked registers clocked using clk allows the construction of cascading conditional statements for the assignments to flows encoding the state within SM_M (“last” flows). A similar traversal can be used to construct idle_M as the conjunction of the negation of every guard leading to the assignment of a register. An efficient introduction of oracles can be performed by building a canonical representation of every expression using (basic and/or multi-terminal) binary decision diagrams for instance. Then, every canonical expression e that involves a non-marked variable becomes an oracles $\omega_e \in \Omega_M$.

Note that the constructions above do not necessarily traverse every expression, register or wire of a module declaration, hence only a limited number of oracles might be required even for large modules. This claim is supported by the application of our technique on a


 Figure 4.4: Interface of a Composed Model CM_M .

realistic case study detailed in Section 4.4.

When applied to the Verilog module m of List. 4.1 with marked variables $S_m = \{\text{state}, r, e\}$, our construction technique for Sub-module Models builds the **Heptagon** node of List. 4.4. The assignment to e on line 9 corresponds to the assignment on line 8 in List. 4.1: the value of “`cnt == 100`” is abstracted away using an oracle as `cnt` does not belong to S_m . In turn, the predicate that describes the idleness condition of m is $\text{idle}_m = (\text{state} = \text{Idle} \ \& \ \text{not } r)$. Finally, we show in List. 4.5 the additions to module m that are necessary to construct the corresponding CGL-enabled Verilog module m' . m' features three additional output wires (one for as many oracles in Ω_m , one per variable in $S_m \cap R_m$, plus $\text{allow}\eta_m$). The bottom-up clock-inhibition allowance output $\text{allow}\eta_m$ is assigned to 1 as no sub-module instantiation exists within m to prevent the clock of m from being inhibited.

4.3.3 Building Composed Module Models

A Composed Model CM_M is derived from SM_M by taking sub-module instantiations mi into account, and formulating the computation of $\tilde{\eta}_{mi}$'s as a symbolic DCS problem. Basically, the instantiation of a sub-module m by M translates within CM_M into the instantiation (say, SM_{mi}) of the **Heptagon** node SM_m . The input τ of each **Heptagon** node instantiation SM_{mi} is assigned to the negation of the corresponding CIP “`not $\tilde{\eta}_{mi}$` ”. CIPs, in turn, are the controllable flows as they represent the CGL outputs. (The input flow τ modeling the clocks in Sub-module Models is no longer required, and can be substituted with `true` everywhere else in CM_M .) Further, we build Ω_{mi} and idle_{mi} according to the appropriate renaming in Ω_m and substitutions in idle_m . Note that, with the additional output flows from Sub-module Models, some oracles in Ω_M may represent expressions that are now fully determined. A substitution of such oracles with their respective expressions is thus necessary in CM_M so that marked outputs of sub-modules are taken into account. Let Ω'_M be Ω_M pruned from the latter oracles. At last, the invariant to enforce by control φ_M states that a CIP $\tilde{\eta}_{mi}$ for a sub-module instance mi should not hold unless idle_{mi} holds:

$$\varphi_M = \bigwedge_{mi \in Sub_M} (\tilde{\eta}_{mi} \Rightarrow \text{idle}_{mi}).$$

```

1 type cfg_t = LQ | HQ
node CMmain (ωr, ωm1_cnt==100, ωm2_cnt==100: bool; ωmode: cfg_t)
returns (last wait_m1: bool = false; last wait_m2: bool = false;
        last cfg: cfg_t = LQ; φM: bool)
5 contract enforce φM with (ñm1, ñm2: bool)
var m1_state, m2_state: m_state_t; e1, e2: bool;
let
  (m1_state, e1) = inlined SMm (not ñm1, ωr & not last wait_m1 & not last wait_m2, ωm1_cnt==100);
  (m2_state, e2) = inlined SMm (not ñm2, e1 & not last wait_m2 & last cfg = HQ, ωm2_cnt==100);
10 cfg = if ωr & not last wait_m1 & not last wait_m2 then ωmode else
        last cfg;
wait_m1 = if ωr & not last wait_m1 & not last wait_m2 then true else
          if e1 then false else
            last wait_m1;
15 wait_m2 = if last cfg = HQ & e1 & not last wait_m2 then true else
            if e2 then false else
              last wait_m2;
φM = (ñm1 => (m1_state = Idle & not (ωr & not last wait_m1 & not last wait_m2)))
      & (ñm2 => (m2_state = Idle & not (e1 & not last wait_m2 & last cfg = HQ)));
20 tel

```

List. 4.6: Heptagon node CM_{main} obtained from the Verilog module `main` of List. 4.2 using marked variables $S_{\text{main}} = \{\text{cfg}, \text{wait_m1}, \text{wait_m2}\}$ and S_{m} as used in List. 4.4.

We sketch the interface of a Heptagon node CM_{M} in Figure 4.4; note that it also admits as inputs the oracles of every instantiated sub-module. We give in List. 4.6 the result we obtain for CM_{main} .

4.3.4 Computing & Integrating the CGLs

As stated in Section 4.2.2, the compilation of a Heptagon node that features a contract (as Composed Models do), involves a symbolic DCS computation step that produces a controller made of one predicate per controllable flow (*i.e.*, CIPs). By virtue of the semantics assigned to such flows by the Heptagon compiler (*i.e.*, assigning them to `true` whenever possible), one can eventually translate the controller into some Verilog code encoding a CGL that inhibits the clock of sub-module instances whenever possible. We show in List. 4.7 excerpts of the end result that we obtain for our running example. The assignments to registers¹ holding $\tilde{\eta}_{m1}$ and $\tilde{\eta}_{m2}$ are clocked using `clk`: their respective input value consists in the conjunction between their respective bottom-up clock-inhibition allowance (`allow η_{m1}` and `allow η_{m2}`), and their respective CIPs as computed by using symbolic DCS. The clocks of sub-module instances `m1` and `m2` are now filtered according to $\tilde{\eta}_{m1}$ and $\tilde{\eta}_{m2}$. As a side note, remark that ω_{e1} and ω_{e2} are output wires of `main`' since these outputs of sub-module

¹Although using wires for CIPs would seem sufficient from a functional point of view, registers are required to avoid glitches [11].

```

1 module main' (input clk, input r, input mode, input [7:0] i,
  output [7:0] o, output e, output error,
  // additional outputs:
  output wait_m1', output wait_m2', output cfg',
5  output  $\omega_{e1}$ , output  $\omega_{e2}$ , output allow $\eta_{main}$ );
  reg  $\tilde{\eta}_{m1}$ ,  $\tilde{\eta}_{m2}$ ;
  always @(posedge clk) begin
     $\tilde{\eta}_{m1}$  <= allow $\eta_{m1}$  & (m1_state == idle) & (wait_m1 | wait_m2 | ~r);
     $\tilde{\eta}_{m2}$  <= allow $\eta_{m2}$  & (m2_state == idle) & ((cfg == LQ) | wait_m2 | ~ $\omega_{m1\_cnt==100}$ );
10  end
  wire allow $\eta_{m1}$ , m1_state,  $\omega_{m1\_cnt==100}$ ;
  m' m1 (.clk(clk & ~ $\tilde{\eta}_{m1}$ ), .allow $\eta_m$ (allow $\eta_{m1}$ ), ...
    .state'(m1_state), . $\omega_{cnt==100}$ ( $\omega_{m1\_cnt==100}$ ));
  wire allow $\eta_{m2}$ , m2_state,  $\omega_{m2\_cnt==100}$ ;
15  m' m2 (.clk(clk & ~ $\tilde{\eta}_{m2}$ ), .allow $\eta_m$ (allow $\eta_{m2}$ ), ...
    .state'(m2_state), . $\omega_{cnt==100}$ ( $\omega_{m2\_cnt==100}$ ));
    ...
    // assignments to outputs to be CGL-enabled:
    assign wait_m1' = wait_m1, wait_m2' = wait_m2;
20  assign cfg' = cfg; assign  $\omega_{e1}$  = e1,  $\omega_{e2}$  = e2;
    assign allow $\eta_{main}$  =  $\tilde{\eta}_{m1}$  &  $\tilde{\eta}_{m2}$ ; // <- clock-inhibition allowance
  endmodule

```

List. 4.7: Excerpts of resulting Verilog module obtained from module main of List. 4.2.

instances are required to construct SM_{main} and $idle_{main}$. However, ω_r and ω_{mode} are not part of these outputs as they represent input wires only.

4.4 Application on a Case Study

To experimentally validate our approach, we have manually applied our clock-gating synthesis technique on a real hardware component. To this end, we chose to use a Reed-Solomon (RS) decoder². RS codes are a group of error-correcting codes, that have wide range of applications in digital communications and storage [91]. Basically, this decoder takes coded words of 204 bytes as inputs, and outputs decoded words of 188 bytes. The original decoder is made of 23 modules that build up a circuit with around 52,000 gates and 3,000 registers (flip-flops). Among them, 6 modules drive the operations to be performed on the data: they feature two easily identifiable families of wires and registers that we marked for our modeling: (i) register arrays named **state** or **step**, that take their values into discrete domains made of a few constants (similarly to **state** in List. 4.1); these registers are typically used to encode some command automaton that drives operations on data; and (ii) input and output wires named ***_ready** or ***_done**, that signal end of computations. We then produced a

²https://opencores.org/project,reed_solomon_decoder.

	Original	CGL-enabled	Saving (%)
Cyclone IV (100MHz)	69.98	58.55	16.33
Stratix III (100MHz)	86.99	74.16	14.75
HardCopy IV (100MHz)	15.48	13.84	10.59
Cyclone IV (1GHz)	699.80	585.48	16.34
Stratix III (1GHz)	869.94	741.60	14.75
HardCopy IV (1GHz)	154.85	138.46	10.58

Table 4.1: Estimated mean power dissipation (in mW) of original and resulting RS decoders.

“CGL-enabled” circuit including CIPs produced using symbolic DCS.

To experimentally assess the functional correctness and compare the respective dynamic power dissipation of each of the designs at hand (original and CGL-enabled), we first performed logic synthesis on both of them using the Altera Quartus synthesizer. We then used the Altera ModelSim simulation tool to perform functional simulations using the same benchmark (provided with the source code of the original decoder) for the two circuits, and checked that the resulting traces were strictly equivalent. To assess actual dynamic power savings, we have carried out estimations of mean power dissipation on simulations of the benchmarks, for various target technologies and main clock frequencies as these factors have a great impact on dynamic power. The Altera PowerPlay Power Analyzer tool offers several pre-configured target technologies, among which we chose the Cyclone IV (dedicated to low-power FPGA designs), the Stratix III (for high-performance FPGA designs), and the HardCopy IV (ASICs) families. We also setup the main clock frequencies to be either 100Mz or 1GHz.

We show the resulting estimations of power dissipation and respective dynamic power savings in Table 4.1. We consider that these results are promising when put in perspective with the relative simplicity of our approach. Indeed, we generated effective CIPs by imposing invariants only, as our technique do not even incorporate control techniques towards any sort of optimization yet.

Clock gating is the most used technique for reducing the dynamic power consumption in synchronous circuits, where the aim is generally switch-off the clock of a component when it is idle. The subject has been studied for a single register or group of registers, in several works. However, it has not been applied for module level to the best of our knowledge. The biggest advantage of this approach is that only one clock-gating cell is used per module, unlike the other approaches. Thus, our approach requires only a small

piece of extra logic (*i.e.*, small area usage and higher power saving, compared with other approaches). Furthermore, our approach relies on formal control techniques. Compared to other existing techniques for applying clock-gating, the main advantage of it provides a synthesisable controller with formal correctness. We applied our approach on a realistic case study, where today's RTL designs are generally composed as a hierarchical structure. Thus, we show that it can be applied for any hierarchical design. Although it is not available to calculate the best available theoretical estimation of a given design, our results (module based) are promising compared with the register based approaches in the survey paper [87] (roughly %25 power saving).

4.5 Summary and Discussion

Low-Power Chip Design: Several families of design methods permit the (semi-)automated use of power-saving technologies: they can be integrated into high-level (*aka* system level) or RTL descriptions, or further down the implementation process, during “synthesis” (*i.e.*, translation of RTL descriptions into a network of gates and wires), or placement and routing steps. Nonetheless, [28] found that considering higher levels of abstraction generally leads to more power savings. Designers most commonly rely on the RTL code itself to implement clock-gating, yet a few approaches automatically generate RTL code with integrated clock gating from higher-level descriptions. Among them, [1] developed an environment for high-level design with their own procedural language. [2] also provide a solution to design circuits directly using the C language. In these approaches, designers are responsible for the selection of gated components.

One can distinguish three classes of RTL clock-gating algorithms based on the hierarchical level at which they consider the circuit: combinatorial or sequential ones focus on individual registers [100, 67], system- or module-level [19] focus on clock-gating whole modules or blocks of clock-triggered assignments. We rely on the latter level for the layered abstractions that it allows.

Several commercial and academic tools already target automated clock-gating from RTL code. [89] developed an algorithm that automatically inserts CGL into RTL descriptions of circuits. They focus on the exact computation of idleness conditions for individual registers within a single module, hence their solution suffers from scalability issues. To partially overcome these issues, [7] suggest an algorithm that automatically tries to approximate idleness conditions. Later, [24] used a control-based adaptive clock-gating algorithm to shut

down IP cores based on given explicit finite-state models. In an approach that targets the conditional activation of individual hardware components using their “enable” signal (an approach similar to clock-gating), [12] tried to detect idleness conditions by using explicit finite-state machines. At last, [89] exploited conditional statements and case structures within blocks of clock-triggered assignments in HDLs to determine such conditions. Our approach draws from the latter ones in the sense that we also operate at the HDL level, and build symbolic finite-state machines from conditional clock-triggered assignments. We additionally bring layered and semi-automated abstractions for the sake of scalability.

Applying DCS for Low-power Hardware Design: Few hardware design techniques rely on DCS for saving power. [88] present a high-level design flow for reconfigurable FPGA-based System-on-Chip (SoC); they model potential reconfiguration behaviors and manually derive a “controller” that automatically takes reconfiguration decisions. Later, [50] and [5, 6] were among the firsts to apply DCS algorithms for reconfiguration management in SoC design. Doing so, they could automate the generation of controllers, thereby exploiting the formal correctness and guarantees that DCS techniques provide. In particular, [5, 6] model the applications’ behaviors and the needed resources (area in hardware—*i.e.*, regions of the FPGA) using explicit automata; they then use a symbolic DCS algorithm to automatically compute a reconfiguration manager for the system.

In this work, we have described a systematic approach for computing the CGL of synchronous circuits described using the Verilog hardware description language. This approach exercises symbolic DCS algorithms by means of a semi-automated modeling in Heptagon of each individual Verilog modules. We have demonstrated its principles using an example, and have reported on its manual application on a realistic case study.

The next steps involve a formalization of our modeling algorithm to validate its correctness, along with the development of an implementation in a tool. Our approach can also be extended to compute CGLs for individual registers within modules. Although we exercised our technique to implement clock-gating as it currently offers the best trade-off between extra occupied circuit area and power savings [58], it is also applicable to other low-power design mechanisms such as power-gating (especially for computation-intensive modules that can be shut down for long periods of time). Also, the abstractions induced by our modeling approach make it a good candidate for constructing models suitable for the application of control algorithms that do not scale up to exact whole-circuit models. In particular, the adaptation of our algorithm for the computation of “suspendability” predicates would allow to suspend the computations of sub-module instances by control. Combined with the

recent advances in control algorithms for symbolic infinite-state systems with applications to quantitative models [14, 15], our framework could permit the application of optimal control techniques towards the minimization of peak dynamic power or energy dissipation over several clock cycles. Similarly, incorporating stochastic models (*e.g.*, inferred from simulation traces) would provide interesting cases for developing new optimal control algorithms targeting such goals. Also, automatically identifying “good” sets of marked variables constitutes an interesting challenge. At last, the support of black-box sub-modules with simple user-provided models can also be considered.

Chapter 5

Power-Aware Scheduling of Data-Flow Hardware Circuits with Symbolic Control

In this chapter, we advance the framework introduced in the previous chapter. We offer more abstract models for data-flow hardware circuits, where we achieve the power efficiency by tackling another hardware design problem, scheduling policy. We also add a new set of control objectives to implement a scheduling policy, so we do not provide power saving only in the idleness condition of a sub-circuit, but also the scheduled inactive period of sub-circuits. Our approach relies on formal control techniques, where the goal is to compute a strategy that can be used to drive a given model so that it satisfies a set of control objectives. More specifically, we give an algorithm that derives abstract behavioral models directly in a symbolic form from original designs described at Register-transfer Level using a Hardware Description Language, and for formulating suitable scheduling constraints and power-efficiency objectives. We show how a resulting strategy can be translated into a piece of synchronous circuit that, when paired with the original design, ensures the aforementioned objectives. We illustrate and validate our approach experimentally using various hardware designs and objectives. Our paper¹, which includes the contents and results of this chapter, have been under the submission and review.

¹Author: Mete Özbaltan and Nicolas Berthier.

5.1 Introduction

High-level models are often required to reason on synchronous circuit designs, as well as apply scalable techniques to translate them into equivalent designs that meet various performance goals such as energy-efficiency [106]. Among these models, the family of *data-flow process networks* [64] sees circuits as actors that communicate tokens through communication channels (*FIFOs*) and react according to specific firing rules. *Kahn Process Networks* (KPN) [49] are a sub-class of such models where channels are considered unbounded, and processes are fired whenever there are enough tokens in its input channels. A consequence of this property is that writes can be considered *non-blocking*—*i.e.*, a process is never blocked when it writes to a channel—, whereas reads to empty channels are *blocking*. KPNs can be used to describe systems where the amount of data produced and consumed by a process is not statically determined.

We consider designs that implement KPNs, and where each process implementation is described at Register-Transfer Level in a Hardware Description Language (HDL) such as Verilog. We advance a (mostly-)automated procedure that translates such designs into functionally equivalent ones that in addition enjoy *power-awareness* guarantees, in a bid to reduce their *dynamic power dissipation*. To this end, we compute a *strategy* that implements a *power-aware scheduling policy* by selectively clock-gating [19] each process. Notice we do not seek to reduce the total *energy* consumed (*i.e.*, power integrated over the total computation time). Rather, we seek to reduce the *instantaneous power* (possibly integrated over a small time window), as this kind of power-efficiency policy usually has a positive impact on the lifetime of battery-powered devices [84], and also provides a means to limit chip temperature. In effect, we allow ourselves to degrade timing performance to achieve such goals.

Our approach relies on the construction of *abstract symbolic models* of the designs, and employs *discrete control* techniques to compute a piece of hardware circuit that implements some power-aware scheduling policies specified *in a declarative way*. This piece of circuit can eventually be used to selectively filter the clocks of the processes involved.

Outline: The remainder of the chapter is organized as follows: Section 5.2 gives necessary background on symbolic models and control. Section 5.3 presents our approach, which is experimentally evaluated in Section 5.4. At last, Section 5.5 conclude with the summary and discussion.

5.2 Discrete Control with Symbolic Systems

5.2.1 Symbolic Notations

The symbolic systems we consider are built upon a set of *symbols* \mathcal{S} , each associated with a domain of definition $\mathbb{D} \in \mathcal{D}$ according to the mapping $\text{Dom}: \mathcal{S} \rightarrow \mathcal{D}$. \mathcal{D} comprises at least the Boolean domain $\mathbb{B} \stackrel{\text{def}}{=} \{\text{tt}, \text{ff}\}$, and numerical domains $\mathcal{D}_{\text{num}} \subseteq \mathcal{D}$ such as the Integers (\mathbb{Z}) and Rationals (\mathbb{Q}), and \mathcal{S} notably comprises all *constants* used to define domains in \mathcal{D} (e.g., $\{\text{tt}, \text{ff}, 0, -\frac{1}{2} \dots\}$). Every domain in \mathcal{D} is equipped with *equality*. \mathcal{D}_{fin} denotes the set of all finite domains in \mathcal{D} .

Given a domain $\mathbb{D} \in \mathcal{D}$, the set $\mathcal{X}_{\mathbb{D}}$ of all \mathbb{D} -valued *symbolic expressions* comprises all formulae $\psi_{\mathbb{D}}$ that can be generated according to the following grammar:

$$\begin{aligned}
 \psi_{\mathbb{D}} &::= s \mid \text{if } \psi_{\mathbb{B}} \text{ then } \psi_{\mathbb{D}} \text{ else } \psi_{\mathbb{D}} && (\mathbb{D} \in \mathcal{D}, s \in \mathcal{S}, \text{Dom}(s) = \mathbb{D}) \\
 \psi_{\mathbb{B}} &::= \neg \psi_{\mathbb{B}} \mid \psi_{\mathbb{B}} \vee \psi_{\mathbb{B}} \mid \psi_{\mathbb{B}} \wedge \psi_{\mathbb{B}} \mid \psi_{\mathbb{B}} \Rightarrow \psi_{\mathbb{B}} \\
 &\quad \mid \psi_{\mathbb{D}} = \psi_{\mathbb{D}} \mid \psi_{\mathbb{D}} \neq \psi_{\mathbb{D}} && (\mathbb{D} \in \mathcal{D}) \\
 &\quad \mid \psi_{\mathcal{N}} \bowtie \psi_{\mathcal{N}} && (\mathcal{N} \in \mathcal{D}_{\text{num}}, \bowtie \in \{<, \leq\}) \\
 \psi_{\mathcal{N}} &::= c \psi_{\mathcal{N}} \mid \psi_{\mathcal{N}} \boxtimes \psi_{\mathcal{N}} && (\mathcal{N} \in \mathcal{D}_{\text{num}}, c \in \mathcal{N}, \boxtimes \in \{+, -\}).
 \end{aligned}$$

Note that rule $\psi_{\mathbb{D}}$ is polymorphic (*i.e.*, generic) in the domain \mathbb{D} , and restricts conditional constructs to cases where the two rightmost expressions are of the same type. Also, this grammar features the following constructs that are available as syntactic sugar to ease readability: \neg , \vee , \wedge and \Rightarrow are usual logical connectives that can be expressed using conditional constructs, and $\varphi - \psi$ is equivalent to $\varphi + -1\psi$. Informally, the rules $\psi_{\mathcal{N}}$ and $\psi_{\mathbb{B}}$ respectively produce *guarded linear arithmetic expressions* and *propositional predicates with equality and linear (in)equalities*.

Non-constant symbols are *variables*, and a *valuation* $\nu \in \text{Val}(V)$ for each variable in $V \subseteq \mathcal{S}$ maps every variable x from V to $\text{Dom}(x)$. We also note $\text{Dom}(\nu)$ the set of variables bound by a valuation ν , and compose valuations for disjoint sets of variables with \uplus . The *support* of e , denoted $\text{Support}(e)$, is the set of variables that appear anywhere in its constructs. We shall index sets of expressions with the variables that make up the smallest super-set of their support when such a precision appears relevant for our exposition: e.g., $\mathcal{X}_{\mathbb{D}, V} \stackrel{\text{def}}{=} \{e \in \mathcal{X}_{\mathbb{D}} \mid \text{Support}(e) \subseteq V\}$; when domains appear irrelevant, however, we abbreviate the set of all symbolic expressions as $\mathcal{X} \stackrel{\text{def}}{=} \bigcup_{\mathbb{D} \in \mathcal{D}} \mathcal{X}_{\mathbb{D}}$.

$\mathcal{P} \stackrel{\text{def}}{=} \mathcal{X}_{\mathbb{B}}$ is the set of all *propositional predicates* with (in-)equalities; \mathcal{P} is closed under elimination of variables defined on finite domains. Further, for every domain $\mathbb{D} \in \mathcal{D}$, $\mathcal{X}_{\mathbb{D}}$ is also closed under substitution of any symbolic expression that belongs to $\mathcal{X}_{\mathbb{D}'}$ for any variable v s.t. $\text{Dom}(v) = \mathbb{D}'$. We generalize to multiple variables and denote with $e[\mu]$ such a substitution in $e \in \mathcal{X}$ according to a mapping μ from a set of variables to symbolic expressions of the same domain of definition. We use the variable assignment denotation $x := e_x$ to incrementally construct such a mapping in Section 6.3².

Given $e \in \mathcal{X}_{\mathbb{D},V}$ and a valuation ν s.t. $\text{Dom}(\nu) \supseteq V$, the value of $\llbracket e \rrbracket \nu$ belongs to \mathbb{D} and can be computed as follows:

- $\llbracket s \rrbracket \nu = \nu(s)$, if $s \in V$, s otherwise;
- $\llbracket \text{if } \varphi \text{ then } \psi \text{ else } \psi' \rrbracket \nu = \begin{cases} \llbracket \psi \rrbracket \nu & \text{if } \llbracket \varphi \rrbracket \nu = \text{tt}, \\ \llbracket \psi' \rrbracket \nu & \text{otherwise;} \end{cases}$
- $\llbracket \psi = \psi' \rrbracket \nu = \text{tt}$ if $\llbracket \psi \rrbracket \nu = \llbracket \psi' \rrbracket \nu$, ff otherwise;
- $\llbracket \psi \bowtie \psi' \rrbracket \nu = \text{tt}$ if $\llbracket \psi \rrbracket \nu \bowtie \llbracket \psi' \rrbracket \nu$, $\bowtie \in \{<, \leq\}$, ff otherwise;
- $\llbracket c\psi \rrbracket \nu = c \times \llbracket \psi \rrbracket \nu$;
- $\llbracket \psi + \psi' \rrbracket \nu = \llbracket \psi \rrbracket \nu + \llbracket \psi' \rrbracket \nu$.

5.2.2 Symbolic Systems

Symbolic systems are made of disjoint sets of *state* and *input* variables, respectively denoted X and I . The values associated to state variables are initialized with constants, and evolve according to a discrete step (*lock-step*) semantics very similar to that of synchronous circuits: *discrete evolutions* are defined using a mapping T with one *assignment* $x := e_x$ per state variable $x \in X$, where e_x is a $\text{Dom}(x)$ -valued symbolic expression that determines the value to be memorized by variable x based on the current valuations for state and input variables, at each *tick* of an *implicit basic clock*. The systems we consider may additionally be equipped with an *assertion* A , which is a predicate that belongs to $\mathcal{P}_{X \cup I}$: A encodes *assumptions* on the possible valuations for input variables based on that of state variables.

Such a system induces a state machine whose *state-space* consists of all possible valuations for every variable in X , and whose transitions are encoded by the discrete evolutions T and assertion A . This machine is a *finite-state machine* (FSM) iff the domain of definition of every variable in X is finite.

²Throughout the thesis, and unlike $:=$, $s \stackrel{\Delta}{=} e$ denotes the classical formal definition of a left-hand side symbol s with a right-hand side expression e : s can be seen as a placeholder for expression e everywhere in the thesis. Alternatively, we use $\stackrel{\text{def}}{=}$ when defining structures and algorithms.

5.2.3 Symbolic Control

Solving *symbolic control problems* on such systems can be seen as solving a game where, at each tick, one player (the environment) gives a value for a fixed portion of the input variables, *then* the other player (the *controller*) assigns values to every other input variable, *and then* the game evolves into a subsequent state according to the discrete evolutions and the inputs given by the players. The *control objectives* assigned to the second player are expressed as logic formulas that involve state and/or input variables, and the solution of the control problem consists in a *strategy* that this player can follow to win the game by fulfilling all its objectives. The input variables assigned by the first (resp. second) player are said *non-controllable* (resp. *controllable*); we denote these sets of variables U and C , respectively. Note that at each turn, the assertion A *restricts* the choices available for the players to valuations of their respective input variables that *do not falsify* A : in a state q , the first player must pick values u for non-controllable variables such that there exists at least one valuation for controllable variables (*i.e.*, such that the controller can choose c *s.t.* $\llbracket A \rrbracket q \uplus u \uplus c = \text{tt}$). Systems where such a choice for the first player always exists are said *deadlock-free*, and algorithms solving control problems should preserve (or enforce) this property.

The control objectives that we use in this chapter and Chapter 6 are twofold: First, achieving a *safety control objective* consists in enforcing a *safety property*. Such properties can be expressed using some temporal logic like LTL [26]. In our case however, we use the same symbolic constructs as for the system to build *stateful observers* that represent the temporal aspects of the properties we need (*e.g.*, sequence, iteration), and can therefore restrict the safety objective formulas to propositional logic. Second, satisfying *optimal control objectives* consists in minimizing a *cost function*, possibly summed over a sliding window of a given number of ticks. In systems as described in Section 5.2.2, when costs are associated with the *transitions* of the underlying FSM instead of its states, the cost function is a total mapping from state and input valuations into some numerical domain: in symbolic terms, it can therefore be expressed as an expression in $\mathcal{X}_{\mathcal{N}, \mathcal{X} \cup I}$, with $\mathcal{N} \in \mathcal{D}_{\text{num}}$. When both a safety and an optimal control objective are to be enforced, the combined strategy can be obtained by first computing a strategy that ensures the safety objective, and then refining it to satisfy the optimal control objective.

Observe that there does not always exist a strategy that fulfills the desired safety control objectives, and in this case safety control algorithms terminate but produce no output. We

shall see that in this work, the absence of a strategy specifically reveals unrealizable objectives regarding the limitation of dynamic power consumption (*w.r.t.* modeling abstractions).

1) *Strategies as Efficient Sequential Code*: Usually, strategies that are computed by algorithms dedicated to operate on symbolic systems eventually take the form of a *predicate* over state and input variables: *i.e.*, much as the assertion A , they belong to \mathcal{P}_{XUI} . Then, given a valuation for state and non-controllable inputs, a constraint solver needs to be used to find a suitable valuation for controllable inputs that satisfies the predicate. The existence of such a solution is guaranteed by the control algorithm; when this solution is always unique, moreover, the strategy is *deterministic*.

To avoid relying on a constraint solver, a *triangularization* procedure [53] can be used to translate the strategy into a mapping from valuations for state and non-controllable input variables into valuations for controllable input variables, which is basically a combinatorial circuit. This translation operates by using successive variable substitutions and partial evaluations of the predicate strategy. Triangularizing non-deterministic strategies notably requires a total order on solutions, *e.g.*, with a total order on both the controllable input variables and their respective domains of definition: this can be achieved by ordering (prioritizing) the controllable input variables, and assigning them with “default” or “preferred” values.

The triangularized strategy can directly be combined with the original system to form an controlled system that, when fed with values for non-controllable inputs, keeps track of the state of the model and outputs appropriate values for controllable variables to enforce the desired control objectives. This controlled system does not rely on any constraint solver, and can therefore easily be implemented as a efficient piece of sequential code or synchronous circuit.

5.2.4 Tooling and Related Works on Optimal Control

Few works have addressed the problems of enforcing safety control and optimization objectives on the kind of symbolic systems we construct; they mostly derive from the seminal work of [90]. [72] and [77] implemented tools that are suitable for enforcing safety objectives. [15, 16] extended these algorithms to operate on systems where state variables may take their values in infinite numerical domains like Integers or Rationals: they implemented these algorithms as part of the ReaX tool³.

³Available at <https://reatk.gforge.inria.fr/>.

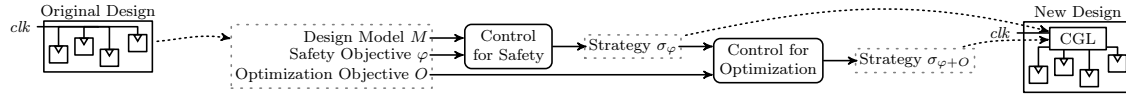


Figure 5.1: Overview of the possible work-flows for computing the power-aware CGL.

In turn [73] and [34] implemented solutions for optimization objectives; these works focus on finite-state systems equipped with cost functions relating to states only, and essentially consist in symbolic adaptations of Bellman’s algorithm for the computation of optimal strategies using dynamic programming [10]: as such, they rely on a given set of *target states* in the underlying FSM, to be reached using a path that incurs an optimal cost.

The algorithm we present in the next chapter alleviates the need for specifying target states to better accommodate the modeling of reactive systems for which the solutions above are inadequate. Instead, we take an alternative approach by focusing on the optimization on all future paths of a given length at once.

5.3 Models and Objectives for Power-aware Scheduling

5.3.1 Overview of the Approach and Contributions

We describe in Figure 5.1 the work-flows offered by our approach: our goal is to automatically construct the *Clock-gating Logic* (CGL) that implements a power-aware scheduler for all processes involved. A *Design Model* M is first built from the original design. M is made of the synchronous parallel composition of, for each process in the design: (i) a *process model*, which is a symbolic behavioral abstraction of the process, constructed from the HDL description of its implementation; (ii) an *idleness predicate* $idle_p$ that must not hold if the value of any register within p is not strictly equivalent before and after the edge of the clock (*i.e.*, it is an under-approximation); and (iii) a symbolic *power expression* P_p that gives an estimated measure of the instantaneous dynamic power consumption of the modeled process based on the state and input symbols of p . Each process p is associated with a *clock-inhibition signal*, $inhibit_p$, which is a *controllable input symbol* that shall hold when the computations of p can be suspended, *i.e.*, its clock can be inhibited.

Then, a *safety objective* φ is built, in the form of a conjunction of propositional formulas involving state symbols of both M plus, possibly, supporting stateful observers specified using additional state symbols. These objectives fall into three categories: *clock-gating constraints* relate each clock-inhibition signal with the model of its process within M ;

scheduling constraints restrict the set of eligible control strategies to those that ensure *progress*, *fairness*, and *absence of starvation*; lastly, *peak power constraints* can be used to specify an upper-bound on the sum of all power expressions upon any tick.

At this stage, a symbolic safety control algorithm is used to compute a strategy σ_φ , which is guaranteed to select values for the controllable inputs that ensure that the safety objective φ are fulfilled. Since clock-inhibition signals belong to the set of controllable inputs, σ_φ and M can be used in combination to form a piece of circuit that filters the individual clock signals for the respective processes. Alternatively, the strategy σ_φ can be improved by using our new symbolic control algorithm that ensures an additional optimization objective O . The cost function that we use to define O basically consists of the sum of power expressions P_p for all processes. The resulting refined strategy $\sigma_{\varphi+O}$ can be used in the same way as σ_φ to filter the clocks of each process.

Other Contributions: We have implemented a set of open-source tools⁴ that helps putting our approach into practice. These tools allow designers to: (1) construct the model M and the associated safety and optimization objectives from an HDL description of the original design; (2) compute suitable strategies, and (3) translate them into an HDL description of the CGL (the sought after strategies for clock-inhibition are given in a symbolic form, so their translation into a description of synchronous circuit in an HDL is essentially syntactical); (4) construct a new design that integrates this CGL. Note that only step (1) above requires some insight from designers about the design, and all the other stages are fully automated.

We further detail the process abstraction procedure and associated definition of control objectives to obtain a power-aware CGL in the remainder of this Section.

5.3.2 Abstracting Process Implementation Behaviors

Our translation and modeling algorithm takes as inputs the set P of all processes in the original design, and produces a model M along with objectives suitable for computing an implementable CGL by means of symbolic control. The computations within each individual process is described using a module in an HDL, that accepts a dedicated `clk` signal used to drive the updates of its registers.

1) *Selected HDL Variables:* Our abstraction algorithm is parametrized with a set of

⁴Each available at <https://github.com/mozbaltan/dcs4cgl>, <https://scm.gforge.inria.fr/anonscm/git/reatk/reatk.git/>, and <https://gforge.inria.fr/anonscm/git/reatk/ctrl2hdl.git>.

selected HDL variables that make up the portion of state and input spaces that is *precisely represented* in the constructed model of each process. This key aspect allows designers to exploit the knowledge they have on their designs, in particular the usual distinction between command parts and operational parts. Every wire and register that is not explicitly selected is abstracted away and replaced by *oracles*:

2) *Oracle Symbols*: Indeed, while translating HDL expressions into their symbolic counterpart, our algorithm abstracts away sub-expressions by creating a set of *oracles* to replace them in the models. From the point of view of the constructed models, oracles are *non-controllable input symbols*, which means that the sought after strategy must be computed by assuming they can take any value at any tick. From the point of view of the resulting CGL, however, the actual values of the expressions abstracted away with oracles need to be known so as to evaluate the strategy it encodes. To this end, we also produce an “open” HDL implementation of every process, that features additional output wires carrying the values of the oracles (*i.e.*, the value of the expressions they represent). These additional wires are then used to feed the CGL when building up the new design.

Given an HDL expression e on any set of variables, the oracle symbol ω_e is an unknown input whose value is that of e at every tick. w_e can thus be used to model behaviors where e itself is abstracted away and can take any value in its domain. Every knowledge about the modeled behaviors is not lost however. Indeed, assuming that e and e' admit the same canonical representation e'' , every occurrence of e and e' can be replaced with the same oracle $\omega_{e''}$, and the equality of valuations for e and e' can still be represented. Then, every expression e that involves a non-selected HDL variable is first translated into a canonical form e' , and replaced with $\omega_{e'}$. An efficient way to compute canonical representations consists in using (multi-terminal) binary decision diagrams [20].

3) *HDL Traversal Procedure*: Our process behavior abstraction algorithm operates on a representation of the module implementation of each process p where occurrences of local wires have all been substituted with their respective expression. The algorithm first associates a guard with every clock-triggered assignment to selected registers by traversing every conditional/case constructs of the implementation. Then, for each selected register, it generates a series of cascading conditional constructs, whose leave expressions and predicate conditions are respectively built from values and guards by substituting any expression from the module implementation that involves non-selected HDL variables with oracles. In turn, the *idleness predicate* $idle_p$ corresponds to the conjunction of the negation of all guards from the above mapping.

```

1 module p (input clk, input start, input [31:0] i,
           output reg done, output reg [31:0] o);
   reg r1, r2; reg [31:0] r3;
   initial begin r1=0; r2=0 r3=0; o=0; done=0; end
5 always @(posedge clk) begin
   if (start & ~done) r1<=1;
   else if (r1 & ~done) begin
       if (r2 & r3==i) begin o<=r3+i; done<=1; r1<=0; r2<=0; end
       else if (r2) begin o<=i; done<=1; r1<=0; r2<=0; end
10      else begin r2<=1; r3<=i; end
   end else if (done) done<=0;
   end
endmodule

```

List. 5.1: A simple process in Verilog.

```

done := if inhibitp then done else
        if ¬(start ∧ ¬done) ∧ (r1 ∧ ¬done) ∧ ωr2 ∧ ωi-r3=0 then 1 else
        if ¬(start ∧ ¬done) ∧ (r1 ∧ ¬done) ∧ ωr2 ∧ ¬ωi-r3=0 then 1 else
        if ¬(start ∧ ¬done) ∧ ¬(r1 ∧ ¬done) ∧ done then 0 else done
idlep ≜ ¬(start ∧ ¬done) ∧ ¬(r1 ∧ ¬done) ∧ ¬done
Pp ≜ if inhibitp then 0 else
      if (start ∧ ¬done) then 1 else
      if ¬(start ∧ ¬done) ∧ (r1 ∧ ¬done) ∧ ωr2 ∧ ωi-r3=0 then 35 else
      if ¬(start ∧ ¬done) ∧ (r1 ∧ ¬done) ∧ ωr2 ∧ ¬ωi-r3=0 then 35 else
      if ¬(start ∧ ¬done) ∧ (r1 ∧ ¬done) ∧ ¬ωr2 then 33 else
      if ¬(start ∧ ¬done) ∧ ¬(r1 ∧ ¬done) ∧ done then 1 else 0

```

Figure 5.2: Extracts of symbolic model built from the example process of List. 5.1.

A similar process as for selected registers is used to construct the *power expression* P_p , where leaves of cascading conditional constructs denote the amount of potential register bit-flips (*i.e.*, the sum of the width of assigned registers) instead of assigned values or symbols. We claim that this measure gives us cost functions that are suitable for demonstrating the effectiveness of our approach.

Let us exemplify our translation procedure by examining the result we obtain from the toy process of List. 5.1. Assuming **start**, **r1**, and **done** constitute the set of selected HDL variables (as they all carry control-flow within **p**), we give in Figure 5.2 the discrete evolution that corresponds to **done**, the idleness predicate, and the power expression (the result for **r1** is similar to that of **done**). State symbol *done* is the counterpart of **done** in the model. Notice it does not evolve when the computations of **p** are inhibited. Also, observe that there is a one-to-one correspondence between every portion of guards of conditional constructs

and conditions in the HDL code. Further, P_p states that, *e.g.*, 33 bits may flip whenever the HDL expression $\sim(\text{start} \ \& \ \sim\text{done}) \ \& \ (\text{r1} \ \& \ \sim\text{done}) \ \& \ \sim\text{r2}$ holds, which manifests in the model as a guard involving the oracle ω_{r2} since r2 is abstracted away (r2 is not selected).

5.3.3 Abstract Process Observers & Control Means

We now describe in this section the additional parts of the constructed model, *i.e.*, observers and control objectives, that allow us to compute a strategy suitable to obtain a power-aware CGL. All these parts are automatically derived from the original design. For each process p , our construction assumes the availability of the following additional non-controllable input symbols: (i) one *FIFO emptiness* symbol empty_p , that holds whenever all FIFOs p feeds from are empty; and (ii) one *termination* symbol done_p , that holds for one tick whenever p terminates a job. A “slow” global clock slow-clk non-controllable input symbol may be used to enforce a preemptive model of concurrency among all processes.

1) *Abstract Process Observer*: The operational status of each process is modeled using a symbolic encoding of a two-state Mealy machine that transitions whenever its input holds. For a process p , this symbolic model is defined as

$$\text{suspended}_p := \text{if } \text{suspended}_p \text{ then } \neg c_p \text{ else } c_p. \quad (5.1)$$

The suspended_p symbol is a Boolean component of the *state space* that holds whenever the operations of p are suspended, whereas the symbol c_p denotes the Boolean input that drives suspended_p . c_p is *controllable*, meaning that it serves as a lever for the control strategy to suspend or activate p so as to fulfill its objectives. For readability, we additionally define $\text{activate}_p \triangleq \text{suspended}_p \wedge c_p$ as a symbol that holds iff the process resumes its computations; we also define $\text{activate} \triangleq \bigvee_{p \in M} \text{activate}_p$ that holds iff at least one process of the design is being activated at the current tick.

Observe that the design objectives that we are aiming for (*e.g.*, power-optimization), can straightforwardly be fulfilled by preventing every process involved from computing at all. A very simple strategy that induces such a behavior consists in ensuring that every suspended_p state variable holds at any tick; one can observe in Eq. (5.1) that such a strategy always exists. To restrict the set of eligible strategies to those that ensure *progress* and *fairness*, we devise a set of additional *safety objectives* that the target design must satisfy.

2) *Enforcing Strict Progress*: The most basic progress objective states that at least one process must be active at every clock tick unless all FIFOs are empty. It is ensured by

means of the predicate $\varphi_{\text{strict-progress}} \triangleq \bigvee_{\mathbf{p} \in \mathbf{P}} \neg \text{suspended}_{\mathbf{p}} \vee \bigwedge_{\mathbf{p} \in \mathbf{P}} \text{empty}_{\mathbf{p}}$.

3) *Enforcing Fairness*: In order to express the fairness objective, we symbolically encode *scheduling constraints* as part of safety objectives. To this end, we first augment the set of state components of the model by introducing one *bounded inactivity counter* $q_{\mathbf{p}}$ per process \mathbf{p} : $q_{\mathbf{p}}$ is reset whenever \mathbf{p} is activated, and increases if any process except \mathbf{p} is activated:

$$q_{\mathbf{p}} := \begin{cases} 0 & \text{if } \text{activate}_{\mathbf{p}} \\ q_{\mathbf{p}} + 1 & \text{if } \text{activate} \wedge \neg \text{activate}_{\mathbf{p}} \wedge q_{\mathbf{p}} + 1 < |\mathbf{P}| \\ q_{\mathbf{p}} & \text{otherwise.} \end{cases}$$

Remark that every counter $q_{\mathbf{p}}$ takes its values in the domain $Q \triangleq \{0, \dots, |\mathbf{P}| - 1\}$. We further declare a *priority list* by using an additional set of symbols p_i , for $i \in \{1, \dots, |\mathbf{P}|\}$. The p_i 's also take their values in Q , and are constrained according to the invariant φ_{prios} defined as:

$$\begin{aligned} \varphi_{\text{prios}} &\triangleq \bigwedge_{i \in \{1, \dots, |\mathbf{P}|-1\}} p_i \geq p_{i+1} \wedge && \text{[p-sorted]} \\ &\bigwedge_{i \in \{1, \dots, |\mathbf{P}|\}} \bigvee_{\mathbf{p} \in \mathbf{P}} p_i = q_{\mathbf{p}} \wedge \sum_{i \in \{1, \dots, |\mathbf{P}|\}} p_i = \sum_{\mathbf{p} \in \mathbf{P}} q_{\mathbf{p}}. && \text{[p-values]} \end{aligned}$$

The above constraint basically states that the list of values associated to the sequence $(p_i)_{i \in \{1, \dots, |\mathbf{P}|\}}$ is decreasing [p-sorted], and only contains values that belong to the set of all inactivity counters $q_{\mathbf{p}}$'s [p-values]. The p_i 's belong to the *controllable input space* of the model: this means that the actual computation of the priority list that these symbols denote (*i.e.*, sorting the values of all activity counters) is *encoded* as part of the target control strategy, and thus eventually within the piece of circuit that computes the CGL.

We eventually express the *fairness* constraint in terms of the above symbols as the conjunction

$$\varphi_{\text{fairness}} \triangleq \varphi_{\text{prios}} \wedge \bigwedge_{\mathbf{p} \in \mathbf{P}} (\text{activate}_{\mathbf{p}} \Rightarrow q_{\mathbf{p}} \in \{p_1, \dots, p_{|\mathbf{P}_{\text{act}}|}\})$$

where $|\mathbf{P}_{\text{act}}| \triangleq |\{\mathbf{p} \in \mathbf{P} \mid \text{activate}_{\mathbf{p}}\}|$ denotes the number of processes that are being activated. $\varphi_{\text{fairness}}$ states that if a process \mathbf{p} is activated ($\text{activate}_{\mathbf{p}}$), then the value of its inactivity counter must belong to the $|\mathbf{P}_{\text{act}}|$ highest ones. Put another way, $\varphi_{\text{fairness}}$ imposes that if a process is activated, then every other process whose inactivity counter is strictly higher is

also activated.

4) *Avoiding Starvation by Enforcing Concurrency*: One must also ensure the absence of starvation, which in our case may happen if a process is never suspended. A model of concurrency is therefore declared to ensure that processes are forcibly suspended upon certain circumstances. To clarify the definitions below, we define $stalled_p \triangleq suspended_p \wedge \neg idle_p$ to hold whenever the computations of some process p are suspended while its idleness condition does not hold. We give below two invariants that each correspond to a model of concurrency:

Forcing *cooperation* between processes makes use of the job termination symbol $done_p$ of each process p :

$$\varphi_{\text{coop}} \triangleq \bigwedge_{p \in P} \left(term_p \Rightarrow \bigvee_{p' \in P \setminus \{p\}} stalled_{p'} \Rightarrow \bigvee_{p' \in P \setminus \{p\}} activate_{p'} \right),$$

where $term_p \triangleq \neg suspended_p \wedge (done_p \vee idle_p)$, holds iff at least one stalled process distinct from p is activated whenever p terminates its current job or becomes idle. Alternatively, one can implement *periodic preemption* of processes. This is achieved with the help of the additional non-controllable input symbol $slow-clk$ that acts as a “slow” clock, with

$$\varphi_{\text{preempt}} \triangleq slow-clk \Rightarrow \bigvee_{p \in P} stalled_p \Rightarrow \bigvee_{p \in P} activate_p,$$

which states that at least one process be activated whenever $slow-clk$ holds while some process is stalled.

$\varphi_{\text{concurrency}}$ can be defined so as to hold whenever either or both φ_{coop} and/or φ_{preempt} hold, according to the desired model of concurrency.

5) *Putting it All Together with Clock-inhibition Signals*: We finally relate the controllable input symbols $inhibit_p$ with the model using the invariant

$$\varphi_{\text{inhib-suspended-only}} \triangleq \bigwedge_{p \in P} (inhibit_p \Rightarrow suspended_p),$$

which states that a process must be suspended for its clock to be gated. Overall, the global safety objective that the strategy must ensure by choosing values for all controllable signals (*i.e.*, for each process p , c_p and $inhibit_p$, and the p_i 's) corresponds to the conjunction

$$\varphi_{\text{strict-progress}} \wedge \varphi_{\text{fairness}} \wedge \varphi_{\text{concurrency}} \wedge \varphi_{\text{inhib-suspended-only}}. \quad (5.2)$$

Besides enforcing that clock-inhibition signals hold at appropriate clock cycles, this objective also enforces progress, fairness, and a suitable model of concurrency, by virtue of the encoded scheduling constraints.

5.3.4 Achieving Power-efficiency by Control

We define the estimated measure of the total instantaneous power consumption (in number of register flips) of all process from the original design as $P_P \triangleq \sum_{p \in P} P_p$.

Optional Peak Power Constraint: The first means of using our derived models to achieve objectives related to power efficiency is to specify that the sought after strategy should ensure a given upper-bound P_{\max} on the value of P_P : one can achieve this with the help of the additional safety objective $\varphi_{P_{\max}} \triangleq P_P \leq P_{\max}$, to be appended with a conjunction to Eq. (5.2).

Energy Minimization: Further, our models offer an alternative means for enforcing some level of power-awareness of the scheduling induced by the CGL. They can indeed be used to refine a strategy towards minimizing the value of P_P (*i.e.*, some measure of instantaneous power consumption) summed over a time window, *i.e.*, minimizing the energy consumed.

5.3.5 Enabling Dynamic CGL Reconfiguration

The invariant of the model as defined above can optionally be refined to support the dynamic reconfiguration of power-awareness policies. For instance, introducing a user-accessible register that switches to/from a clock-gating logic based on idleness conditions only boils down to: (i) add a non-controllable Boolean input symbol cfg_{idle} to the model; and (ii) replace $\varphi_{\text{inhib-suspended-only}}$ in Eq. (5.2) with $\varphi_{\text{inhib-configurable}} \triangleq$

$$\bigwedge_{p \in P} (\text{inhibit}_p \Rightarrow (\text{if } cfg_{\text{idle}} \text{ then } \text{idle}_p \text{ else } \text{suspended}_p)).$$

This way, the power-aware scheduling policy can be turned off by setting the input wire of the resulting CGL that corresponds to cfg_{idle} to 1, thereby inducing a CGL behavior that corresponds to a classical clock-gating based on idle conditions only.

5.4 Experimental Evaluation

We have applied our approach on a series of various designs built from three RTL implementations of widespread signal coding or decoding algorithms: we use a Run-Length Encoder (RLE), a Huffman decoder, and a serial Reed-Solomon (RS) decoder⁵: (a) our first design consists of a pipeline made of the RS decoder followed by the RLE, and then the Huffman decoder; (b) remaining designs comprise a variable number N of RLEs put in parallel (*i.e.*, they receive and output jobs from external ports). Due to the size of the RS decoder, and its internal structure divided into many sub-modules, for this particular process we have applied our implementation abstraction procedure (*cf.* Section 5.3.2) on a clearly identifiable sub-module that encodes its control-flow. In addition, in all cases selected registers were easy to identify, as most control-flow related HDL variables were named “state”, “done”, etc. Overall, each resulting abstract process implementation model features around 7 Boolean state symbols, and 15 Boolean non-controllable input symbols (oracles).

To experimentally assess the functional correctness and compare the respective dynamic power dissipation of each of the designs at hand (the originals and all the power-aware ones), we first carried out logic synthesis on all of them using the Altera Quartus synthesizer. We then used the Altera ModelSim simulation tool to perform functional simulations using the same benchmark for all circuits originating from the same designs, and checked that the resulting output traces were strictly equivalent.

⁵Each available at <https://github.com/peterqt95/rle>, <https://github.com/rahuldhameja/Huffman-Decoder>, and https://opencores.org/project,reed_solomon_decoder.

Design Objective	Strategy Computation		Total Size (bits)		Power Consumption of Resulting Design		
	Time (s)	Max Memory (MB)	Logic	Registers	Cycles	Cycle Average (mW)	Saving (%)
Original	n/a	n/a	30914	23835	5969	353.68	–
Idleness	0.08	45.56	30924	23837	5969	344.81	2.51
$P_{\max} \leq 200$	0.40	48.12	31270	23857	11935	231.49	34.55
$P_{\max} \leq 199$	0.25	47.27	n/a	n/a	n/a	n/a	n/a
Optim.	3.96	262.87	31224	23857	11932	223.20	36.89
Cfg.-Idleness	5.69	235.68	31305	23858	5969	363.97	-2.91
Cfg.-Optim.	5.69	235.68	31305	23858	11932	226.60	35.93

Table 5.1: Evaluation results for the original and power-aware designs; the “Cycles” column denotes the total number of clock cycles required for processing the considered test-bench (Device: Cyclone IV, Frequency: 100Mhz).

N	Design Objective	Time (s)	Memory (MB)	Feasibility
2	$P_{\max} \leq 70$	0.08	45676	✓
	$P_{\max} \leq 69$	0.10	45676	✗
	$P_{\max} \leq 70$ +Optim.	0.22	46844	✓
	$P_{\max} \leq 69$ +Optim.	0.09	47328	✗
3	$P_{\max} \leq 70$	0.47	48148	✓
	$P_{\max} \leq 69$	0.64	50760	✗
	$P_{\max} \leq 70$ +Optim.	3.54	126504	✓
	$P_{\max} \leq 69$ +Optim.	0.66	50840	✗
4	$P_{\max} \leq 70$	3.54	78980	✓
	$P_{\max} \leq 69$	23.18	225616	✗
	$P_{\max} \leq 70$ +Optim.	250.62	3249848	✓
	$P_{\max} \leq 69$ +Optim.	23.08	225784	✗

Table 5.2: Performance of the strategy computation tool for N RLE instances.

The original design (a) was subject to various objectives and configurations, the corresponding results of which we report in Table 5.1. The “Idleness” objective corresponds to a design where the CGL operates based on idleness conditions only. “ $P_{\max} \leq 200$ ” and “ $P_{\max} \leq 199$ ” both seek to impose a strict upper-limit to instantaneous power consumption based on some maximum amount of register flips. “Optim.” results from a CGL that achieves a minimization of energy over a sliding window of 3 ticks. Lastly, lines where objectives are prefixed with “Cfg.-” correspond to a single design that incorporates a reconfigurable CGL, as described in Section 5.3.5.

We first observe that there does not exist a strategy that is able to impose the safety objective $P_{\max} \leq 199$; this actually reveals that it is not possible to meet both this objective and the scheduling constraints for this particular design. Further, even using a small time window of 3 ticks for the optimization objective, we can compute a CGL that reduces the (simulated) average power consumption per cycle by about 37%.

We have further evaluated the efficiency of the strategy computation tool by applying our approach on original designs (b). We report the results in Table 5.2, where objectives suffixed with “+Optim.” correspond to designs where both an upper-limit on power consumption and a minimization of energy over a sliding window of 3 ticks is desired.

In the literature, there are several works for the implementation of scheduling policies on hardware designs, where the inactivity is supported by the clock-gating technique. However, the switch-off signal of a clock signal is an external signal, where its control logic is built in

the software part of a system. Similarly, a few works, which addressed the management of the dynamic region of reconfigurable devices, use an external signal, which is built with software, for management (*e.g.*, [4]). However, the scheduling policy has not been applied in the literature for the module level (*i.e.*, in the hardware part of a system). We offer embedded control signal generation, in the hardware part of a system, unlike the other approaches. The main advantage of our approach is that one can access any variable of a design and combine the problem with some kind of hardware design problems such as idleness condition. Our approach allows for higher power savings with a single controller, where the results support our claim although it is not available to calculate the best theoretical estimation. Furthermore, we applied our approach on realistic RTL designs, where today's RTL designs are generally composed as a hierarchical structure. Thus, we show that it can be applied for any hierarchical design.

5.5 Summary and Discussion

Several commercial and academic tools already target automated clock-gating from RTL code. [89] developed an algorithm that automatically introduce the some clock-gating logic into RTL descriptions of circuits, although they focus on the exact computation of idleness conditions for each individual registers. In turn, [7] suggest an algorithm that automatically tries to approximate idleness conditions. [12] detect idleness conditions by using explicit finite-state machines in an approach that targets the conditional activation of individual hardware components using their “enable” signal (an approach similar to clock-gating). At last, [89] exploited conditional statements and case structures within blocks of clock-triggered assignments in HDL languages to determine such conditions. Our approach draws from the latter since it also works based on module-level HDL descriptions, and we build our symbolic models based on the conditional clock-triggered assignments.

[105] develop a formal framework based on data-flow models to analyze hardware circuits, and derive some control logic to drive them towards various performance objectives; our technique operates on similar models, and rely on abstracted versions of the circuits to tackle the state-space explosion problem. More practically, [19] also focus on system-level to suggest a clock gating technique that operates on whole modules.

We have advanced a tool-supported framework for producing power-aware designs from RTL implementations of KPNs. Our technique permits the automated construction of an abstract symbolic model of the design, as well as associated control objectives. The

automatically computed strategy is then translated into a piece of circuit that encodes a clock-gating logic for the design, and guarantees that the specified objectives are met. We plan to design guidelines for using our approach on black-box IPs with user-provided symbolic models. As stated in Section 5.2, the strategies are usually computed under the assumption that the environment of the model behaves as an adversary: in a sense the strategy is pessimistic. A natural extension of our work is to take some stochastic models of the environment (*e.g.*, inferred from simulation traces) into account to compute strategies that achieve better power efficiency on average.

Chapter 6

A Case for Symbolic Limited Optimal Control: Energy Minimization in Data-flow Circuits

In the previous chapter, we constructed a generic abstract model for data-flow networks to implement the power-aware scheduler. In this chapter, we build our models for configurable data-flow networks, where the hardware design problem we consider is achieving the efficient dynamic reconfiguration of configurable data-flow networks subject to global design objectives such as mutual exclusions and minimization of energy consumption. Thus, we re-specify the control objectives and the abstraction behavior of processes introduced in the previous chapter, and also abstract the FIFOs, in order to implement the energy-aware configuration manager. Our approach relies on discrete control techniques, where the goal is to compute a strategy that can be used to drive a given system model so that it satisfies a set of control objectives. We base our framework on a new symbolic limited optimal control algorithm that is able to optimize a cost function summed over a sliding window of a given number of ticks (steps). We present an original technique for constructing symbolic models of configurable data-flow networks and automatically compute dynamic configuration managers. We use these models to experimentally evaluate our new control algorithm, and make the case for optimal control on such networks. Our paper¹, which includes the contents and results of this chapter, have been under the submission and review.

¹Author: Mete Özbaltan and Nicolas Berthier.

6.1 Introduction

In this work, we consider *reactive data-flow circuit* designs similar to KPNs, where each process offers a *discrete configuration means* that impacts its computation speeds and power consumption. This can be the case in practice when several implementations are available to perform a given computational task, each with various levels of resource consumption, quality of service, etc. Process implementations may also make use of shared resources in such a way that *mutual exclusion constraints* must be enforced. Here, reactive additionally means that the designs has to respond to a potentially infinite stream of new data. Finding the optimal configuration for such a network at any given time is intrinsically a challenging task.

We advance a framework for automatically computing and integrating a *dynamic configuration manager* into such designs. This manager is able to satisfy *global design objectives*, such as reducing the overall energy consumption of the design while meeting the mutual exclusion constraints. We rely on the construction of an *abstract symbolic system model* of the design, and employ *discrete control* techniques to compute a piece of code (*e.g.*, a synchronous circuit) that implements *energy-efficient configuration management*.

In addition, as we shall observe in Section 5.2, existing discrete optimal control solutions that address the kind of optimization problems only focus on runs that reach a given set of target states. As such, they do not suit the needs imposed by the reactive designs we consider, for which one cannot identify a suitable set of target states. Therefore, we first devise a *new algorithm for achieving symbolic limited optimal control* in Section 6.2, and then present in Section 6.3 our approach for constructing symbolic system models of configurable data-flow networks that permit the construction of energy-aware dynamic configuration managers. In Section 6.4, we then use the models to: (i) experimentally evaluate our new algorithm; (ii) make the case for applying limited optimal control on such designs; (iii) demonstrate the power of such symbolic techniques for producing results that can directly be implemented as control mechanisms in, *e.g.*, hardware circuits. At last, we conclude with the summary and discussion in Section 6.5.

6.2 An Algorithm for Symbolic Limited Optimal Control

Let us now present the algorithm we devised for refining a base strategy σ towards fulfilling a given optimization objective.

Inputs: We consider the optimization objective that seeks the minimization of a cost function $\zeta \in \mathcal{X}_{\mathcal{N}, X \cup I}$ summed over $k \in \mathbb{N}^+$ ticks, with $\mathcal{N} \in \mathcal{D}_{\text{num}}$. Observe that ζ actually associates a cost on *every transition* of the model: given a state $q \in \text{Val}(X)$ and a valuation for every inputs $\iota \in \text{Val}(I)$, $\llbracket \zeta \rrbracket q \uplus \iota$ gives the cost incurred by the current tick as a quantity in \mathcal{N} . In turn, the base strategy σ is given as a predicate on state and input variables: it belongs to $\mathcal{P}_{X \cup I}$.

6.2.1 Computing Expected Outcomes

Our algorithm starts by computing the *best outcome* η_k as a symbolic outcome expression on the numerical domain $\mathcal{N}^* \stackrel{\text{def}}{=} \mathcal{N} \cup \{\infty\}$, *i.e.*, \mathcal{N} extended with ∞ so that ∞ is the supremum element for \mathcal{N}^* , and $\infty < \infty$ does not hold². $\eta_k \in \mathcal{X}_{\mathcal{N}^*, X \cup I}$ gives the best outcome that any strategy refined from σ can achieve from any valuation of the state and input variables on a time window of k ticks. η_k can be recursively computed as

$$\eta_1 \stackrel{\text{def}}{=} \text{if } \sigma \text{ then } \zeta \text{ else } \infty \quad (6.1)$$

$$\eta_{i+1} \stackrel{\text{def}}{=} \text{if } \sigma \text{ then } \left(\max_U \circ \min_C(\eta_i) \right) [T] + \zeta \text{ else } \infty \quad (6.2)$$

where

$$\min_V(s), s \in \mathcal{S} \stackrel{\text{def}}{=} s \quad (6.3)$$

$$\min_V(ce), c \in \mathcal{N} \stackrel{\text{def}}{=} \begin{cases} c \min_V(e) & \text{if } c \geq 0 \\ c \max_V(e) & \text{if } c < 0 \end{cases} \quad (6.4)$$

$$\min_V(e_1 + e_2) \stackrel{\text{def}}{=} \min_V(e_1) + \min_V(e_2) \quad (6.5)$$

$$\begin{aligned} \min_V(\text{if } p \text{ then } e_1 \text{ else } e_2) &\stackrel{\text{def}}{=} \\ &\text{if } \exists_V p \wedge \exists_V \neg p \text{ then } \min(\min_V(e_1), \min_V(e_2)) \\ &\text{else if } \exists_V p \text{ then } \min_V(e_1) \text{ else } \min_V(e_2) \end{aligned} \quad (6.6)$$

with $\min(e_1, e_2) \stackrel{\text{def}}{=} \text{if } e_1 < e_2 \text{ then } e_1 \text{ else } e_2$ (and similarly for \max_V , \max). $\exists_V p$ denotes the existential elimination of every *finite* variable in V from a predicate p . Eqs (6.3)-(6.6) describe a solution to the *symbolic optimization* problem that consists in finding, given a set of variables V and a numerical expression $e \in \mathcal{X}_{\mathcal{N}, W}$, a numerical expression e' *without*

²In the case of a maximization, \mathcal{N} is extended with $-\infty$ instead.

any variable from V (i.e., $e' \in \mathcal{X}_{\mathcal{N}, W \setminus V}$) such that: (i) e evaluated according to any possible valuation for all variables in V is always greater or equal than e' ; and (ii) there exists a valuation for V such that e equals e' ; i.e., $\forall \omega \in \text{Val}(\text{Support}(e) \setminus V)$,

- (i) $\forall \nu \in \text{Val}(V), \llbracket e \rrbracket \omega \uplus \nu \geq \llbracket e' \rrbracket \omega$; and
- (ii) $\exists \nu \in \text{Val}(V), \llbracket e \rrbracket \omega \uplus \nu = \llbracket e' \rrbracket \omega$.

Eq. (6.3) is straightforward since V must not contain infinite variables and $s \in \mathcal{X}_{\mathcal{N}}$: therefore $s \notin V$. Eqs (6.4)-(6.6) operate recursively on the structure of the expression e . Elimination in conditional constructs essentially involves building a new expression that separately handles three sub-cases based on whether there exist valuations for V that maintain satisfaction of the condition p or not—the fourth case, that does not appear in Eq. (6.6), equates to p unsatisfiable.

Going back to Eqs (6.1)-(6.2) for the computation of η_k , the base case for $k = 1$ consists in associating every transition that does not satisfy σ with the supremum cost ∞ . In turn, the computation of η_{i+1} given η_i in Eq. (6.2) can be broken down as follows: (\min_C) solve the symbolic optimization problem of finding the minimum for every controllable variable (C): this actually represents the choice of values for controllable input variables that best fulfill the objective; (\max_U) solve the dual symbolic optimization problem for every non-controllable variable (U), representing the worst possible move of the environment against the desired objective. This leads to a numerical expression that only involves state variables since the combined optimization problems above eliminate all input variables; ($\cdot[X \mapsto T]$) substitute every state variable with its corresponding evolution expression (note that this may re-introduce input variables); ($\cdot + \zeta$) add ζ to account for the cost of the additional transition. The result of this operation builds a new cost function that associates any transition in the underlying state-machine with the best expected outcome that can be achieved using any choice for controllable variables against the worst choices for non-controllable variables on a *subsequent* path of length i ; (*if σ then \cdot else ∞*) lastly, associate any choice for inputs that does not satisfy the strategy to be refined σ with the supremum cost.

6.2.2 Computing the Refined Strategy

The result $\eta_k \in \mathcal{X}_{\mathcal{N}^*, X \cup I}$ of the above computations represents the best expected outcome towards the optimization objective depending on the valuations of state and input variables. Therefore, a strategy that fulfills the objective consists in choosing values for variables in C

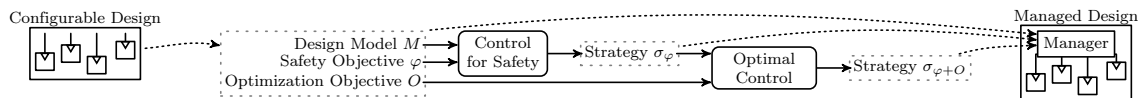


Figure 6.1: Overview of our suggested work-flows for computing energy-aware configuration managers.

that minimize η_k at the current tick, given valuations for state and non-controllable input variables. We compute the predicate that encodes this strategy as

$$\sigma' \stackrel{\text{def}}{=} (\nexists_{C'} (\eta_k [C \mapsto C'] < \eta_k)) \wedge \sigma \quad (6.7)$$

where C' are primed versions of all variables in C . The innermost parenthesized expression in Eq. (6.7) denotes the condition upon which choices for variables in C' are expected to produce a strictly better outcome according to η_k , than choices for variables in C . The resulting strategy σ' thus consists in keeping only choices for C such that no strictly better choice for variables in C' exists, and ensuring that these choices are indeed compatible with the strategy to be refined σ .

6.3 Use-case: Energy-aware Configuration Management

6.3.1 Management of Configurable Designs

We consider *controllable designs* made of *processes* that communicate through *channels*. Each process consumes and processes data from one or more input channels whenever possible—*i.e.*, a process does not wait unless one of its input channels is empty—and produces results into at least one output channel. *Upon consumption* of new pieces of data, each process reads a *configuration signal* that influences the speed (*e.g.*, the number of clock cycles in a hardware circuit) and power consumption it takes it to finish processing the data and produce an output. Lastly, the design may feature a set of shared resources that processes may make use of depending on their configuration (*e.g.*, a specialized signal processing unit in an FPGA): choices for process configurations must therefore obey a set of *mutual exclusion constraints*. The design receives pieces of data from its environment through one or more input queues.

Given such a design, our goal is to automatically construct an *energy-aware configuration manager* whose role is to *dynamically* select configurations for each process according to

various *global design objectives*.

6.3.2 Overview of the Approach for Computing Managers

We describe in Figure 6.1 the work-flows offered by our approach. A symbolic system model M is first constructed from the configurable design. M is made of the synchronous parallel composition of: (i) a model for each channel, defined using state and non-controllable variables, and appropriate encoding of discrete evolutions; (ii) a similar model for each process, that additionally involves controllable variables that offer levers for the sought-after manager to select configurations. We associate the symbolic system model with a series of definitions based on process and channel models, that permit the expression of control objectives (see below).

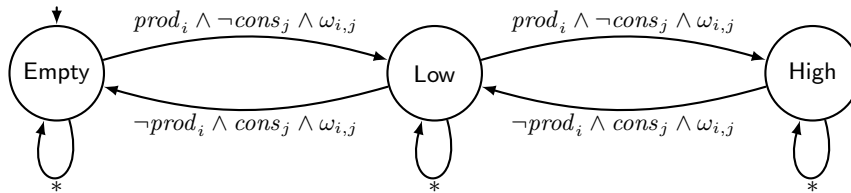
Then, a *safety objective* φ is built, in the form of a conjunction of propositional formulas that involve the state variables of process models. Each one of these formulas expresses a mutual exclusion constraint between process configurations that make use of a shared resource.

At this stage, a symbolic safety control algorithm can be used to compute a strategy σ_φ . This strategy is guaranteed to select values for the process configuration inputs (*i.e.*, controllable input variables of M) that ensure that the safety objective φ is satisfied. σ_φ and M can be used in combination to form a manager that outputs configuration choices for each process; a design whose processes are configured according to the outputs of this manager cannot violate any of the aforementioned mutual exclusion constraints. Alternatively, the strategy σ_φ can be improved by using a symbolic optimal control algorithm, such as the one we will present in Section 6.2, that ensures an additional optimization objective O . In our case, the cost function that we use to define O basically consists of an estimation of the energy consumption for all processes. The resulting refined strategy $\sigma_{\varphi+O}$ can be used in the same way as σ_φ to dynamically select configurations for each process.

We further detail the construction of the symbolic system model M and associated definition of control objectives to obtain energy-aware designs in the remainder of this Section.

6.3.3 Abstract Channel Model

Several options are available when modeling the kind of channels featured in the configurable models we consider. The choice for one option or the other notably depends on whether

Figure 6.2: Automaton representation for 3-state channel model encoded as $q_{i,j}$.

the manager that we seek to obtain needs to precisely track the level of occupation of the FIFOs or not, or, if FIFOs were bounded for instance, if it actually needs to ensure the absence of overflows.

In our use-case the configuration manager can only alter the speed of processes by selecting appropriate configurations, and preventing FIFO overflows would therefore require more insight (such as computation rates) about the processes at hand. We can however suppose that a manager that is able to distinguish *almost empty* FIFOs will be able to leverage this additional knowledge of *future process activities* to achieve a more energy-efficient *planning for process configurations*. In order to permit such a distinction while limiting the growth of the state-space, we *abstract* the state of a queue from process i to process j using a state variable $q_{i,j} \in X$ that takes its values in $\{\text{Empty}, \text{Low}, \text{High}\} \in \mathcal{D}_{\text{fin}}$. We further introduce *non-determinism* in each channel model $q_{i,j}$ with the help of a Boolean *oracle* input $\omega_{i,j} \in U$. This oracle is a non-controllable input used to non-deterministically transition between the abstract states of the queue.

We illustrate in Figure 6.2 the abstract behavior of a FIFO using an automaton partitioned according to the domain of $q_{i,j}$; this automaton is strictly equivalent to the assignment that we use to define the discrete evolutions of every $q_{i,j}$ in the symbolic model M . $prod_i$ and $cons_j$ denote predicates that hold whenever production and consumption of elements occurs in the queue: we define the associated symbolic expressions in the next Section as part of the model of processes. Observe that, for instance, this automaton clearly shows that the shortest path from High to Empty involves two ticks where $cons_j$ holds (and $prod_i$ does not).

6.3.4 Abstract Process Model

Let us now turn our attention to the symbolic model of a process p , defined as:

$$\begin{aligned}
cons_p &\triangleq \bigwedge_{i \in Q_p} (q_{i,p} \neq \text{Empty}) \wedge \text{if } p_p = \text{Idle} \text{ then tt else } e_p \\
prod_p &\triangleq p_p \neq \text{Idle} \wedge e_p \\
p'_p &\triangleq \text{if } cons_p \text{ then Active-}cfg_p \text{ else} \\
&\quad \text{if } \neg cons_p \wedge prod_p \text{ then Idle else } p_p \\
p_p &:= p'_p
\end{aligned} \tag{6.8}$$

A process p becomes or remains active whenever a data token is available in all its input queues (a set that we note here with Q_p): this is represented in the model using a predicate $cons_p$. The model of a process p also features an predicate $prod_p$, which indicates that the active process p has terminated its computations and produced a new piece of data into (each one of) its output queue(s). Since the model does not track any internal operational process state, and in particular none of its progress towards terminating its computations, an input variable e_p is used to drive process terminations. Upon termination of its current computations (this event manifests in the system model as a tick where e_p holds), p either becomes idle if any one of its input queue is empty, or consumes a new element from all of them and remains active. Eq (6.8) defining expression p'_p represents the value held by state variable p_p *starting from the next tick*; the new abstract state of p 's model that this expression represents is either `Idle`, or some active state that is tagged with the configuration that was selected by the controller as controllable variable cfg_p when p last consumed data. The state variable $p_p \in X$ takes its values in the domain $\text{Modes}_p \stackrel{\text{def}}{=} \{\text{Idle}, \text{Active-1}, \dots, \text{Active-}|\text{Configs}_p|\} \in \mathcal{D}_{\text{fin}}$ where $|\text{Configs}_p| \in \mathbb{N}^+$ is the number of available configuration options for p ; in turn, $cfg_p \in C$ is defined in the domain $\{1, \dots, |\text{Configs}_p|\} \in \mathcal{D}_{\text{fin}}$.

Worst-case Energy Estimation: In our modeling approach, the designer associates each configuration c for a process p with a worst-case execution-time $wcet_{p,c} \in \mathcal{N}$ and peak-power consumption $pp_{p,c} \in \mathcal{N}$, using a numerical domain $\mathcal{N} \in \mathcal{D}_{\text{num}}$. An over-approximation of the energy consumption incurred by any configuration choice for a process p that consumes some data during a tick (and thus starts a new computation cycle) can therefore be expressed

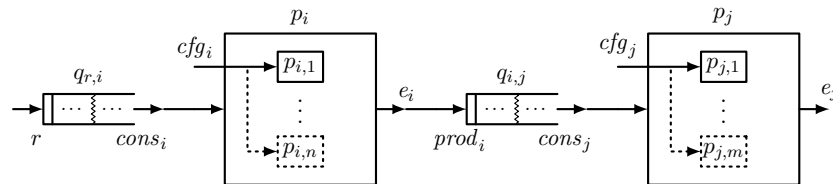


Figure 6.3: Graphical representation of a 2-process 2-channel configurable design; the $p_{i,1}, \dots, p_{i,n}$'s represent the distinct configurations of process i —similarly for j . Notice that r, cfg_i, cfg_j, e_i , and e_j are inputs of the system model, whereas $cons_i, prod_i$ and $cons_j$ denote expressions (predicates).

as

$$we_p \triangleq \sum_{c \in \text{Modes}_p} \text{if } p'_p = c \text{ then } w_{cet_{p,c}} \times pp_{p,c} \text{ else } 0$$

and the global energy consumption $we \in \mathcal{X}_{\mathcal{N}, X_{UI}}$ is the sum over all processes P :

$$we \triangleq \sum_{p \in P} we_p.$$

Observe that $(w_{cet_{p,c}} \times pp_{p,c})$ terms are constants that can be evaluated during the construction of the symbolic model. Also, we involves expressions for p'_p as defined in Eq (6.8), and it thus associates energy costs to *transitions* of the system.

We illustrate a simple configurable design in Figure 6.3, using the state and input variables for process and channel models as defined above. Input r denotes the arrival of a new piece of data into the network.

6.3.5 Control Objectives

Our control objectives fall into two categories: *safety objective* φ and *optimization objective* O . In our use-case, the former category embodies mutual exclusion constraints between process configurations that share resources. For instance, a safety property given as predicate $\varphi \triangleq \neg(p_p = \text{Active-}i \wedge p_q = \text{Active-}j)$ indicates that processes p and q cannot be active in their respective i th and j th configurations at the same time. At last, one can make use of the worst-case energy estimation expression we defined above to specify an optimization objective O to be enforced using the symbolic optimal control algorithm we presented in Section 6.2.

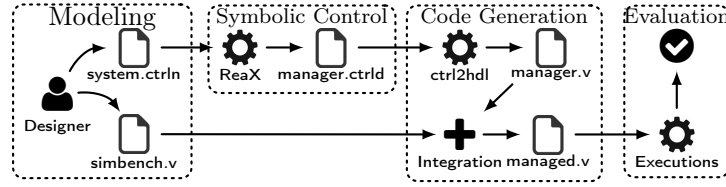


Figure 6.4: Synthesis & simulation tool-chain for assessing global design objectives.

6.4 Experimental Evaluations

Let us now turn to the experimental evaluations of our contributions. Apart from carrying out some performance evaluations on our implementation of the new optimal control algorithm of Section 6.2, we also want to empirically assess that it is actually able to enforce optimization goals by using the models constructed above. Indeed, such actual performance evaluations are necessary in the case of symbolic implementations, where one usually observes significant gaps between the practical performances and theoretical complexity results. We also want to experimentally assess whether configuration managers as produced using our approach effectively improve energy consumption whatever the sequences of inputs the resulting designs are subject to. This essentially means that the energy-efficiency of a manager needs to be evaluated on as many (realistic) scenarios as possible. At last, we want to observe the impact of the size of the sliding window used to specify the limited optimal control objectives.

6.4.1 Constructing Simulators of Managed Designs

We represent in Figure 6.4 an overview of the series of modeling steps and tools that we used to carry out our evaluations. The “Modeling” box on the left-hand side depicts the designer’s manual tasks of both constructing the symbolic system model and control objectives (in `system.ctrln`), as well as a simulation bench in the form of a Verilog file `simbench.v`. The latter will be used to carry out multiple stochastic simulations of the resulting managed design, encoded as a sequential circuit (we give more details on the simulation bench in Section 6.4.2 below).

In the “Symbolic Control” phase, we use `ReaX` to perform safety and limited optimal control on the system, triangularize the resulting strategy, and combine it with the original system to construct the manager. The latter is a controlled system (in file `manager.ctrln`), that can directly be translated into a sequential circuit using `ReaX`’s companion tool `ctrl2hdl`:

```

1 module process (input clk, input r, output reg e, input [15:0] et);
  reg [15:0] cnt;
  reg      state;
  initial begin e = 0; cnt = 0; state = 0; end
5  always @(posedge clk) begin
    if (r) state <= 1;          // start counting upon 'r'
    if (state) begin
      if (cnt != et) cnt <= cnt + 1;
      else begin e <= 1; state <= 0; cnt <= 0; end
10   end
    if (e) e <= 0;           // reset end signal 'e'
  end
endmodule

```

List. 6.1: Verilog module for simulating a process's behaviors.

```

1 module fifo (input clk, input c, input p, output ω);
  reg [31:0] fifo;
  wire      low_u = 10;
  wire      empty = 0;
5  assign ω = (p & ~c & (fifo==empty | fifo==low_u )) |
             (~p & c & (fifo==empty+1 | fifo==low_u+1));
  initial fifo = 0;
  always @(posedge clk) begin
    if (p & ~c) fifo <= fifo + 1;
10   if (~p & c) fifo <= fifo - 1;
  end
endmodule

```

List. 6.2: Verilog module for FIFO's simulated behaviors

this translator turns deterministic symbolic systems into equivalent code in a hardware definition language such as Verilog: this step produces `manager.v`. The integration of this manager with the test-bench produces a hardware circuit that can be efficiently executed using compilers and simulators for such synchronous circuits.

6.4.2 Simulation Bench

We give in List.s 6.1, 6.2, 6.3 and 6.4 the Verilog modules that make up a simulation bench for a design with 5 processes.

Module `process` of List. 6.1 simulates computations of dynamically parameterizable length. It accepts a parameter `et`, and basically starts counting upon receiving a *run* signal `r`. It emits `e` when its counter reaches the value of its parameter `et`. The value of `et` for a process will be randomly drawn upon every one of its data consumptions (*cf.* `simbench.v` below).

In turn, module `fifo` of List. 6.2 accurately tracks the amount of simulated payload

in a FIFO channel, and reports when this occupation level crosses some fixed boundaries using a dedicated output signal ω , which is fed as input to the manager (see module `main` below) to drive the corresponding abstract 3-state behavior illustrated in the automaton of Figure 6.2. In this particular instance, every concrete simulated FIFO that has between 1 and 10 elements is considered in the abstract state `Low`; the value of 10 is here chosen arbitrarily.

Module `main` of List. 6.3 assembles processes and FIFOs along with the manager produced by `ReaX` and `ctrl2hdl` (note that for an easier accounting of energy consumption by the simulation bench we include as many instances of the `process` module as individual process configurations). This module reflects the behavior of the modeled system: the simulated data tokens it receives through input `r` are directly pushed into FIFO `f01`. The manager is instantiated on line 47. It monitors `r`, the end of computation signal output by every simulated process (`e*`), as well as the oracle signals output by every FIFO (`w*`). The manager outputs the state of each channel (`q*`) and process (`p*`) as per the symbolic system model used to construct it. The state of processes as output by the manager encodes its configuration (it is a member of `Modes2`), and it can therefore be used in `main` to drive the corresponding `process` module instance.

At last, the Verilog bench that performs simulations of the managed design is given in List. 6.4. The `simbench` module feeds the simulated design by periodically raising `r`. When a simulated process configuration terminates, it draws new random values that are used in place of its actual power consumption and execution time in accord with the specification used to define the system model (in `system.ctrlIn` in Figure 6.4). `simbench` also reports simulated energy consumption data every 1000 clock cycles. This way of partitioning the resulting stochastic simulation trace into chunks of 1000 clock cycles allows us to gather statistics on the simulated energy consumption for the design under multiple configurations and loads (*i.e.*, state of the queues) at once.


```

1 module main (input clk, input r);
  wire [1:0] p1,p2,p3,p4,p5;
  wire [1:0] q01,q12,q14,q23,q35,q45;
  wire
5   e1,e2,e3,e4,e5;
  wire
  e11,e12,e21,e22,e31,e32,e41,e42,e51,e52;
  wire [15:0] et11,et12,et21,et22,et31,et32,et41,et42,et51,et52;
  wire
  cons1,cons2,cons3,cons4,cons5;
  wire
  w01,w12,w14,w23,w35,w45;

10 // Empty channel: value of Empty assigned by ctrl2hdl
  wire [1:0] empty = 2'b10;

  // Process modes: constants corresponding to modes Idle, Active-1, Active-2, assigned by ctrl2hdl
  wire [1:0] idle = 2'b10;
15  wire [1:0] active1 = 2'b01;
  wire [1:0] active2 = 2'b00;

  assign e1 = e11 | e12;
  assign e2 = e21 | e22;
20  assign e3 = e31 | e32;
  assign e4 = e41 | e42;
  assign e5 = e51 | e52;
  assign cons1 = ((p1!=idle ? e1 : 1) & q01!=empty);
  assign cons2 = ((p2!=idle ? e2 : 1) & q12!=empty);
25  assign cons3 = ((p3!=idle ? e3 : 1) & q23!=empty);
  assign cons4 = ((p4!=idle ? e4 : 1) & q14!=empty);
  assign cons5 = ((p5!=idle ? e5 : 1) & q35!=empty & q45!=empty);

  process p11 (.clk(clk & p1==active1), .r(cons1), .e(e11), .et(et11));
  process p12 (.clk(clk & p1==active2), .r(cons1), .e(e12), .et(et12));
  process p21 (.clk(clk & p2==active1), .r(cons2), .e(e21), .et(et21));
  process p22 (.clk(clk & p2==active2), .r(cons2), .e(e22), .et(et22));
  process p31 (.clk(clk & p3==active1), .r(cons3), .e(e31), .et(et31));
  process p32 (.clk(clk & p3==active2), .r(cons3), .e(e32), .et(et32));
35  process p41 (.clk(clk & p4==active1), .r(cons4), .e(e41), .et(et41));
  process p42 (.clk(clk & p4==active2), .r(cons4), .e(e42), .et(et42));
  process p51 (.clk(clk & p5==active1), .r(cons5), .e(e51), .et(et51));
  process p52 (.clk(clk & p5==active2), .r(cons5), .e(e52), .et(et52));

40  fifo f01 (.clk(clk), .c(cons1), .p(r), .w(w01));
  fifo f12 (.clk(clk), .c(cons2), .p(e1), .w(w12));
  fifo f14 (.clk(clk), .c(cons4), .p(e1), .w(w14));
  fifo f23 (.clk(clk), .c(cons3), .p(e2), .w(w23));
  fifo f35 (.clk(clk), .c(cons5), .p(e3), .w(w35));
45  fifo f45 (.clk(clk), .c(cons5), .p(e4), .w(w45));

  manager manager(.clock(clk),
    .r(r), .e1(e1), .e2(e2), .e3(e3), .e4(e4), .e5(e5),
    .q01(q01), .q12(c12), .q14(q14), .q23(q23), .q35(c35), .q45(q45),
    .w01(w01), .w12(w12), .w14(w14), .w23(w23), .w35(w35), .w45(w45),
50    .p1(p1), .p2(p2), .p3(p3), .p4(p4), .p5(p5));
endmodule

```

List. 6.3: Verilog module for a managed design with 5 processes; the `manager` module is the result of the control algorithms (*cf.* Figure 6.4).

```

1 module simbench;
  integer cnt_100,cnt_1000,cnt_completed;           // counters
  integer power11,...,power52;                     // configurations's power
  integer power1,...,power5;                       // processes' power
5  integer energy;                                  // accumulates total energy
  integer pp11 = 55,...,pp52 = 60;                 // peak-power specs.
  integer wcet11 = 100,...wcet52 = 90;             // wcet specs.
  // Process modes: constants corresponding to modes Idle, Active-1, Active-2, assigned by ctrl2hdl
  integer idle, active1, active2;

10
  reg clk, r;
  main DUT(.clk(clk),r(r));                        // main module is instantiated as DUT

  initial begin
15    // randomly draw first (mean) powers and execution times for each simulated process instance
    power11 <= $random(0.7 * pp11, pp11);
    DUT.et11 <= $random(0.7 * wcet11, wcet11);
    // ...
  end
20
  always @ (posedge(clk)) begin                   // at each clock-cycle:
    cnt_100 <= cnt_100+1; cnt_1000 <= cnt_1000+1;
    if (r) r <= 0;                                // reset r when it holds
    // In these simulations, raise r (simulate the arrival of a new piece of data) every 100 cc
25    if (cnt_100==100) begin r <= 1; cnt_100 <=0; end
    // process power depends on the process mode in main (DUT)
    if (DUT.p1==idle) power1 <= 0;
    if (DUT.p1==active1) power1 <= power11;
    if (DUT.p1==active2) power1 <= power12;
30    // ...
    // accumulate all processes' power consumption
    energy <= energy+power1+...+power5;
    if (cnt_1000==1000) begin
      // display total energy/completely processed data, and reset
35      $display("%d",energy/cnt_completed);
      energy <= 0; cnt_completed <= 0; cnt_1000 <=0;
    end
  end
  end

40  always @ (posedge(DUT.e5))
    // Increment data processing completion counter when p5 terminates
    cnt_completed <= cnt_completed+1;

    // At each cycle where p1 terminates,
45    // randomly draw new (already averaged) power-consumptions and execution times
    // for each one of its modes
    always @ (posedge(DUT.e1)) begin
      power11 <= $random(0.7 * pp11, pp11);
      DUT.et11 <= $random(0.7 * wcet11, wcet11);
50      power12 <= $random(0.7 * pp12, pp12);
      DUT.et12 <= $random(0.7 * wcet12, wcet12);
    end
  end
  // ...
endmodule

```

List. 6.4: Excerpts of stochastic simulation bench in Verilog for a design with 5 processes; this particular bench simulates the arrival of a new piece of data once every 100 clock cycles (cc), and reports every 1000 clock cycles the total energy consumed by the simulated design per piece of data completely processed. The quantitative values (power consumption and execution times of each process configuration) are drawn within 70% and 100% of their respective specifications. The code that stops the simulation is not shown.

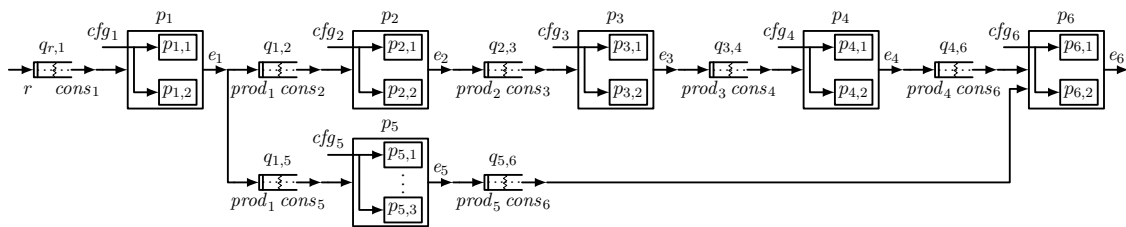


Figure 6.5: Example of configurable design with 6 processes.

Process p	$c \in \text{Modes}_p$	$wcet_{p,c}$	$pp_{p,c}$
p_1	Active-1	100	55
	Active-2	150	35
p_2	Active-1	40	70
	Active-2	50	27
p_3	Active-1	40	50
	Active-2	43	80
p_4	Active-1	40	50
	Active-2	43	80
p_5	Active-1	180	8
	Active-2	210	7
	Active-3	195	7
p_6	Active-1	80	70
	Active-2	90	60

Table 6.1: Specification values for our example design with 6 processes; the Worst-case Execution Times are given in number of clock cycles.

6.4.3 Simulation Results

Let us now present the simulation results we obtain for the design with 6 processes shown in Figure 6.5, whose specification values for worst-case execution times and power peaks for each process configuration are given in Table 6.1. Observe that in this instance process p_5 offers 3 possible configurations. Additional mutual exclusion constraints must ensure that process p_3 must not be in mode Active-1 while p_5 is in mode Active-1 (and conversely); similarly for mode Active-1 of p_4 and Active-2 of p_5 .

We represent in Figure 6.6 the gains in energy consumption per processed data that we observe *w.r.t.* the size of the sliding window used for the limited optimal control algorithm; observe that a plateau is reached for sliding windows of size greater than 5. This graph clearly illustrates that the size of the sliding window can greatly impact the ability of the limited optimal control algorithm to actually achieve a noticeable reduction in energy consumption.

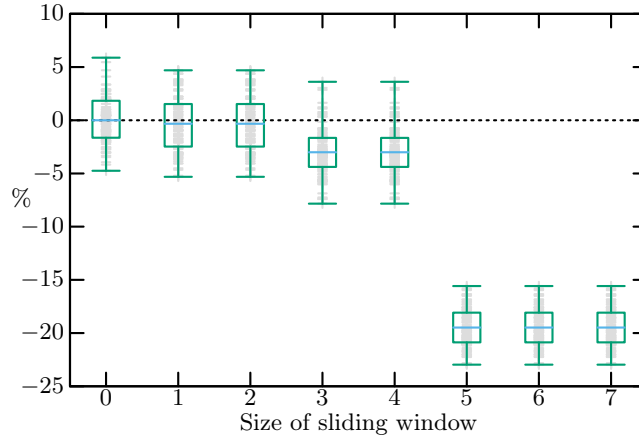


Figure 6.6: Gain in simulated energy per completely processed data for the design with 6 processes illustrated in Figure 6.5. For each size of sliding window (x-axis), we give the minimum, average, and maximum gain in percentage, as well as the low and high quartiles. A window of size 0 indicates that the manager is produced without any enforcement of optimal control objective. All gains (y-axis) are given in percentage of the average energy consumption obtained for the non-optimized design (represented with the dashed horizontal line).

N	k	Time (s)	Memory (KB)	Manager size (Logic elements)
5	0	0.07	45924	-
	1	0.78	61264	-
	2	2.44	94108	-
	3	6.14	246576	-
	4	8.73	284640	-
	5	10.38	276716	-
	6	10.99	266640	-
6	7	12.45	290976	-
	0	0.13	50424	660
	1	7.49	192940	661
	2	37.40	523300	663
	3	89.86	1572140	839
	4	133.2	2848112	1035
	5	164.42	2531132	1161
	6	198.01	3030064	1190
7	210.84	3054040	1192	

Table 6.2: Synthesis time and memory footprint of ReaX for designs involving either 5 or 6 processes (N), *w.r.t.* size of sliding window selected for the limited optimal control algorithm (k); we also report the size of the resulting manager circuit for the design with 6 processes.

We further report in Table 6.2 the run-time performances of ReaX for the designs models with 5 and 6 processes.

The studies of configuration management works have started to increase since the dynamic region has been integrated into reconfigurable devices. The management is implemented on the dynamic region as a software program (*e.g.*, [4, 95]). However, there is not any configuration management work on the static region of reconfigurable devices. We have addressed this issue for classical reconfigurable devices (*i.e.*, they do not have a dynamic region). We offer embedded controllers in the static region, so one can access any variable of a design and combine the problem with some kind of hardware design problems such as idleness condition. Furthermore, we propose our new optimization control algorithm, which alleviates the need for specifying target states to better accommodate the modeling of reactive systems for which the solutions are inadequate. As an example, [4]’s work focuses on finite-state systems equipped with cost functions relating to states only, and the work has not shown the effect of the higher steps. Thus, our approach allows for higher power savings, where the results support our claim although it is not available to calculate the best theoretical estimation. We applied our approach to a generic artificial design model, so one can easily adapt any RTL implementation to our generic design model.

6.5 Summary and Discussion

We have presented a new algorithm for achieving limited optimal control on symbolic system models. This algorithm alleviates the need for specifying target states by operating on a finite sliding window of parameterizable length. It also accepts cost functions directly defined on transitions, which is a novelty among symbolic approaches for optimal control.

In order to carry out some empirical evaluations of this new control algorithm, we have also advanced a framework for producing an energy-aware dynamic configuration manager for reactive data-flow circuits. Our technique permits the systematic construction of abstract symbolic models of such designs, as well as associated global control objectives. Through the construction of an efficient simulator using a hardware description language, we have also demonstrated that the symbolic model and the resulting strategy can be translated into a piece of circuit that encodes an efficient configuration manager. By construction, this manager ensures any set of mutual exclusion constraints on the design, and is able to reduce energy consumption.

We plan to develop a tool and design guidelines for using our approach on the automated

construction of the abstract symbolic models. As stated in Section 5.2, the strategies are usually computed under the assumption that the environment of the model behaves as an adversary: in a sense the strategy is pessimistic. A natural extension of our work is to take some stochastic models of the environment (*e.g.*, inferred from simulation traces) into account to compute strategies that achieve better energy efficiency *on average*. Last, our symbolic optimization algorithm could be extended in the two following directions: (i) the tool ReaX that we use for computing strategies is already able to enforce safety properties on infinite-state systems. An extension of our limited optimal control algorithm towards handling such systems appears a natural extension of our work; (ii) in some cases more than one design metrics are relevant for specifying optimal control objectives. Regarding our circuit use-case, one can for instance also identify the temperature as a factor that would be interesting to take into account.

Chapter 7

Conclusions and Future Recommendations

The research presented in the thesis achieves the power efficiency in hardware circuits with symbolic discrete control. The power efficiency of circuits is nowadays of paramount importance for constructing embedded electronic devices, as it is one of the major design constraints in today embedded systems limiting performance, battery life, and reliability [9]. However, as the technology advances and the transistor size decreases, designers are placing a higher amount of transistors on a circuit in order to build high-performance devices, but it brings extra power consumption and complex design [44, 45, 81]. To manage the increasing design complexity and power consumption of hardware circuits as well as to optimize various performance properties, the modeling and analysis are becoming a common practice [105]. We advocate safe design methodologies based on formal control techniques and formal abstract models for synchronous circuits. We have developed a systematic modeling framework, which can assist designers in tackling average and/or instantaneous power consumption after designed their behavioral models. The framework can be automatically employed to derive power-efficient versions from original circuit designs. Furthermore, we also used the framework to evaluate the power efficiency of various hardware designs. This chapter summarizes the main conclusions and future recommendations of this thesis.

This chapter is organized as follows: Section 7.1 summarizes the main conclusions of the work presented in this thesis; and Section 7.2 identifies open issues and recommends research and investigation areas for future directions.

7.1 Conclusions

This thesis started in Chapter 2 by providing the necessary overview of the current state-of-the-art in the context of integrated circuits, their design methodologies and low-power design techniques, and some CAD tools. More specifically, the chapter discussed the major challenges that occur while designing new generation integrated circuits (*i.e.*, containing a higher amount of transistors and including more complex design). The design complexity of circuits has been dealt with top-down abstraction methodology; however, performance goals (*e.g.*, power efficiency) still be a task to be applied [44, 45, 81]. Furthermore, a single integrated circuit can include different kinds of sub-circuits and architectures by using a HDL, which can support that a circuit is modeled by using a mixture of concepts of different design steps. As a result, it is essential to have a general knowledge about integrated circuits for performing scalable techniques to meet various performance goals.

Among these performance goals, the power efficiency is one of the major design constraint limiting performance, battery life, and reliability [9, 58, 94]. RTL clock gating is a low-power technique that promises maximum power saving among other low-power techniques applied on hardware circuits. Meanwhile, the implementation of this kind of technique applied on a hardware circuit is generally tedious and error-prone. As a result, providing a systematic framework that applies RTL clock gating allows automatically deriving power-efficient versions from original circuit designs.

To apply a technique for meeting some performance goals requires an efficient analysis; the use of modeling formalisms significantly reduces the effort of analysis to achieve such performance goals on hardware circuits, where the models abstract away the irrelevant details so that they allow designers to focus only on the essential properties of the system under design [105]. Chapter 3 discussed the existing modeling formalisms on hardware circuits designs emerged in the literature. As it is mentioned, the data-flow family is more effective modeling method than automata-based models to express hardware circuits, as they do not include the well-known state explosion problem [4, 105]. There are three main distinct variants of the data-flow model of computation emerged in the literature. KPNs can be used to describe systems where the amount of data produced and consumed by a process is not statically determined, and it can be generally assumed that a behavioral hardware design implements KPN, as KPN is a generalization form of the data-flow family. On the other hand, the main advantage of the synchronous language is that it allows the model can be built by effectively using conditional expressions; unlike the others, such as SDF. As a

result, it is essential to perform abstract modeling in the context of synchronous language in order to analyze a behavioral hardware design that implements KPN, while applying a technique that allows achieving some performance goals.

The techniques applied on hardware models for meeting various performance properties can be provided by means of several kinds of self-management approaches, such as control theory, model checking, heuristics, and machine learning techniques [4]. Compared to other existing techniques for achieving various performance goals, the main advantage of DCS is that it provides a synthesizable controller with formal correctness. Chapter 3 also provides the necessary overview of DCS and its implementations. As a result, performing the DCS technique on an abstract model of a hardware circuit to achieve various performance goals by means of some tools (*e.g.*, ReaX and BZR) encapsulated with the DCS operation, where the model is built in the form of the synchronous languages aspect, provides scalability and formal correctness for the controller synthesized within the compilation process.

In Chapter 4, we proposed a tool-supported framework for achieving power-efficiency of synchronous circuits from RTL behavioral designs, as hierarchical compositions of sub-circuits, described using the popular hardware description language Verilog. We have described a systematic approach that computes the CGL of synchronous circuits in order to switch off the clock of each sub-circuits (*i.e.*, to save power) when they are in an idleness status. More specifically, we encode the computation as several small symbolic discrete controller synthesis problems (*i.e.*, an abstract symbolic synchronous model by using Heptagon language for each individual Verilog modules) by means of the BZR tool, and use the resulting controllers (where controllable variables represent output wires of CGLs) to derive power efficient versions from original circuit designs. We have demonstrated the principles using an example, reported on its manual application on a realistic case study, and validated our approach experimentally. As a result, the resulting designs provide considerable power efficiency and guarantee the correct behavior (*i.e.*, the original design and its power-efficient version have the same behavior and produce strictly equivalent outputs). Furthermore, compared to other clock gating approaches applied to provide power efficiency on idle portions of a circuit, the approach comes with just a small number of logic units.

In Chapter 5, we have advanced a tool-supported framework for producing power-aware designs from RTL implementations of KPNs (described using a HDL, such as Verilog). Our approach relies on the automatic construction of abstract symbolic models of the designs, as well as associated control objectives, and employs discrete control techniques to compute a

piece of hardware circuit that implements some power-aware scheduling policies specified in a declarative way. This piece of circuit can eventually be used to selectively filter the clocks of the processes involved. The resulted designs are automatically produced by means of our tool `dcs4cgl`, where the DCS technique is performed by the tool `ReaX`, and the abstract models are encoded on the `ReaX` environment as well. We have illustrated and validated our approach and the strategies it provides, experimentally using various RTL designs described using Verilog. As a result, performing the approach/strategies provides the automatic construction of power-aware hardware circuits with the trade of timing performance from the considered original designs, where the resulted designs are functionally equivalent to originals and guarantee formal correctness; such designs offer a considerable impact on hardware resources (*e.g.*, increasing the lifetime of battery-powered devices and reducing the chip temperature and maximum supply voltage), where the power-awareness offered the approach is based on the minimization or limitation of the instantaneous power.

In Chapter 6, we have described a tool-supported framework in order to employ an energy-efficient configuration manager for choosing the optimal configuration, by means of the clock-gating logic, among the alternatives on data-flow hardware circuits implemented as KPNs, with parallel synchronous processes. Our technique permits the systematic construction of abstract symbolic models of such designs, as well as associated control objectives, so it is easily implementable. Then, we have employed the DCS technique along with the associated control objectives on the models by means of the `ReaX` tool, in order to produce a piece of hardware circuits that implements a configuration manager with energy reduction guarantees, where the manager behaves as a clock-gating constraint. We have applied and validated our approach on a series of various design flows built from RTL implementations (described using the HDL Verilog), where we use artificial processing modules and memory components. As a result, using our approach offers the systematic and semi-automated construction of an energy-aware configuration manager for such designs, where the manager then dynamically chooses the optimal one among all configuration alternatives involved in each process, by ensuring the given restrictions and the parallel synchronization of all processes, over a sliding window of a given number of ticks.

7.2 Open Issues and Future Recommendations

In this section, some open issues, which have been identified in the research presented in this thesis, are listed together with possible future research directions.

- With respect to the framework of Chapter 4 for the construction of energy-efficient synchronous circuits with symbolic discrete control based on the idleness conditions:
 - Our approach can be applied to hierarchical Verilog designs, which we applied to several such designs; however, it can also be extended to compute CGLs for individual registers within modules, and/or to apply for other HDLs and abstraction levels.
 - Although we exercised our technique to implement clock-gating as it currently offers the best trade-off between extra occupied circuit area and power savings, it is also applicable to other low-power design mechanisms such as power-gating (especially for computation-intensive modules that can be shut down for long periods of time).
 - Automatically identifying good sets of marked variables constitutes an interesting challenge.
 - The support of black-box sub-modules with simple user-provided models can also be considered.

- With respect to the framework of Chapter 5 for the construction of power-aware hardware circuits with symbolic discrete control based on the scheduling constraints and power-efficiency objectives:
 - Guidelines can be designed for using our approach on black-box IPs with user-provided symbolic models.
 - Our approach can be applied to the RTL designs that implement KPN, and we have implemented the approach on Verilog codes; however, it can be easily applied to other HDLs and can be easily extended for other abstraction levels.
 - Symbolic power expression that gives an estimated measure of the instantaneous dynamic power consumption of the modeled process is based on the state and input symbols of the process. The estimated measurement can be provided more accurately using a different kind of approaches, such as simulation.
 - The strategies stated in Chapter 5 are usually computed under the assumption that the environment of the model behaves as an adversary: in a sense the strategy is pessimistic. A natural extension of our work is to take some stochastic

models of the environment (e.g., inferred from simulation traces) into account to compute strategies that achieve better power efficiency on average.

- With respect to the framework of Chapter 6 for the construction of an energy-aware configuration manager with symbolic discrete control, based on the energy optimization objective and configuration constraints by supporting with parallel synchronous processes:
 - A tool and design guidelines can be developed for using our approach on the automated construction of the abstract symbolic models.
 - Optimization control algorithms work, from current states to target states, to produce a controller at the best cost of controllable events against the worst moves of uncontrollable events, so they (as well as our control algorithm) are pessimistic. Our control algorithm can be advanced in order to make a computation under the assumption of some stochastic inputs, where the computation achieves better energy-efficiency on average.
 - Our approach can be combined with some existing control techniques, such as that applies some power-aware scheduling policies, and/or the classical clock-gating implementation for the idleness objective.
 - Our approach can be enriched with other design metrics, such as temperature and area.

Bibliography

- [1] Nainesh Agarwal and Nikitas Dimopoulos. High-level fsmd design and automated clock gating with codel. *Canadian Journal of Electrical and Computer Engineering*, 33(1), 2008.
- [2] Sumit Ahuja, Wei Zhang, Avinash Lakshminarayana, and Sandeep K Shukla. A methodology for power aware high-level synthesis of co-processors from software algorithms. In *Proceedings of the 23rd International Conference on VLSI Design, VLSID '10*, pages 282–287. IEEE, 2010.
- [3] Karine Altisen, Aurélie Clodic, Florence Maraninchi, and Eric Rutten. Using controller-synthesis techniques to build property-enforcing layers. In *European Symposium on Programming*, pages 174–188. Springer, 2003.
- [4] Xin An. *High Level Design and Control of Adaptive Multiprocessor Systems-on-Chip*. PhD thesis, Université de Grenoble, 2013.
- [5] Xin An, Eric Rutten, Jean-Philippe Diguët, and Abdoulaye Gamatié. Model-based design of correct controllers for dynamically reconfigurable architectures. *ACM Transactions on Embedded Computing Systems (TECS)*, 15(3):51, 2016.
- [6] Xin An, Eric Rutten, Jean-Philippe Diguët, Nicolas Le Griguer, and Abdoulaye Gamatié. Discrete control for reconfigurable fpga-based embedded systems. *IFAC Proceedings Volumes*, 46(22):151–156, 2013.
- [7] Pietro Babighian, Luca Benini, and Enrico Macii. A scalable algorithm for rtl insertion of gated clocks based on odcs computation. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, 24(1):29–42, 2005.

-
- [8] Nilanjan Banerjee, Kaushik Roy, Hamid Mahmoodi, and Swarup Bhunia. Low power synthesis of dynamic logic circuits using fine-grained clock gating. In *Proceedings of the conference on Design, automation and test in Europe: Proceedings*, pages 862–867. European Design and Automation Association, 2006.
 - [9] Edith Beigné, Fabien Clermidy, Hélène Lhermet, Sylvain Miermont, Yvain Thonnart, Xuan-Tu Tran, Alexandre Valentian, Didier Varreau, Pascal Vivet, Xavier Popon, et al. An asynchronous power aware and adaptive noc based circuit. *IEEE Journal of Solid-State Circuits*, 44(4):1167–1177, 2009.
 - [10] Richard Bellman. *Dynamic Programming*. Princeton University Press, Princeton, NJ, USA, 1 edition, 1957.
 - [11] Luca Benini, Giovanni De Micheli, Enrico Macii, Massimo Poncino, and Riccardo Scarsi. Symbolic synthesis of clock-gating logic for power optimization of synchronous controllers. *ACM Transactions on Design Automation of Electronic Systems (TODAES)*, 4(4):351–375, 1999.
 - [12] Luca Benini, Polly Siegel, and Giovanni De Micheli. Saving power by synthesizing gated clocks for sequential circuits. *IEEE Design & Test of Computers*, 11(4):32–41, 1994.
 - [13] Nicolas Berthier. *Programmation synchrone de pilotes de périphériques pour un contrôle global de ressources dans les systèmes embarqués*. PhD thesis, Grenoble, 2012.
 - [14] Nicolas Berthier, Xin An, and Hervé Marchand. Towards applying logico-numerical control to dynamically partially reconfigurable architectures. *IFAC-PapersOnLine*, 48(7):132–138, 2015.
 - [15] Nicolas Berthier and Hervé Marchand. Discrete controller synthesis for infinite state systems with reax. *IFAC Proceedings Volumes*, 47(2):46–53, 2014.
 - [16] Nicolas Berthier and Hervé Marchand. Deadlock-free discrete controller synthesis for infinite state systems. In *2015 54th IEEE Conference on Decision and Control (CDC)*, pages 1000–1007. IEEE, 2015.
 - [17] Nicolas Berthier, Hervé Marchand, and Éric Rutten. Symbolic limited lookahead control for best-effort dynamic computing resource management. *IFAC-PapersOnLine*, 51(7):112–119, 2018.

-
- [18] Nicolas Berthier, Eric Rutten, Noël De Palma, and Soguy Mak-Karé Gueye. Designing autonomic management systems by using reactive control techniques. *IEEE Transactions on Software Engineering*, 42(7):640–657, 2016.
- [19] Rani Bhutada and Yiannos Manoli. Complex clock gating with integrated clock gating logic cell. In *Design & Technology of Integrated Systems in Nanoscale Era, 2007. DTIS. International Conference on*, pages 164–169. IEEE, 2007.
- [20] JP Billon. Perfect normal forms for discrete programs. *Bull, Tech. Rep*, 1987.
- [21] David C Black, Jack Donovan, Bill Bunton, and Anna Keist. *SystemC: From the Ground Up*, chapter 1–2. Springer Science & Business Media, 2 edition, 2010.
- [22] Leticia Bolzani, Andrea Calimera, Alberto Macii, Enrico Macii, and Massimo Poncino. Enabling concurrent clock and power gating in an industrial design flow. In *Proceedings of the Conference on Design, Automation and Test in Europe*, pages 334–339. European Design and Automation Association, 2009.
- [23] Christos G Cassandras and Stephane Lafortune. *Introduction to Discrete Event Systems*. Springer Science & Business Media, 2009.
- [24] Xiaotao Chang, Mingming Zhang, Ge Zhang, Zhimin Zhang, and Jun Wang. Adaptive clock gating technique for low power ip core in soc design. In *2007 IEEE International Symposium on Circuits and Systems*, pages 2120–2123. IEEE, 2007.
- [25] Pong P Chu. *RTL hardware design using VHDL: Coding for Efficiency, Portability, and Scalability*, chapter 1. John Wiley & Sons, 2006.
- [26] Edmund M. Clarke, E Allen Emerson, and A Prasad Sistla. Automatic verification of finite-state concurrent systems using temporal logic specifications. *ACM Transactions on Programming Languages and Systems (TOPLAS)*, 8(2):244–263, 1986.
- [27] David E Culler, Jaswinder Pal Singh, and Anoop Gupta. *Parallel Computer Architecture: A Hardware/Software Approach*, chapter 5, pages 260–262. Gulf Professional Publishing, 1999.
- [28] Mitch Dale. Utilizing clock-gating efficiency to reduce power. *EE Times, India*, 2008.

-
- [29] Gwenaël Delaval, Hervé Marchand, and Eric Rutten. Contracts for modular discrete controller synthesis. In *ACM Sigplan Notices*, volume 45, pages 57–66. ACM, 2010.
- [30] Gwenaël Delaval, Éric Rutten, and Hervé Marchand. Integrating discrete controller synthesis into a reactive programming language compiler. *Discrete Event Dynamic Systems*, 23(4):385–418, 2013.
- [31] JackB Dennis. *Data Flow Graphs*, page 513. Springer US, 2011.
- [32] Srinivas Devadas and Sharad Malik. A survey of optimization techniques targeting low power vlsi circuits. In *Proceedings of the 32nd annual ACM/IEEE Design Automation Conference*, pages 242–247. ACM, 1995.
- [33] Monica Donno, Alessandro Ivaldi, Luca Benini, and Enrico Macii. Clock-tree power optimization based on rtl clock-gating. In *Proceedings of the 40th annual Design Automation Conference*, pages 622–627. ACM, 2003.
- [34] Emil Dumitrescu, Alain Girault, Hervé Marchand, and Éric Rutten. Multicriteria optimal reconfiguration of fault-tolerant real-time tasks. *IFAC Proceedings Volumes*, 43(12):356–363, 2010.
- [35] Aniryudh Reddy Durgam and Ken Choi. Optimized clock gating cell for low power design in nanoscale cmos technology. In *Fifth Asia Symposium on Quality Electronic Design (ASQED 2013)*, pages 85–88. IEEE, 2013.
- [36] Bruno Dutertre. *Spécification et Preuve de Systemes Dynamiques*. PhD thesis, Rennes 1, 1992.
- [37] Maurizio Di Paolo Emilio. *Microelectronics: From Fundamentals to Applied Design*, page ix. Springer, 2015.
- [38] Cagkan Erbas. *System-level Modeling and Design Space Exploration for Multiprocessor Embedded System-on-Chip Architectures*, volume 132, page 1. Amsterdam University Press, 2007.
- [39] Yvan Eustache and Jean-Philippe Diguët. Specification and os-based implementation of self-adaptive, hardware/software embedded systems. In *Proceedings of the 6th IEEE/ACM/IFIP international conference on Hardware/Software codesign and system synthesis*, pages 67–72. ACM, 2008.

-
- [40] Tiziana Fanni, Lin Li, Timo Viitanen, Carlo Sau, Renjie Xie, Francesca Palumbo, Luigi Raffo, Heikki Huttunen, Jarmo Takala, and Shuvra S Bhattacharyya. Hardware design methodology using lightweight dataflow and its integration with low power techniques. *Journal of Systems Architecture*, 78:15–29, 2017.
- [41] Mohammed Ferdjallah. *Introduction to Digital Systems: Modeling, Synthesis, and Simulation Using VHDL*, chapter 9, pages 165–166. John Wiley & Sons, 2011.
- [42] Pascal Fradet, Alain Girault, Ruby Krishnaswamy, Xavier Nicollin, and Arash Shafiei. Rdf: Reconfigurable dataflow (extended version). Technical report, INRIA Grenoble - Rhône-Alpes, 2018.
- [43] Stephen B Furber, James D Garside, Peter Riocreux, Steven Temple, Paul Day, Jianwei Liu, and Nigel C Paver. Amulet2e: An asynchronous embedded controller. *Proceedings of the IEEE*, 87(2):243–256, 1999.
- [44] Daniel D Gajski, Samar Abdi, Andreas Gerstlauer, and Gunar Schirner. *Embedded System Design: Modeling, Synthesis and Verification*, chapter 1–2. Springer Science & Business Media, 2009.
- [45] Daniel D Gajski, Nikil Dutt, Allen Wu, and Steve Lin. High level synthesis, introduction to chip and system design, 1992.
- [46] Daniel D Gajski and Robert H. Kuhn. New vlsi tools. *Computer*, pages 11–14, 1983.
- [47] Govin Das Gautam, Shyam Akashe, and Sanjay Sharma. Transistor sizing for low power cmos circuits. *International Journal*, 1(1):37–59, 2011.
- [48] Frank Ghenassia et al. *Transaction-Level Modeling With SystemC*, volume 2, chapter 1–3. Springer, 2005.
- [49] KAHN Gilles. The semantics of a simple language for parallel programming. *Information processing*, 74:471–475, 1974.
- [50] Sébastien Guillet, Florent de Lamotte, Nicolas Le Griguer, Eric Rutten, Guy Gogniat, and Jean-Philippe Diguët. Designing formal reconfiguration control using uml/marte. In *7th International Workshop on Reconfigurable and Communication-Centric Systems-on-Chip (ReCoSoC)*, pages 1–8. IEEE, 2012.

-
- [51] K Hariharan and C Jaya Kumar. Clock gating for low power circuit design by merge and split methods. *IOSR Journal of Engineering*, 2(4):1–5, 2012.
- [52] David Harris and Sarah Harris. *Digital Design and Computer Architecture*, chapter 2–3. Morgan Kaufmann, 2010.
- [53] Yann Hietter, Jean-Marc Roussel, and Jean-Jacques Lesage. Algebraic synthesis of transition conditions of a state model. In *2008 9th International Workshop on Discrete Event Systems*, pages 187–192. IEEE, 2008.
- [54] Idress Husien, Nicolas Berthier, and Sven Schewe. A hot method for synthesising cool controllers. In *Proceedings of the 24th ACM SIGSOFT International SPIN Symposium on Model Checking of Software*, pages 122–131. ACM, 2017.
- [55] Hans M Jacobson, Prabhakar N Kudva, Pradip Bose, Peter W Cook, Stanley E Schuster, Eric G Mercer, and Chris J Myers. Synchronous interlocked pipelines. In *Proceedings Eighth International Symposium on Asynchronous Circuits and Systems*, pages 3–12. IEEE, 2002.
- [56] Ravindra Jejurikar, Cristiano Pereira, and Rajesh Gupta. Leakage aware dynamic voltage scaling for real-time embedded systems. In *Proceedings of the 41st annual Design Automation Conference*, pages 275–280. ACM, 2004.
- [57] Ahmed Jerraya and Wayne Wolf. *Multiprocessor Systems-on-Chips*, chapter 1–10, page 1. Elsevier, 2004.
- [58] Jagrit Kathuria, M Ayoubkhan, and Arti Noor. A review of clock gating techniques. *MIT International Journal of Electronics and Communication Engineering*, 1(2):106–114, 2011.
- [59] Krishna Kavali, S Rajendar, and R Naresh. Design of low power adaptive pulse triggered flip-flop using modified clock gating scheme at 90 nm technology. *Procedia Materials Science*, 10:323–330, 2015.
- [60] SV Lakshmi, PS Vishnu Priya, and Mrs S Prema. Performance comparison of various clock gating techniques. *IOSR Journal of VLSI and Signal Processing (IOSR-JVSP)*, 5(1):2319–4197, 2015.

- [61] Marco Lanuzza, Pasquale Corsonello, and Stefania Perri. Low-power level shifter for multi-supply voltage designs. *IEEE Transactions on Circuits and Systems II: Express Briefs*, 59(12):922–926, 2012.
- [62] Bilung Lee. *Fusing Dataflow with Finite State Machines*, chapter 1–2. Electronics Research Laboratory, College of Engineering, University of California Berkeley, 1996.
- [63] EA Lee. A denotational semantics for dataflow with firing, memorandum ucb/erlm97/3. *Electronics Research Laboratory, UC Berkeley*, 1997.
- [64] Edward A Lee and Thomas M Parks. Dataflow process networks. *Proceedings of the IEEE*, 83(5):773–801, 1995.
- [65] Li Li. *Power Optimization From Register Transfer Level To Transistor Level In Deeply Scaled CMOS Technology*, chapter 1–2. Illinois Institute of Technology, 2012.
- [66] Li Li, Ken Choi, and Haiqing Nan. Activity-driven fine-grained clock gating and run time power gating integration. *IEEE transactions on very large scale integration (VLSI) systems*, 21(8):1540–1544, 2013.
- [67] Jianfeng Liu, Mi-Suk Hong, Kyungtae Do, Jung Yun Choi, Jaehong Park, Mohit Kumar, Manish Kumar, Nikhil Tripathi, and Abhishek Ranjan. Clock domain crossing aware sequential clock gating. In *Proceedings of the 2015 Design, Automation & Test in Europe Conference & Exhibition*, pages 1–6. EDA Consortium, 2015.
- [68] Yan Luo, Jia Yu, Jun Yang, and Laxmi Bhuyan. Low power network processor design using clock gating. In *Proceedings of the 42nd annual Design Automation Conference*, pages 712–715. ACM, 2005.
- [69] Martina Maggio, Henry Hoffmann, Alessandro V Papadopoulos, Jacopo Panerati, Marco D Santambrogio, Anant Agarwal, and Alberto Leva. Comparison of decision-making strategies for self-optimization in autonomic computing systems. *ACM Transactions on Autonomous and Adaptive Systems (TAAS)*, 7(4):36, 2012.
- [70] Hamid Mahmoodi, Vishy Tirumalashetty, Matthew Cooke, and Kaushik Roy. Ultra low-power clocking scheme using energy recovery and clock gating. *IEEE transactions on very large scale integration (VLSI) systems*, 17(1):33–44, 2009.

- [71] Florence Maraninchi and Yann Rémond. Argos: an automaton-based synchronous language. *Computer languages*, 27(1-3):61–92, 2001.
- [72] Hervé Marchand, Patricia Bournai, Michel Le Borgne, and Paul Le Guernic. Synthesis of discrete-event controllers based on the signal environment. *Discrete Event Dynamic Systems*, 10(4):325–346, 2000.
- [73] Herve Marchand and Michel Le Borgne. On the optimal control of polynomial dynamical systems over z/pz . In *4th International Workshop on Discrete Event Systems*, pages 385–390, 1998.
- [74] Hervé Marchand and Mazen Samaan. Incremental design of a power transformer station controller using a controller synthesis methodology. *IEEE Transactions on Software Engineering*, 26(8):729–741, 2000.
- [75] Peter Marwedel. *Embedded System Design*, volume 1, chapter 2. Springer, 2006.
- [76] Uwe Meyer-Baese, A Vera, A Meyer-Baese, M Pattichis, and R Perry. Smart altera firmware for dsp with fpgas. In *Independent Component Analyses, Wavelets, Unsupervised Nano-Biomimetic Sensors, and Neural Networks V*, volume 6576, page 65760T. International Society for Optics and Photonics, 2007.
- [77] Sajed Miremadi, Bengt Lennartson, and Knut Akesson. A bdd-based approach for modeling plant and supervisor by extended finite automata. *IEEE Transactions on Control Systems Technology*, 20(6):1421–1435, 2011.
- [78] Gordon E Moore et al. Progress in digital integrated electronics. In *Electron Devices Meeting*, volume 21, pages 11–13, 1975.
- [79] Juanjo Noguera and Rosa M Badia. System-level power-performance trade-offs in task scheduling for dynamically reconfigurable architectures. In *Proceedings of the 2003 international conference on Compilers, architecture and synthesis for embedded systems*, pages 73–83. ACM, 2003.
- [80] Metė Özbaltan and Nicolas Berthier. Exercising symbolic discrete control for designing low-power hardware circuits: an application to clock-gating. *IFAC-PapersOnLine*, 51(7):120–126, 2018.

-
- [81] Samir Palnitkar. *Verilog HDL: A Guide to Digital Design and Synthesis*, volume 1, chapter 1–9,14. Prentice Hall Professional, 2003.
- [82] Thomas M Parks, José Luis Pino, and Edward A Lee. A comparison of synchronous and cycle-static dataflow. In *Conference Record of the Twenty-Ninth Asilomar Conference on Signals, Systems and Computers*, volume 1, pages 204–210. IEEE, 1995.
- [83] Sangeeta Parshionikar, Deepak V Bhoir, and Sapna Prabhu. Leakage power reduction using multi threshold voltage cmos technique. *International Journal of Scientific & Engineering Research*, 4(10), 2013.
- [84] Massoud Pedram and Qing Wu. Design considerations for battery-powered electronics. In *Proceedings 1999 Design Automation Conference (Cat. No. 99CH36361)*, pages 861–866. IEEE, 1999.
- [85] Volnei A Pedroni. *Circuit Design with VHDL*, chapter 1. MIT press, 2004.
- [86] Douglas L Perry. *VHDL: Programming by Example*, volume 4, chapter 1. McGraw-Hill New York, NY, USA, 2002.
- [87] Georgios Pouiklis and Georgios Ch Sirakoulis. Clock gating methodologies and tools: a survey. *International Journal of Circuit Theory and Applications*, 44(4):798–816, 2016.
- [88] Imran Rafiq Quadri, Huafeng Yu, Abdoulaye Gamatié, Samy Meftali, Jean-Luc Dekeyser, and Éric Rutten. Targeting reconfigurable fpga based socs using the marte uml profile: from high abstraction levels to code generation. *International Journal of Embedded Systems*, 2010.
- [89] Nithya Raghavan, Venkatesh Akella, and Smita Bakshi. Automatic insertion of gated clocks at register transfer level. In *Proceedings of the 12th International Conference on VLSI Design, VLSID '99*, pages 48–54. IEEE, 1999.
- [90] Peter JG Ramadge and W Murray Wonham. The control of discrete event systems. *Proceedings of the IEEE*, 77(1):81–98, 1989.
- [91] Irving S Reed and Gustave Solomon. Polynomial codes over certain finite fields. *Journal of the society for industrial and applied mathematics*, 8(2):300–304, 1960.

- [92] Dushyant Kumar Sharma. Effects of different clock gating techniques on design. *International Journal of Scientific & Engineering Research*, 3(5):1–4, 2012.
- [93] Richard Sharp. *Higher-Level Hardware Synthesis*, chapter 1–2. Springer, 2004.
- [94] Jitesh Shinde and SS Salankar. Clock gating—a power optimizing technique for vlsi circuits. In *2011 Annual IEEE India Conference*, pages 1–4. IEEE, 2011.
- [95] Kamana Sigdel. *System-Level Design Space Exploration of Reconfigurable Architectures*, chapter 1. PhD Thesis, Delft University of Technology, 2011.
- [96] Harpreet Singh and Sukhwinder Singh. A review on clock gating methodologies for power minimization in vlsi circuits. *Int. J. Sci. Engin. App. Sci*, 2, 2016.
- [97] Priya Singh and Ravi Goel. Clock gating: A comprehensive power optimization technique for sequential circuits. *International Journal of Advanced Research in Computer Science & Technology (IJARCST)*, 2(2):321–324, 2014.
- [98] Nandita Srinivasan, Navamitha S Prakash, D Shalakra, D Sivaranjani, B Bala Tripura Sundari, et al. Power reduction by clock gating technique. *Procedia Technology*, 21:631–635, 2015.
- [99] Antonio GM Strollo, Ettore Napoli, and Davide De Caro. New clock-gating techniques for low-power flip-flops. In *Proceedings of the 2000 international symposium on Low power electronics and design*, pages 114–119. ACM, 2000.
- [100] J Sudhakar, A Mallikarjuna Prasad, and Ajit Kumar Panda. Gfcg: Glitch free combinational clock gating approach in nanometer vlsi circuits. In *2015 2nd International Conference on Electronics and Communication Systems (ICECS)*, pages 146–150. IEEE, 2015.
- [101] BL Theraja and AK Theraja. *Textbook of Electrical Technology*, volume 4, chapter 67, pages 2471–2478. S. Chand, 2005.
- [102] Donald E Thomas, Elizabeth D Lagnese, Robert A Walker, Jayanth V Rajan, Robert L Blackburn, and John A Nestor. *Algorithmic and Register-Transfer Level Synthesis: The System Architect’s Workbench: The System Architect’s Workbench*, volume 85, chapter 1,2. Springer Science & Business Media, 1989.

-
- [103] Jason G Tong, Ian DL Anderson, and Mohammed AS Khalid. Soft-core processors for embedded systems. In *2006 International Conference on Microelectronics*, pages 170–173. IEEE, 2006.
- [104] Lionel Torres, Pascal Benoit, Gilles Sassatelli, Michel Robert, Fabien Clermidy, and Diego Puschini. An introduction to multi-core system on chip – trends and challenges. In *Multiprocessor System-on-Chip*, pages 1–21. Springer, 2011.
- [105] Stavros Tripakis, Rhishikesh Limaye, Kaushik Ravindran, Guoqiang Wang, Hugo Andrade, and Arkadeb Ghosal. Tokens vs. signals: On conformance between formal models of dataflow and hardware. *Journal of Signal Processing Systems*, 85(1):23–43, 2016.
- [106] Yan Zhang, Jussi Roivainen, and Aarne Mammela. Clock-gating in fpgas: A novel and comparative evaluation. In *9th EUROMICRO Conference on Digital System Design (DSD'06)*, pages 584–590. IEEE, 2006.