Use of High-Level Design Information for Enabling Automation of Fine-Grained Power Gating

by

Abhinav Agarwal

Submitted to the Department of Electrical Engineering and Computer Science
in partial fulfillment of the requirements for the degree of

Doctor of Philosophy

at the

MASSACHUSETTS INSTITUTE OF TECHNOLOGY

September 2014

Signature redacted

Author .....
Department of Electrical Engineering and Computer Science
August 7, 2014

Signature redacted

Certified by ....
Arvind
Johnson Professor of Computer Science and Engineering
Thesis Supervisor

Signature redacted

Accepted by .....
Professor Leslie A. Kolodziejski
Chair, Department Committee on Graduate Students

Use of High-Level Design Information for Enabling Automation of Fine-Grained Power Gating

by

Abhinav Agarwal

Submitted to the Department of Electrical Engineering and Computer Science
on August 7, 2014, in partial fulfillment of the
requirements for the degree of
Doctor of Philosophy

Abstract

Leakage power reduction through power gating requires considerable design and verification effort. Conventionally, extensive analysis is required for dividing a heterogeneous design into power domains and generating control signals for power switches because of the need to preserve design behavior. In this thesis, I present a scheme which uses high-level design description to automatically generate a collection of fine-grain power domains and associated control signals. For this purpose, we explore the field of high-level design languages and select rule-based languages on Quality-of-Result metrics.

We provide algorithms to enable automatic power domain partitioning in designs generated using rule-based languages. We also describe techniques for collecting the dynamic activity characteristics of a domain, viz. *total inactivity* and *frequency of inactive-active transitions*. These metrics are necessary to decide the generated domains' viability for power gating after accounting for energy loss due to transitions. Our automated power gating technique provides power savings without exacerbating the verification problem because the power domains are correct by construction. We illustrate our technique using various test-cases: two wireless decoder designs, a million-point sparse FFT design and a RISC processor design.

Thesis Supervisor: Arvind
Title: Johnson Professor of Computer Science and Engineering

Acknowledgments

As the time that I have spent at MIT working towards my PhD draws to a close, I owe a deep sense of gratitude to a lot of people who have helped me in this journey.

The first person I wish to thank would be my research advisor, Prof. Arvind. It has been an absolute pleasure to work with him and learn from his deep knowledge and experience. Over the years, I have had the privilege of working with him on a multitude of projects, teaching graduate and undergraduate classes, writing proposals and making research presentations. He gave me the freedom to explore ideas that aim to bring together disparate fields, and to create systems of surprising complexity. At the same time, I learned how to focus on crystallizing my work to fundamental concepts that can be applied in new applications. I consider Prof. Arvind as a role model for his incredible thirst for new ideas, for his positive outlook and for the personal care that he provides to everyone around him. It has been a great honor to be a part of his research group.

I thank Prof. Anantha Chandrakasan for his guidance to me when I entered MIT. The support he provided during my fellowship was valuable to me in my search for a research group. During the course of my thesis, Prof. Chandrakasan's guidance regarding use of MTL tools and technology libraries have been very helpful. I would also like to thank Prof. Li-Shiuan Peh for her help in framing the questions explored in my thesis. I am highly grateful to Prof. Chandrakasan and Prof. Peh for being a part of my thesis committee and guiding me towards the completion of my PhD.

At MIT, I got an opportunity to interact and work with several other distinguished researchers and faculty members. It was a great experience being part of the teaching staff for the first edition of an undergraduate computer architecture course taught by Prof. Arvind and Prof. Joel Emer. I also collaborated with Prof. Emer while conducting the design contest as part of the 2010 MEMOCODE conference. Working with Prof. Dina Katabi during the course of the Sparse FFT project provided me with great learning opportunities in algorithms and digital signal processing. It was an honor to work next door to offices of faculty members like Prof. Jack Dennis, Prof.

Daniel Sanchez and Prof. Srini Devadas, with whom I had several valuable discussions about research ideas. I am grateful to Prof. Anant Agarwal and Prof. Chandrakasan for evaluating my research qualification exam. I would like to thank my academic advisor, Prof. Harry Lee for guiding me through the academic requirements at MIT.

A special note of gratitude to the MIT EECS graduate administrator, Janet Fischer, and our research group's administrative assistant, Sally Lee, who were always ready to quickly provide countless employer letters for immigration and travel documents.

Several ideas contained in this thesis originated from projects that I started during my three research internships at Qualcomm, Inc., in San Diego, CA. I would like to thank my supervisor at Qualcomm, Dr. Suhas Pai, for giving me the opportunity and encouragement to pursue these projects. During these internships, I worked with fellow graduate students, Afred Ng and Joonsoo Kim. It was a great experience to discuss and experiment with various project ideas with them. I would also like to acknowledge the extensive assistance provided by Bluespec Inc. during our work to introduce use of BSV at Qualcomm, in particular the help given by Dr. Rishiyur Nikhil, Dr. Don Baltus and Steve Allen.

I would like to thank all the graduate students of the Computation Structures Group. First one on this list has to be Alfred Ng, who took me under his wing and helped me with my first project at MIT, as well as at Qualcomm. We were office-mates and apartment-mates, and had discussions on every topic under the sun. Kermin Elliott Fleming and Nirav Dave, both worked with me on several projects and the key element of this thesis was crystallized from my discussions with them. Michael Pellauer, Myron King, Asif Khan and Murali Vijayaraghavan, I have great memories of working on various MEMOCODE design contests will all of them. Conducting the 2010 contest with Michael was a great learning experience. The long hours spent debugging FPGA platforms with Myron, Asif, Richard Uhler and Oriol Arcas, became fun due to the constant jokes and pranks shared by everyone. Murali has been the lab-mate with whom I have shared most of my time at MIT. I was able to learn a lot from my discussions and arguments with him, and I am sure I am going to miss all

of them when I leave. I would also like to acknowledge the more recent members of our group, Sang Woo, Ming Liu, Andy Wright, Sizhuo Zhang, Shuo Tao and Sungjin Lee. It was great fun to go on lab outings with all the CSG members, and Prof. Arvind's passion for hiking has definitely been inculcated in all of us. A word of thanks for students from Prof. Katabi's group, especially Haitham Hassanieh, with whom I had extensive discussions in the Sparse FFT project. I want to thank my friends in Cambridge who were part of weekly outings and dinners, including Tushar Krishna, Harshad Kasture, Hemant Bajpai, Vibhor Jain, Amit Soni and Siddharth Bhardwaj.

I will be forever grateful to my parents, for their support and encouragement to me throughout my education. My parents have shown incredible patience and given me absolute freedom as I pursued my dream for a PHD from MIT. I want to thank my sister, Isha Agarwal, for her affectionate support and motivation given to me. Finally, I would thank my wife, Anushree Kamath, for being with me and guiding me through the ups and downs of graduate student life during the last few years. It has been a wonderful experience to share the joys and struggles of academic life with her, and the support of a spouse who has 'been there and done that' was invaluable.

Contents

1 Introduction **17**

1.1 Trends in power consumption of digital hardware designs 17

1.1.1 Sub-threshold leakage current 18

1.2 Power management in hardware designs 21

1.2.1 Power gating 21

1.2.2 Motivation for fine-grained power gating 22

1.2.3 Design cost of power gating 23

1.3 High-level hardware design 24

1.3.1 Quality of hardware 26

1.3.2 Ability to leverage rule-based designs 27

1.4 Contributions 28

1.5 Thesis organization 29

2 Related work **31**

2.1 Voltage islands and circuit techniques 31

2.2 Operand Isolation 32

2.3 Automation of power gating 33

2.4 Netlist-based partitioning techniques 34

2.5 High-level designs 34

3 Power gating and partitioning **37**

3.1 Introduction 37

3.2 Power domain partitioning 39

3.2.1 Partitioning technique for explicit-control circuits 39
3.2.2 Partitioning algorithm as graph-coloring 40
3.2.3 Example for generation of gating conditions 42
3.2.4 Nested domains 42
3.3 General partitioning strategies 43

4 Use of rule-based designs for power gating **51**
4.1 Hardware compilation from rule-based designs 51
4.1.1 Scheduling logic in rule-based designs 52
4.1.2 Example of hardware generation from BSV 53
4.2 Selection of power domain logic 55
4.3 Generating UPF/CPF description 57
4.4 Collecting statistical information 58

5 Dynamic Activity Metrics **61**
5.1 Metrics for the Reed-Solomon decoder 64
5.2 Importance of macro factors on activity 68
5.3 Metrics for the Viterbi decoder 70

6 Larger and complex test cases **73**
6.1 Million-point SFFT design 73
6.1.1 Design overview 73
6.1.2 Design Architecture 75
6.1.3 SFFT design activity metrics 85
6.2 RISC processor design 88
6.2.1 Design overview 88
6.2.2 RISC processor design activity metrics 89

7 Breakeven threshold and impact of fine-grained gating **93**
7.1 Leakage power consumption 93
7.2 Breakeven threshold of inactivity interval 94
7.3 Performance Impact 96

7.4 Qualifying domains for the design testcases 97
7.5 Energy Savings 98
7.5.1 Cycle by cycle profile 98
7.5.2 Net reduction in power consumption 98

8 Comparison of high-level languages for hardware design 101
8.1 Software-based design tools 102
8.1.1 Relationship between C-based design and generated hardware 102
8.2 The Application: Reed-Solomon Decoder 104
8.2.1 Decoding Process 104
8.3 Generating hardware from C/C++ 105
8.3.1 The Initial Design 105
8.3.2 Loop unrolling to increase parallelism 105
8.3.3 Expressing producer-consumer relationships 106
8.3.4 Issues in Streaming Conditionals 108
8.3.5 Fine-grained processing 110
8.4 Implementation in rule-based design languages 111
8.4.1 Initial design 111
8.4.2 Design Refinements 112
8.5 Area and performance results 113
8.6 Application of power gating techniques in C-based design 115

9 Conclusions 117
9.1 Thesis Summary 117
9.2 Future extensions 119
9.2.1 Automatic gating of local state 120
9.2.2 Allowing turn on over multiple cycles 120
9.2.3 Extending to module level gating 121
9.2.4 Integrated power management 123

List of Figures

1-1 Erstwhile projections of total chip leakage and active power consumption with technology scaling, showing increase in power consumption with leakage increasing at a faster rate [20] 19

1-2 Projected ratio of total chip and unit gate leakage to active power consumption with more recent technology scaling, leakage power per unit gate stays roughly the same while active power per gate decreases, leading to increased ratio of total chip leakage [64]. Various power saving techniques have been employed to combat these projected increases in current technologies. 19

1-3 Measured increase in leakage currents of IBM's recent High-Performance (HP) and Low-Power (LP) technology cell libraries [67] 20

1-4 Insertion of header switches in a partitioned design for power gating . 22

1-5 Example BSV design - GCD 28

3-1 Nodes of Explicit-Control circuits 40

3-2 Associating activity conditions with logic blocks. Control signals are marked in red. 43

3-3 Hardware generated for GCD RTL design, control signals are marked in red. Details of control module have been omitted because it is kept ungated. Fine-grained power gating is applied to the computational unit for subtraction in the datapath module. 45

3-4 Top-level Verilog RTL design of GCD with separated control and datapath modules 45

3-5 Verilog RTL design of GCD datapath where signals for gating are directly identifiable. Details of control module code has not been shown here. Each combinational unit is contained in a leaf module instantiated in the datapath module. The control signals used for mux selection can be identified due to explicit mux structures being declared in the design. The output of subtractor is only used as an input for the 3-input mux and so the mux selector can be used for gating the subtractor. 46

3-6 Flattened Verilog RTL design for GCD, where it is difficult to identify control signals for gating combinational logic. 47

4-1 Generation of rule firing signals ϕ_i by the scheduler 53

4-2 Conceptual design of a module in BSV 54

4-3 Hardware generated for example BSV module with state updates selected by the rule firing control signals 54

4-4 Power domain partitions and associated gating conditions for example design 55

4-5 Application of partitioning algorithm to rule-based designs 56

4-6 CPF Power specification code segment for the example design 57

4-7 Instrumentation of the example BSV module to collect activity statistics for each rule in the design. 59

5-1 Dynamic Activity profiles 62

5-2 Architecture of Reed-Solomon Decoder 64

5-3 Hierarchical activity considerations for logic blocks within wireless decoders 69

5-4 Architecture of Viterbi Decoder and testbench 70

6-1 Stages of the SFFT algorithm. Core stages are highlighted in blue. . 75

6-2 Modules implementing the SFFT Core 76

6-3 Single parameterized block of streaming FFT 78

6-4 Definition of butterfly structures in R2^{2}SDF design 78
6-5 Architecture of the 4096-point dense FFT 79
6-6 Architecture of the Max Selector module 80
6-7 Architecture of the Voter module implemented as a series of filters. Large buckets for each iteration have been highlighted in blue. 81
6-8 Architecture of the Value Compute module 82
6-9 Architecture of RISC processor 89

8-1 Simple streaming example - source code 107
8-2 Simple streaming example - hardware output 107
8-3 Complex streaming example - Source code 108
8-4 Complex streaming example - Hardware output 108
8-5 Forney's Algorithm Implementation 109
8-6 Bluespec interface for the decoder 111
8-7 Initial version of compute-syndrome rule 112
8-8 Parameterized parallel version of the compute-syndrome rule 113

9-1 Generating module-level gating condition for modules containing no externally read state 122

List of Tables

3.1 Activity conditions for logic gates 42

5.1 Reed-Solomon Activity Metrics: Input data with maximal correctable errors 65

5.2 Reed-Solomon Activity Metrics: Input data with no errors 66

5.3 Reed-Solomon Module-level Inactivity percentage 67

5.4 Viterbi Activity Metrics 71

6.1 Parameters for SFFT Core implementation 76

6.2 FPGA Resource utilization, shown as a percentage of total resources available on Xilinx FPGA XC6VLX240T. 84

6.3 Latency and throughput in FPGA clock cycles 84

6.4 Processing times in milliseconds accounting for the maximum operating frequency of the component. 85

6.5 SFFT Activity Metrics 87

6.6 SFFT Module-level Inactivity percentage 88

6.7 RISC Processor Activity Metrics - part 1 90

6.8 RISC Processor Activity Metrics - part 2 91

6.9 RISC Processor Module-level Inactivity percentage 92

7.1 Power consumption breakdown of Reed-Solomon decoder, Viterbi decoder and SFFT design. The synthesis was done using Nangate 45nm library with the operating frequency set to 100MHz. Power consumption data was obtained using simulation of the placed and routed designs. 94

7.2 Computing breakeven threshold of inactivity interval, and the performance impact for gating typical logic blocks 95

8.1 Comparison of Source code size, FPGA resources and performance . 114

Chapter 1

Introduction

Leakage current dissipation in hardware designs has been increasing with technology scaling. At the same time, power consumption has become increasingly important for electronic applications running on wireless and hand-held devices, due to constraints on the size and weight of batteries, and thermal budgets. This thesis provides a solution to this problem by providing techniques that enable automatic power gating of heterogeneous hardware designs. These techniques rely on high-level design information that is available in rule-based hardware design descriptions.

1.1 Trends in power consumption of digital hardware designs

Power consumption has become a first-order design concern for contemporary hardware designs. With designs having billions of transistors due to scaling trends seen under Moore's law, the thermal and power budgets now control how many of these transistors can operate concurrently, which in turn affects the performance. At the same time, proliferation of battery-operated wireless consumer devices has given rise to an extreme focus on extending dynamic and standby operation times, and in turn optimizing designs for reduced power consumption. Handheld mobile devices have a strict heat budget as well to account for tolerable heat dissipated to the user,

and this in turn places another total power dissipation limit on the devices. Such wireless devices usually have sporadic and data-dependent activity, with most of the operational time spent in quiescent states. This emphasizes the designers to carefully consider the ratio of expected static (leakage) to dynamic (active) power consumption under typical use scenarios.

There have been various projections [17, 20, 46] of the increase in chip power with technology scaling. In particular, the fraction of leakage power is projected to increase dramatically. While the active power consumption has been controlled due to reduced supply voltage with scaling, reduction of threshold voltage of CMOS devices to maintain performance has increased the leakage currents in newer technologies. By one projection [64], a design at 130 nm with equal fractions of static and dynamic power, when scaled to 32 nm and beyond would lead to leakage dissipation being the dominant component of power consumption. Figures 1-1 and 1-2 show some of the earlier and more recent projections of the increase in total chip leakage current dissipation and in the fractional leakage power with scaling of technology nodes. These projections were quite aggressive in predicting the rise in leakage power. The actual measured leakage currents in both high-performance and low-power technologies have also been increasing with technology scaling as seen in Figure 1-3, though at a reduced rate because of several power saving techniques including use of low-leakage devices and use of power gating. However, application of such techniques comes with a large design effort and verification cost.

1.1.1 Sub-threshold leakage current

Leakage current dissipation refers to the current flowing through CMOS devices when they are not turned on. In this work, we will primarily focus on the sub-threshold drain-to-source leakage current I_{DS}, which is dissipated when the gate to source voltage bias (V_{GS}) is less than V_{th}. When the transistor is operating in this sub-threshold region, I_{DS} is approximated [61] by equation 1.1 as defined in the Berkeley Short-channel IGFET Model technical manual [58].

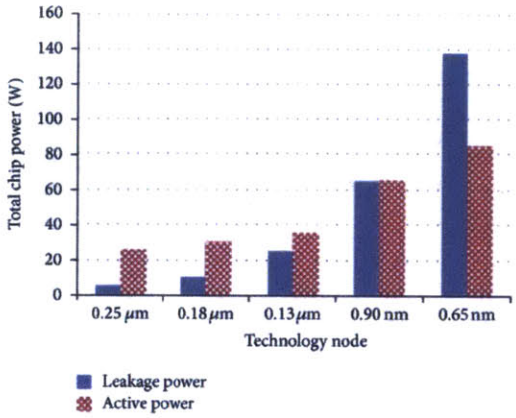


Figure 1-1: Erstwhile projections of total chip leakage and active power consumption with technology scaling, showing increase in power consumption with leakage increasing at a faster rate [20]

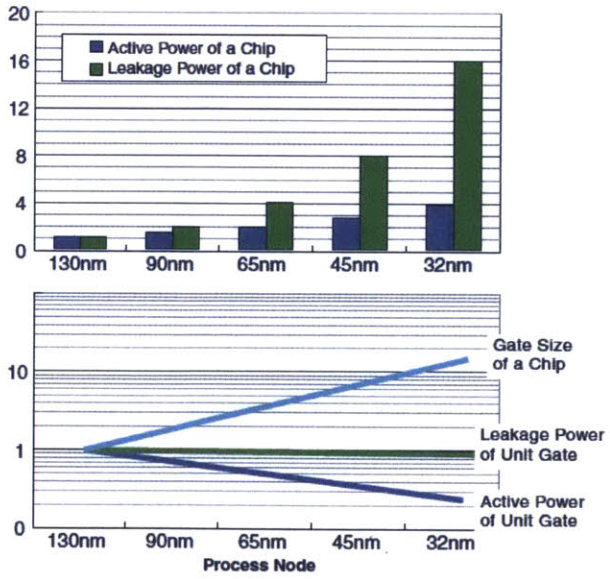


Figure 1-2: Projected ratio of total chip and unit gate leakage to active power consumption with more recent technology scaling, leakage power per unit gate stays roughly the same while active power per gate decreases, leading to increased ratio of total chip leakage [64]. Various power saving techniques have been employed to combat these projected increases in current technologies.

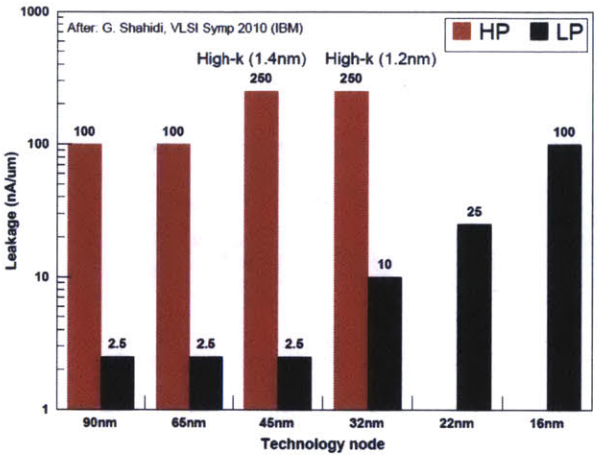


Figure 1-3: Measured increase in leakage currents of IBM's recent High-Performance (HP) and Low-Power (LP) technology cell libraries [67]

$$I_{DS} = I_0(1 - e^{-\frac{V_{DS}}{V_T}})e^{(\frac{V_{GS}-V_{th}-V'_{off}}{nV_T})} \tag{1.1}$$

where

$$I_0 = m\frac{W}{L}{V_T}^2 \tag{1.2}$$

V_T is the thermal voltage which is linearly dependent on temperature. $V'_{off}(= V_{OFF} + V_{OFFL}/L_{eff})$ is the offset voltage, which determines the channel current at $V_{GS} = 0$. W (channel width) and L (channel length) are design parameters, while m and n (sub-threshold swing) are process constants. When the transistor is switched off, with $V_{GS} = 0$, and $V_{DS} = V_{DD}$, which is much greater than the thermal voltage V_T, the sub-threshold current I_{DS} is directly proportional to $e^{(-V_{th})}$. In other words, the sub-threshold current decreases exponentially with increase in the threshold voltage V_{th}. With technology scaling, V_{DD} is reduced to decrease overall power consumption. To maintain performance, by keeping I_{DS} current high during transistor-on, V_{th} is reduced and this leads to an increase in the leakage current. In most techniques for leakage reduction, a transistor with higher threshold voltage (either dynamically changed by biasing or permanently increased by different doping) is inserted in the

circuit path from supply voltage to ground.

1.2 Power management in hardware designs

Current designs have been successful in some mitigation of the dramatic increase in power consumption in general, and leakage power in particular, through the use of several power saving techniques. Dynamic power consumption has been very effectively controlled using Clock gating and Dynamic Voltage and Frequency Scaling (DVFS). Clock gating involves turning off the clock to state elements which are not being updated in a given cycle. DVFS, on the other hand, changes the supply voltage and frequency of operation in proportion to the expected workload and performance. For reducing static power consumption, there are various techniques such as the use of high-threshold devices [14], reverse body biasing [82] and power gating [54].

1.2.1 Power gating

The most effective method of reducing the overall leakage current dissipation of a design is using *Power gating*. Power gating, in the simplest definition, is turning off the current supply to the circuit which is not being used. This is done by adding a control switch, which itself is a low-leakage device, that shuts off current supply based on a control signal. Such switches can be PMOS headers or NMOS footers depending on their placement at the supply voltage or ground terminals. A power controller generates the control signals fed to the power switches and is responsible for turning on the power domain, controlled by the switch, when needed. Power gating is implemented typically through the use of Multi-Threshold CMOS (MTCMOS) technology where switches with high V_{th} are inserted to control supply of current to logic with nominal V_{th}. Figure 1-4 shows a design with two power domains each controlled by header switches. The effective supply voltage for the power domains is denoted by VDD_{eff}, which can differ between domains depending on the size of the switch and amount of current drawn from the supply. To prevent spurious values in signals generated from logic in inactive power domains, isolation logic is inserted

between domains which holds these signal values to predetermined voltages. This work focuses on how to partition designs and identify appropriate control signals for each domain, using high-level design information.

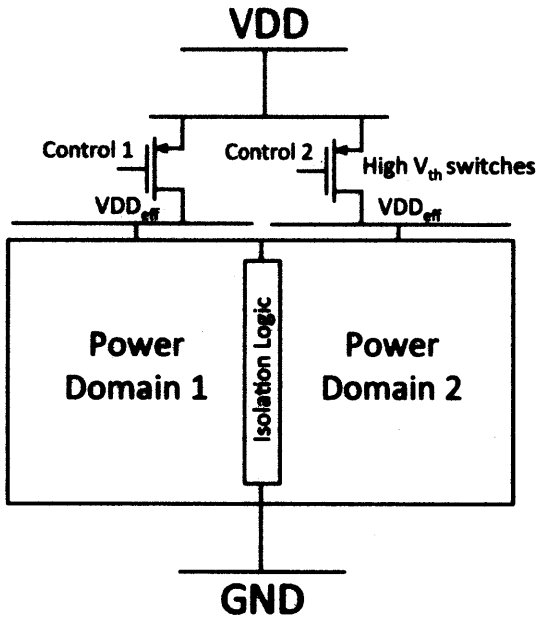


Figure 1-4: Insertion of header switches in a partitioned design for power gating

1.2.2 Motivation for fine-grained power gating

It is relatively straightforward to implement power gating in a broad manner, for example, by turning off processor cores in a multi-core chip depending on processing workload as measured in number of independent threads [30, 47, 63, 78]. However, coarse-grained power gating, even at a modular level, fails to exploit the situations where the entire module is not active at the same time. There is a frequent need for a much more customized application of the technique. Ideally, a designer would want to have exactly those transistors or logic cells turned on each cycle which are directly involved in computing state updates. Such a design would require a power switch for every logic cell, increasing the area of the power gated design by a large factor. In this work, we target a middle path with fine-grained sub-modular logic blocks gated independently, where the granularity is determined by the high-level

design, as discussed later. Having large coarse-grained power domains increases the average activity rate of such domains. By breaking up such a domain into fine-grained domains, we can have a larger fraction of the inactive logic available for power gating.

Consider a design implementing a Finite State Machine (FSM) where depending on the input data, various stages are activated for processing and computation. In such a design, we would want a power domain for each stage with independent control signals such that only those stages are turned on which are needed. Such data-dependent behavior is seen especially in various component blocks of wireless transceivers. As an example, consider error correction modules, where the level of computation activity is dependent on the amount of errors seen in input data, itself determined by the wireless channel's signal to noise ratio (SNR). In this work, we will explore the use of fine-grained power gating for such hardware designs. Decreasing the size of the power domains allows the domains to turn-on much faster as well. As we demonstrate, such domains can even transition between activity states within a single cycle for moderate clock frequencies. This allows fine-grained gating in the time domain as well.

1.2.3 Design cost of power gating

Though power gating allows significant savings in leakage, it also has associated costs for increased chip area for power switches, isolation logic and gating control signals, performance costs due to lower effective supply voltage, increased supply noise, and energy costs due to activation charging. By aiming for sub-modular granularity we balance the area costs with increased power savings as compared to the extremes of single logic-cell domains or large multi-module domains. Performance costs can be mitigated by having customizable gating definition, where the designer can let the critical path stay ungated. To compensate for increased noise in gated designs due to switching and in-rush currents, existing techniques like re-routable network of decoupling capacitances [44, 81] can be used. Finally, improperly chosen power domains could result in net increased power consumption due to frequent switching between inactive and active stages, each time incurring a charging energy cost for the

domain's effective supply line. We demonstrate a technique for automatic collection of appropriate activity metrics by simulation of high-level representative testbenches, and the use of these metrics, along with domain-level breakeven thresholds, to allow a designer to select viable power domains.

In addition, power gating of large heterogeneous designs has a significant design and verification cost, which is exacerbated in fine-grained application of the technique. Fine-grained power domains are not only smaller in area, and thus more numerous for a given design, they can also be turned on and off more quickly. Manual application of such a gating methodology requires insertion of control logic and verification to ensure that only under appropriate conditions inactive logic is being switched off. Since power gating is inserted at a late stage of the design process, any significant change in design functionality due to its insertion would require considerable time and effort for design validation. The ideal scenario is for such gating to happen automatically in a push-button manner with the amount of change in design functionality being minimal to none. Thus, there is a clear need for techniques to automate fine-grained power gating and as we will demonstrate, high-level hardware design has the potential to provide solutions in this space.

To reduce the verification burden, we will ensure that the gated design is bit and cycle accurate to the ungated design for all state elements of the design. Our gating technique will focus on switching off only the combinational logic in designs, aiming at a sub-modular block-level granularity. With regards to power gating of state elements, additional analysis is required to ensure that useful state, which might be read in future, is not lost. We will explore some possibilities in this area towards the end of the thesis in Chapter 9.

1.3 High-level hardware design

Automation of digital hardware design has been a long standing goal which has advanced greatly in recent years. The semiconductor industry has concurrent goals of increased designer productivity, shorter design cycles and ambitious area-performance-

power constraints. As a consequence, designers are increasingly adopting high-level Hardware Description Languages [7, 9, 15, 16, 19, 52, 77] to address the challenging design requirements of large systems without losing productivity. These languages provide a higher level of abstraction than traditional languages such as VHDL or Verilog, implicitly hiding some of the low-level details in favor of an explicit system-wide view.

A lot of these tools synthesize hardware from C, C++ [15] or SystemC [52] algorithmic descriptions. This approach helps the designer by raising the level of abstraction and allowing design iterations to happen at the algorithmic level, with hardware implications handled by the design tools. Compiler techniques aim to ensure that the produced circuit emulates the behavior of the abstract program. This process should allow increased re-use of designs, plug-and-play synthesis for new algorithms and the ability to produce large and complex hardware systems. However, the fact that the languages used by Electronic System-Level (ESL) design tools were not originally designed to describe hardware also prevent obtaining an architectural view of the system: the advanced techniques used to transform C/C++ code into circuits make it difficult to establish relations between the original model and the resulting hardware. There is a gap between the system specification and resulting hardware which can be difficult to bridge while starting from a software-based description.

On the other hand, there exist languages specifically designed for hardware specifications, called Hardware Description Languages (HDLs). These include basic languages like Verilog and VHDL, as well as high-level rule-based languages like Bluespec [9]. Due to the direct relationship of specifications to the micro-architecture, such languages can aid automated tools in design partitioning and identification of relevant control signals needed for power gating. The electronic design industry is increasingly adopting such languages and synthesis tools [6, 7, 9, 15, 52] to address the challenging design requirements of resource constraints, higher performance requirements and minimized power and thermal budgets. Use of high-level languages for designing custom hardware is gaining popularity since they increase design productivity by providing higher abstractions, design modularity, pasteurization, and

significant increase in code reuse. High-level languages facilitate design validation and verification, by allowing generation of designs that are correct-by-construction. Use of formal analysis for verification is also enabled by such design languages.

To enable greater applicability of fine-grained gating in special purpose hardware, the use of high-level hardware design languages significantly eases the design and verification problems. Fundamentally, fine-grained gating requires the algorithmic design intent to be available for power domain partitioning. The higher the abstraction used for designing hardware, the easier it is to extract such design intent. More concretely, individual domains can be defined as centers of logic activity used together as a unit, and hence comprise of design blocks that get activated and de-activated simultaneously. Such domains can be identified from the high-level description. Control signals used for activating such units can also be present in high-level description, and then can be used in the power management. This high-level information can be traced through various synthesis steps, and this is how fine-grained gating is enabled through our power domain partitioning technique.

1.3.1 Quality of hardware

One important question to explore is the impact of the use of high-level design languages on the quality of hardware generated, with respect to increased resource consumption or decrease in system performance. Power savings through fine-grained gating of high-level design could possibly be overwhelmed if there are significant area and performance costs to using high-level languages for designing hardware. In this thesis, we examine this question by comparing software-based design languages to rule-based languages. We also compare the high-level designs to baseline hand-coded RTL designs to verify if there is a penalty in terms of area or performance. We also determine common design constructs that are required for designing efficient hardware for typical heterogeneous designs such as digital signal processing accelerators, and whether such constructs can be efficiently expressed in the high-level designs.

1.3.2 Ability to leverage rule-based designs

One particular high-level HDL, Bluespec SystemVerilog (BSV) [9] uses a rule-based design methodology. Bluespec provides hardware designers with high-level language abstractions and features that can help them to quickly and easily express the micro-architecture they want to generate. Bluespec also facilitates latency insensitive designs by automatically generating handshaking signals between interface methods used to communicate between modules. This allows designers to incrementally refine modules to meet their requirements. Among the examples of designs using the Bluespec design methodology, Dave et al. [25] showed the use of single parameterized design source code to quickly generate Fast Frequency Transform (FFT) blocks with different architectures for different area and performance constraints. Similarly, Ng et al. [56] developed wireless transceiver modules using Bluespec that can be reused across different Orthogonal Frequency Division Multiplexing (OFDM) protocols.

In BSV, the hardware design is expressed in terms of state and module interface definitions with conditional *rules* computing state updates. Figure 1-5 shows the definition of an example module in BSV which computes the Greatest Common Divisor of two 32-bit inputs using Euclid's swap and subtract algorithm.

The design has two register state elements x and y, and two rules *swap* and *subtract* that compute updates to these registers. The input-output methods *start* and *result* are declared separately as the definition of the module interface $InterfGCD$, and themselves defined in the module body. Each rule and method can have an explicit guard condition specified after the name declaration. In addition, there can be implicit guard conditions due to invocation of methods within rule and method bodies. The Bluespec compiler automatically generates control logic for the hardware design using rule and interface method definitions and all guard conditions. Scheduling logic is generated to determine which of rules will be activated in a given cycle. This control logic plays a crucial part in our partitioning methodology, allowing automation of several steps involved with power gating.

As we will demonstrate, rule-based design languages occupy a sweet spot in terms

```
module mkGCD (InterfGCD);
   // state definitions

   Reg#(Bit#(32)) x <- mkReg(0);
   Reg#(Bit#(32)) y <- mkReg(0);

   // rule definitions
   rule swap (x > y && y > 0);
      x <= y;
      y <= x;
   endrule

   rule subtract (x <= y && y > 0);
      y <= y - x;
   endrule

   // method definitions
   method Action start(Bit#(32) a, Bit#(32) b) if (y == 0);
      x <= a;
      y <= b;
   endmethod

   method Bit#(32) result() if (y == 0);
      return x;
   endmethod
endmodule
```

Figure 1-5: Example BSV design - GCD

of raising the abstraction of design description high enough while keeping the resultant hardware quality within reasonable bounds. Hardware designers can keep the algorithmic design intent apparent in the high-level rule description, allowing downstream tools to use this description for optimizing various hardware metrics including area and performance, and even leakage power. *Static power reduction of such rule-based designs is the focus of the work described in this thesis.*

1.4 Contributions

In this work we explore the use of rule-based high-level HDLs for designing fine-grained gating in custom hardware. We evaluate the Quality-of-Results for hardware generated using BSV and compare them to designs generated through software-based design methodology. After establishing clear benefits of using rule-based HDLs, we then demonstrate how such designs can be automatically analyzed and partitioned

for reducing their leakage power dissipation.

The main contributions of this thesis are:

1. A novel technique that uses rule-based design descriptions for automatic partitioning of a digital design into power domains and associated gating signals.

2. Automatic classification of power domains for their suitability for power gating based on their dynamic characteristics.

3. We illustrate our technique using concrete design studies: two high-performance wireless decoders, 32-bit processor that boots Linux and a million-point Sparse Fourier transform design. These examples will show that unlike global power controllers, our technique introduces no new logic or state to generate power gating control signals.

4. We discuss issues related to fine-grained partitioning of hardware starting from Verilog RTL or from C-based high-level designs. This thesis provides reasoning for why rule-based synthesis is the appropriate choice for identifying high-level design information that can be used in power gating.

1.5 Thesis organization

In this chapter, we have discussed the fundamentals of power gating and high-level design that serves as background for the thesis. The rest of this thesis is organized as follows. In Chapter 2, we discuss related work in the area of design partitioning and power gating. In Chapter 3, we introduce and discuss the power domain partitioning problem. We provide a general technique for generating power domains in explicit control circuits. In Chapter 4, we elaborate on how use of rule-based designs eases the partitioning problem and collection of statistical information used to classify domains. Chapter 5 introduces two dynamic activity metrics used for selecting viable domains. We use realistic wireless decoder designs to demonstrate collection and use of activity metrics. We also discuss the need for detailed high-level testbenches to accurately characterize dynamic activity metrics.

In Chapter 6, we describe two large complex designs – a million-point sparse Fourier transform accelerator and a RISC processor that boots Linux, and discuss their fine-grained activity metrics. Chapter 7 quantifies the power and performance impact of our technique. In Chapter 8 we contrast and compare rule-based HDLs with $C/C++$ based design methodologies for area and performance design metrics. Finally, we present a summary of the thesis in Chapter 9, and propose future extensions and research directions for the discussed work.

Chapter 2

Related work

Leakage power reduction is fundamentally linked with trade-offs in performance and area costs of the design. Various techniques have been proposed to allow reducing leakage power of heterogeneous designs, while balancing design effort and performance costs. In this chapter, we give an overview of related work in this area and how the work presented in this thesis adds to it.

2.1 Voltage islands and circuit techniques

There has been a great deal of attention focused on the problem of reduction of power consumption of multi-core designs as the number of processors in a single chip keeps increasing. Frequently, each processor core is located on its own *voltage island* by isolating the power supply provided, in order to control its voltage independently of other components. This allows use of *Dynamic Voltage and Frequency Scaling* [66] for reducing dynamic or active power consumption, where the voltage and clock frequency supplied to the island is scaled proportionally to the performance required or the power and heat budget allocated.

The division of hardware designs into separate islands can also be used for power gating of these components to reduce leakage currents. Sleep transistors are added to control the supply of power to the *Power islands*, with the *sleep* signals generated dynamically by a centralized power controller. Kao *et al.* [42] have shown how sharing

of sleep transistors between mutually exclusive signal paths can be used for minimizing the size of the shared sleep transistors to reduce area penalty while still maintaining performance constraints. Calhoun *et al.* [14] have described how insertion of high-threshold sleep devices can reduce standby current dissipation by 8X in configurable logic blocks of a low-power FPGA core. Vangal *et al.* [75] and Matsutani *et al.* [50] have demonstrated use of similar techniques for reducing leakage in on-chip routers, where the power supply for each router component is individually controlled and the scheduling of the activation signals is based on dynamic information in the network packets.

Intel Nehalem [30, 47] and SandyBridge [63] processors have used such Voltage-Frequency islands controlled by a dedicated centralized hardware power management unit. IBM Power7 processors [78] also use voltage islands for power management in a granular fashion of large multi-core designs. Our technique uses similar partitioning but at a finer granularity and consists of distributed power management with control signals generated by pre-existing logic used for scheduling. The partitioning scheme does not require homogeneity in the hardware design, such as that present in identical processor cores or on-chip router components, and can be applied to general heterogeneous hardware designs.

Leakage currents can also be reduced using device and circuit-level techniques such as *Body Biasing* where the body terminals of the transistors are raised to an independent potential instead of being connected to the source terminals. Reverse biasing the source-body junction increases the effective threshold voltage decreasing the leakage current at the cost of reduced performance [40]. Increased circuit complexity due to extra terminals and increased area decrease the viability of frequent use of this technique.

2.2 Operand Isolation

The analysis used for identifying inactive logic is similar to the concept of *Operand Isolation.* Existing literature in this area, [8, 12, 18], has proposed algorithms for

identifying input operands to computational units that are not needed under dynamic data-dependent conditions. These operands are isolated and their value changes are prevented from propagating to the downstream logic.

Such work has targeted reducing unnecessary dynamic transitions in designs towards the goal of cutting dynamic and peak power consumption [69]. We extend the published work on operand isolation by applying it for reducing leakage power, through the automatic identification of unused multiplexer data inputs and power gating of computational blocks generating them. Our technique also adds the use of statistical activity metrics for the different logic blocks in the design to ensure that switching energy costs do not overwhelm the leakage reduction in the selected domains.

2.3 Automation of power gating

Chinnery *et al.* [21] have presented analysis and optimization techniques for gating automation once the list of modules to be power gated and the sleep signals have been specified. Usami *et al.* [74] proposed a fine-grained power gating scheme that relies on already present enable signals of a gated clock design. Bolzani *et al.* [11], similarly, start from a gated-clock netlist for partitioning the design. In contrast, our work does not require any prior analysis of clock gating, and allows gating of logic that might even be feeding registers being clocked in a given cycle, but not being used to compute its next state update.

Rosinger *et al.* [61, 62] have created RTL estimation models to quantify the transition costs and sizing of sleep transistors for small functional blocks switching between inactive and active states. The activity metrics discussed in this thesis build on such work by coupling the costs with statistical data taken from testbenches, to make design decisions on insertion of gating in viable domains. Ickes *et al.* [37] have also discussed the energy costs associated with turning on inactive power domains and the estimation of break-even time as the minimum inactive time required for net energy savings. We have extended this analysis for differentiating between viable and non-

viable fine-grained power domains in a single design, by automatic instrumentation of high-level design for collection and use of the metrics.

2.4 Netlist-based partitioning techniques

Leinweber *et al.* [49] have described a partitioning algorithm starting from design netlists to reduce leakage power consumption. The proposed graph partitioning scheme clusters logic gates in a manner that allows for identification of logic block cofactors that can be power gated. Potluri *et al.* [59] also proposed the extraction of data signals from post-synthesis netlists to be used for partitioning and gating of multiplier blocks present in signal processing applications.

However, limiting such analysis to the synthesized netlist can obscure essential high-level control signals that are present in the HLS design descriptions. Use of the high-level design description can ease the partitioning problem, provide input on the dynamic activity of clusters as well as allow for utilization of the designer's intent for power reduction.

2.5 High-level designs

Dal *et al.* [23] use high-level synthesis from behavioral specification to cluster activity periods of functional units. This enables switching off logic for inactive periods that are as long as possible. However, the area-wise clustering suggested is at a coarse-grained level with minimal consideration of breakeven periods that take into account the cost of switching states.

Singh *et al.* [70] have analyzed the use of *Concurrent Action-Oriented Specifications* like rule-based design languages, for identifying inactive logic to reduce peak dynamic power. This is accomplished by modifying the scheduling of rules to limit the number of rules firing simultaneously and thus limit the amount of dynamic activity occurring in a clock cycle. Our work focuses on analysis of similar rule-based designs towards reduction of leakage power by gating the inactive logic, without changing the

scheduling logic to maintain performance and cycle-accurate behavior of the gated design. In addition, we demonstrate how the use of dynamic activity statistics collected from high-level design simulation is necessary for the selection of appropriate logic domains to be gated, and provide techniques for automatic insertion of logic to collect these statistics. An important point here is the necessity of a direct relationship between the high-level activity and power domains, for accurate selection of viable domains. We will elaborate on this topic in this thesis.

Chapter 3

Power gating and partitioning

3.1 Introduction

Leakage power reduction is increasingly important in hardware design, especially in implementing wireless applications with long standby times [54]. As discussed in Chapter 1, the leakage to active power ratio is increasing with technology scaling [17]. Power gating, *i.e.,* inserting logic to switch off power to parts of the design, is one way to reduce the leakage power. Introducing power gating in a design involves the following steps:

- Dividing the design into power domains and generating power gating signals for each domain to turn the domain off.
- Determining whether it is useful to power gate a particular domain based on the expected dynamic characteristics such as its periods of activity.
- Insertion of isolation logic and power network layout including the sizing of the power switches, and verification of signal integrity.

Generally, the first two steps are done manually while some tools and techniques exist for automation of step 3 [14, 22, 68, 81]. Several industrial standards [44, 60] are also under development for providing power specifications that direct tools in insertion of gating for specified power domains. In this work, we provide a technique

to automate the first two steps; our technique is orthogonal to the solutions used for step 3.

Since power gating is usually done manually, it is applied only at a fairly coarse granularity. Examples of such gating include turning off one or more processor cores in a multi-core chip or turning off inactive wireless radios, Bluetooth, etc. in the multi-protocol transceiver of a mobile phone. This results in relatively large power domains with logic, state and clock networks being switched off for hundreds of clock cycles. This process requires substantial verification effort to ensure that the functionality is preserved. Such power gated designs rely on a global power controller which requires a significant effort to design and integrate. We propose a technique to automatically power gate the majority of the combinational logic on a per cycle basis. This technique can be applied independently of whether a global power management scheme exists, and to designs of any size and complexity without additional design and verification effort. Using the greater savings of a fine-grained approach, our technique can result in a significant reduction of the total leakage power dissipation.

Power gating can reduce leakage, but it has an additional overhead of the recharging energy associated with the gating transistors and power domains. It also incurs the area cost of power-gating transistors, isolation logic and power network. Thus, a crucial part of the analysis is to determine which power domains are likely to produce net energy savings. Such analysis, by its very nature, is based on *use scenarios* and needs to be captured empirically during the design process. We will provide techniques to collect such information and distinguish viable domains.

Our partitioning technique relies on some crucial high-level information that is often lost in low-level RTL descriptions. Therefore, rather than analyzing at the gate level, we rely on designs done in a high-level HDL where the control signals for potential power domains can be easily identified by the compiler. An advantage of our technique is that the power gating control signals are already present for the normal functioning of the original non-power-gated design, and don't require any additional logic for generation. These signals are updated every cycle and so can be used for very fine-grained time switching of the domains. In this chapter, we discuss how to

partition designs using such control signals. In our analysis, we will aim to power-gate combinational logic in the data path, while keeping all state elements ungated. State elements contain data values that might be used within a module or by an external reader. To turn-off state elements, additional information is required to ensure that useful state values are not lost. In Chapter 9, we will provide a brief discussion on how such analysis could be done for state elements.

3.2 Power domain partitioning

3.2.1 Partitioning technique for explicit-control circuits

Partitioning a digital design into fine-grain power domains requires identifying control signals that indicate which part of the next-state logic is needed. Two types of control signals are of greatest interest: register write enables and multiplexer selectors. Although such signals are generally quite obvious in high-level designs, they can get obscured in synthesized netlists. Our partitioning technique includes the following steps:

1. A circuit level description of the design given in terms of four elements: Registers, Multiplexers, Forks (for representing fan-outs), and all other logic gates, shown in Figure 3-1. We call such descriptions explicit control (EC) circuits. Any digital design can be described as an EC circuit.

   Classification of a netlist into EC elements is not unique, for example, a multiplexer is composed of logic gates itself. Rather than identifying such elements in existing netlists, we use a synthesis procedure for rule-based design descriptions where EC circuits arise naturally.

2. A partitioning of the EC circuit into power domains. This step, as described next, is implemented as a graph-coloring task where the set of colors associated with each logic element gives the power gating condition for it.

Our partitioning algorithm is independent of how the EC circuits are generated.

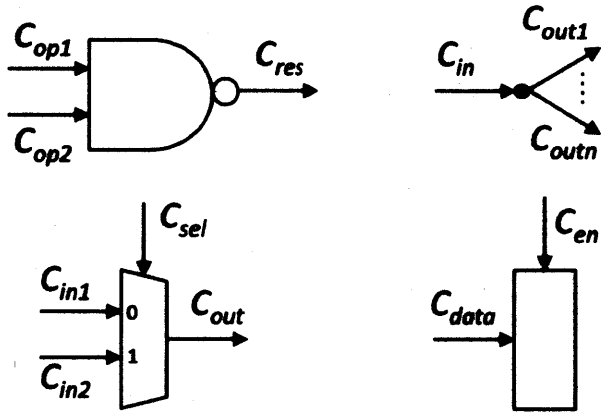


Figure 3-1: Nodes of Explicit-Control circuits

3.2.2 Partitioning algorithm as graph-coloring

We describe our power domain partitioning algorithm as coloring a graph consisting of the EC elements as nodes, and connecting wires as links. The definition of the graph colors is as follows:

1. Seed color: Each register write-enable (en_i) and mux selector signal (sel_j) has a unique seed color given by the dynamic Boolean values of these signals.

2. Link color: Each link in the EC circuit has a color which is a Boolean function of the seed colors.

3. Each Mux and logic gate has a color, which is the same as its output link color.

4. Logic elements having the same color constitute a power domain, with the gating condition defined by the color.

Link colors are derived by solving a set of equations which are set up as follows:

- Register i:

$$C_{data_i} = C_{en_i} \cdot en_i \tag{3.1}$$

- Mux j:

$$\begin{aligned} C_{sel_j} &= C_{out_j} \\ C_{in1_j} &= C_{out_j} \cdot \overline{sel_j} \\ C_{in2_j} &= C_{out_j} \cdot sel_j \end{aligned} \tag{3.2}$$

- Fork k:

$$C_{in_k} = C_{out1_k} + \cdots + C_{outn_k} \tag{3.3}$$

- Logic Gate l:

$$\begin{aligned} C_{op1_l} &= C_{res_l} \\ C_{op2_l} &= C_{res_l} \end{aligned} \tag{3.4}$$

- Constraint due to connectivity: For each link, the sink color is same as the source color.

The reasoning behind the above equations can be understood as follows. In our partitioning technique, registers are always ungated to preserve state. This implies that the write enable signal of the register needs to be computed every cycle to check whether the register value is updated. The color associated with the enable signal, C_{en_i}, evaluates to $True$ because the value of the write enable signal is always needed as a control signal and so the logic computing it is always turned on. The logic generating the data input for a register is only turned on when the value of write enable signal is high, as indicated in equation 3.1. For a mux, the selector signal is needed only when the output of the mux is needed. For the inputs of the mux, colors are determined by value of the selector signal and whether the output is needed, as indicated in equation 3.2. The input signal of a fork is activated if any of the outputs are activated, shown in equation 3.3. In case of a logic gate, if the output is activated, all inputs are required to be activated, shown in equation 3.4.

For a given EC design netlist, it is always possible to construct the above set of equations for every circuit element in the design. Using this set of equations and constraints, it is straightforward to solve for all the link colors in the graph. The solution follows from back-propagation of the seed colors, en_i and sel_j, and gives

Table 3.1: Activity conditions for logic gates

Logic block	Activity Condition
f_1	$\overline{\Phi_1} \cdot \overline{\Phi_3} \cdot \Phi_4$
f_2	$\{\{\Phi_1 \cdot \overline{\Phi_3}\} + \{\overline{\Phi_2} \cdot \Phi_3\}\} \cdot \Phi_4$
f_3	$\Phi_2 \cdot \Phi_3 \cdot \Phi_4$
f_4	$\overline{\Phi_3} \cdot \Phi_4$
f_5	$\Phi_3 \cdot \Phi_4$
f_6	Φ_4

every link a color which is a Boolean expression of seed colors. This is a unique solution as the seed colors are a well-defined set, and their Boolean relationship for every link is determined by the circuit design itself.

3.2.3 Example for generation of gating conditions

As an example of this process, consider the segment of a general design shown in Figure 3-2. Here, each logic block, f_i, consists of a collection of logic gates and forks. The seed colors of the control signals, the selectors for muxes and write enables for registers, are denoted by Φ_i, whose generation logic is kept ungated. In this scenario, each logic block has a distinct activity condition consisting of all the control signal values that allow it to propagate to any state element, as shown in Table 3.1. In this manner, power domains and their gating conditions can be generated for an EC circuit. In section 3.3, we will discuss partitioning in the absence of known control signals and how that affects activity of generated domains which determines energy costs associated with gating.

3.2.4 Nested domains

It is possible that the domains generated through such partitioning can be physically contained within larger domains. For example, consider Φ_i and $\Phi_i \cdot \Phi_j$ as the gating conditions of two domains, where the second domain is contained within the first domain. Such specification of domains is valid under the technique and would

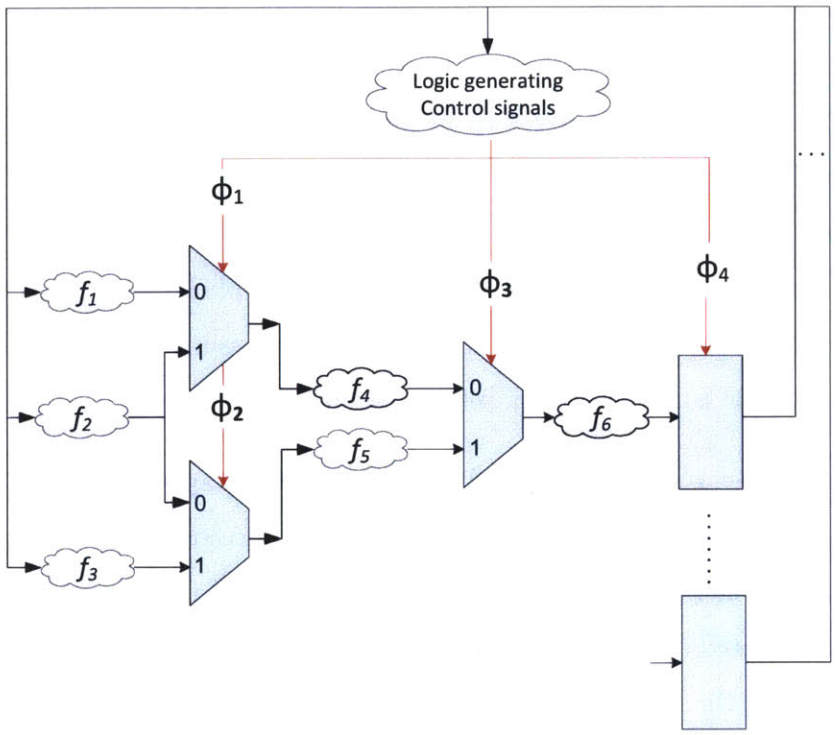


Figure 3-2: Associating activity conditions with logic blocks. Control signals are marked in red.

imply a hierarchical granularity of power gating. The larger domain could be turned on, while the nested domain is turned off depending on the value of Φ_j control signal. This can occur due to the presence of multiple conditional computations in a design block where some computation is common to several selections. Though such nested power domains are allowed under our partitioning algorithm, the rule-based implementation that we will describe in the rest of this thesis will generate disjoint fine-grained domains due to the method used for identifying control signals.

3.3 General partitioning strategies

In this section, we explore various techniques that can be used to partition designs starting from a low level HDL description, such as in Verilog RTL or design netlists. In this analysis we assume no high-level information to be present.

1. *Pre-defined control signals: Information about structure, different types*

For the earlier analysis, we had defined control signals as specifically mux selector signals and register write enables. The mux selector signals choose between two (or more) alternatives to be used as state updates. This is a structural definition, since it relies on knowledge of the presence and location of mux logic units. When starting from high-level rule-based designs, we are interested in preserving information about explicit (and implicit) guards to use as control signals. For general partitioning, we have to assume that it might not always be possible to identify mux selector signals from a given design netlist.

Example — Consider a typical RTL design of a swap-and-subtract GCD module discussed earlier. The hardware design, split into control and datapath modules, is shown in Figure 3-3. The main computational unit is the *subtractor* and it can be gated as a fine-grained power domain. The control signal of this domain would be the AND of conditions that the operand B is non-zero and that the operand A is smaller than operand B. The mux selector signal, A *sel*, can also provide information on whether the subtractor result will be used in a given cycle or not. However, in a general Verilog RTL design, such control signals could be obscured in the design description, and this makes automatically selecting gating controls difficult. Figures 3-4 and 3-5 show the modular RTL code for a design where control signals for gating are directly identifiable due to the explicit structural code used for the design description. On the other hand, Figure 3-6 shows a flattened RTL design where such control signals would be difficult to be automatically identified.

2. *Partitioning using only register write enables*

   For any given design, as Verilog RTL or netlist, we can determine the register write enable signals. However, control signals corresponding to choice of state update (earlier described mux selector signals) need not be explicitly specified. In such cases, we use the sensitivity conditions for the *always* blocks to determine the control signal for the register WE. The coloring algorithm described in section 3.2.2 is still applicable here with registers, logic gates and forks being the

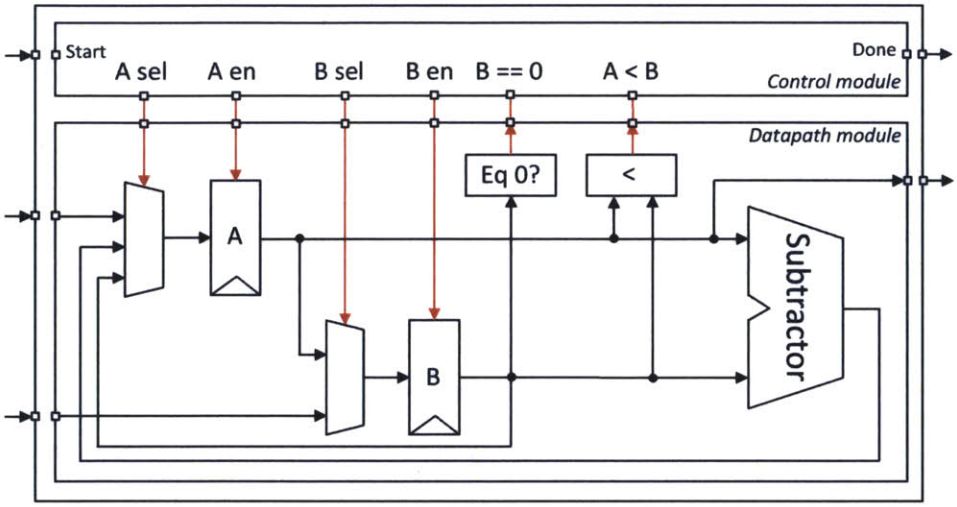


Figure 3-3: Hardware generated for GCD RTL design, control signals are marked in red. Details of control module have been omitted because it is kept ungated. Fine-grained power gating is applied to the computational unit for subtraction in the datapath module.

```
module gcd
(  input clk,
   input [7:0] operand_A,
   input [7:0] operand_B,
   output [7:0] result,
   input start,
   output done );

   wire [1:0] A_sel;
   wire A_en, B_sel, B_en, B_eq_0, A_lt_B;

   gcdDatapath datapath (
   .operand_A(operand_A),
   ...
   .A_mux_sel(A_sel),
   ...
   )
   gcdControl control (
   .A_sel(A_sel),
   ...
   )
endmodule
```

Figure 3-4: Top-level Verilog RTL design of GCD with separated control and datapath modules

```
module gcdDatapath
(   input clk,
    input [7:0] operand_A,
    input [7:0] operand_B,
    output [7:0] result,
    input A_en,
    input B_en,
    input [1:0] A_mux_sel,
    input B_mux_sel,
    output B_eq_0,
    output A_lt_B );

    wire [7:0] B, sub_res, A_mux_out, B_mux_out;
    Mux3Inp#(8) A_mux
    (.in0(operand_A), .in1(B), .in2(sub_out), .sel(A_mux_sel), .out(A_mux_out));

    Mux2Inp#(8) B_mux
    (.in0 (operand_B), .in1 (A), .sel (B_mux_sel), .out (B_mux_out));

    ED_FF#(8) A_ff   // D flip-flop with en
    ( .clk  (clk), .en_p (A_en), .d_p  (A_mux_out), .q_np (A) );
    ED_FF#(8) B_ff
    ( .clk  (clk), .en_p (B_en), .d_p  (B_mux_out), .q_np (B) );

    Subtractor#(8) sub
    ( .in0 (A), .in1 (B), .out (sub_out) );
    EqChk#(8) B_eq_zero
    ( .in0 (B), .in1 (0), .out (B_eq_0) );
    Comparator#(8) comp
    ( .in0 (A), .in1 (B), .out (A_lt_B) );

    result = A;
endmodule


module gcdControl
(   input clk, input start, input B_eq_0 ... output A_sel .. )

// Next state logic
...
// Generating output signals
...
endmodule
```

Figure 3-5: Verilog RTL design of GCD datapath where signals for gating are directly identifiable. Details of control module code has not been shown here. Each combinational unit is contained in a leaf module instantiated in the datapath module. The control signals used for mux selection can be identified due to explicit mux structures being declared in the design. The output of subtractor is only used as an input for the 3-input mux and so the mux selector can be used for gating the subtractor.

```
module gcd
(  input clk,
   input [7:0] operand_A,
   input [7:0] operand_B,
   output [7:0] result,
   input start,
   output done );

   wire [1:0] A_sel;
   wire A_en, B_sel, B_en, B_eq_0, A_lt_B;

   reg [7:0] A, B;
   reg [1:0] state;
   wire [7:0] A_in = operand_A;
   wire [7:0] B_in = operand_B;
   always@(*)
   begin
     if (state == 0 && start)
     begin
       A <= A_in;
       B <= B_in;
       state <= 1;
     end
     else if (state == 1 && (A<B))
     begin
       A <= B;
       B <= A;
     end
     else if (state == 1 && (B != 0))
     begin
       A <= A - B;
     end
     else
     begin
       done <= 1;
       state <= 0;
     end
   end
   result = A;
endmodule
```

Figure 3-6: Flattened Verilog RTL design for GCD, where it is difficult to identify control signals for gating combinational logic.

circuit elements. Such analysis would result in coarser-grained domains. Most of the logic between two registers would be gated by the WE of the destination (later) register. Only logic generating the WEs needs to stay powered on.

Such power domains would have a lower level of inactivity - compared to the

case where each alternative of a state update has a separate domain. In this case, any of the alternatives would make the entire domain be active. In certain cases, where only one possible update (such as an increment) happens to a state element, it will be equivalent to the fine-grained gating.

Large module-level domains have much lower inactivity rates as discussed in chapters 4 and 5. This leads to a reduction in potential savings. At the other end of the spectrum, unit logic cell-level domains have an excessive area cost and very frequent inactive-active transitions which again reduce leakage savings from gating.

3. *Acyclic nature of hardware, register outputs don't affect color of inputs*
   In our analysis, the hardware design is represented as Finite State Machines, where we only attack the static power consumed by combinational logic. Since the register state elements are kept ungated, there is no color associated with the register outputs in our algorithm for the coloring problem. This allows the hardware design analysis to be linearized, getting rid of the existing circular dependency of the next state computation on the values of state elements. For power gating of state elements, analysis is required to determine when state has been read and will not be needed in future. We discuss details of such analysis in Chapter 9.

4. *Possibility of using an arbitrary data signal as control: With no information to distinguish mux selectors from other signals, can we do gating at all?*
   Such analysis would typically result in logic cell-level gating. We can consider the basic logic building blocks of AND (or NAND) gates and use one of the two operand signals as a control signal for use of the other. Only if the chosen control operand is *high* (for the case of AND gates), then the logic cell generating the other operand is turned on. Potentially, this could give us domains bigger than single logic gates. The entire logic tree feeding the other input of the AND gate could form a power domain to be gated. Since the choice of the control signal is arbitrary, it is possible that the activity rates are higher than to be expected

from the mux based scheme. With the number of domains increasing beyond the earlier described gating, the area and performance costs would also increase.

It is clear that high-level information is required for identification of the appropriate control signals by automated tools. Without such information, the partitioning hardware designs for power gating would not achieve desired results. In Chapter 4, we describe how rule-based designs are leveraged to provide such information as they generate explicit-control circuits by default.

It is important to realize that different power domains dissipate different amounts of leakage power. In fact, when we take into account the energy overhead of switching on the power domain from an inactive state, then it may not make sense to power gate logic that is very active or has short inactive intervals. In Chapter 5, we will describe how to differentiate between power domains based on their dynamic activity characteristics.

Chapter 4

Use of rule-based designs for power gating

In this chapter, we discuss a novel technique that uses rule-based design descriptions for automatic partitioning of a digital design into power domains and associated gating signals and automatic classification of power domains for their suitability for power gating based on their dynamic characteristics. We illustrate how such designs naturally provide information about the control signals discussed in the Chapter 3 that can be used for power domain partitioning. In the following chapters, we will discuss concrete examples showing how this technique is applicable to real-world designs. We also show that unlike global power controllers, our technique introduces no new logic or state to generate power gating control signals and thus, is correct by construction reducing design and verification costs.

4.1 Hardware compilation from rule-based designs

Our technique uses design descriptions that consist of state definitions and *rules*, each of which computes some state updates. We use Bluespec System Verilog (BSV) [10] as the source language for rule-based designs. This design methodology is very well established in the field of high-level synthesis and has been extensively used to design complex digital hardware including out-of-order processors [24], wireless base-band

systems [55], video compression [28], multi-core cycle-accurate simulators [45], etc.

In BSV, each rule is a set of state transformations, and is defined as a *guarded atomic action* that can execute atomically in a given cycle if the associated guard condition is true. A precise and useful language semantics ensures that any legal behavior of the system can be understood as a series of atomic actions on the state [5, 36]. All control circuitry used to manage interactions between rules is produced by automatic synthesis by the Bluespec compiler. The compiler generates a scheduler which selects rules to execute every clock cycle. The scheduler logic ensures that there are no double writes to any register *i.e.,* at maximum only one rule can update a register in a clock cycle. On compilation to Verilog RTL, the compiler generates multiplexers to select the state updates based on these scheduler-generated control signals. In this manner, any EC circuit can be generated from a rule-based design. Next, we elaborate on the generation of these control signals.

4.1.1 Scheduling logic in rule-based designs

Detecting and scheduling legal concurrent execution of rules is the central issue in hardware synthesis from Bluespec's operation-centric descriptions [36]. The description framework uses the Term Rewriting Systems (TRS) notation. Here, the state of a system is given by the collective values of state elements in the design. Each rule rewrites or updates these values. Bluespec language semantics apply to specification of the hardware behavior with a sequential application of rules. Scheduling strategies are targeted towards maximizing performance while still maintaining one-rule-at-a-time semantics. The Esposito scheduler algorithm [27] is the standard scheduler generation algorithm in the Bluespec compiler. It is a heuristic to generate efficient scheduling hardware while still achieving reasonable rule-level parallelism.

The challenge in efficient hardware generation is to obtain a scheduler that can pick a maximal set of rules to be executed simultaneously every cycle. *Conflicting* rules make updates to the same state elements, and should not be scheduled to fire in the same cycle even if their guard conditions are true. If two rules do not update the same state, and neither updates state that the other rule accessed, then the rules are

said to be *conflict free* and can execute simultaneously. Two rules (r1 and r2) are said to be *sequentially composable* if one rule (r2) does not access any state updated by the other rule (r1), and execution of r1 does not disable r2. Simultaneous execution of such sequentially composable rules is legal, because it produces the same result as sequential execution of r1 followed by r2. Rules whose guard conditions cannot be simultaneously true are said to be *mutually exclusive*. The compiler aggressively searches for such relationships between the rules to minimize resource consumption while maximizing the performance achieved by the design.

Figure 4-1 [5] shows how the system state is used to compute dynamic values of rule guards, which are predicates (π_i), and updated state (δ_i). The scheduler then uses the predicate values and generates the rule firing signals (ϕ_i). Selection logic is then used to generate the final updates for each state element.

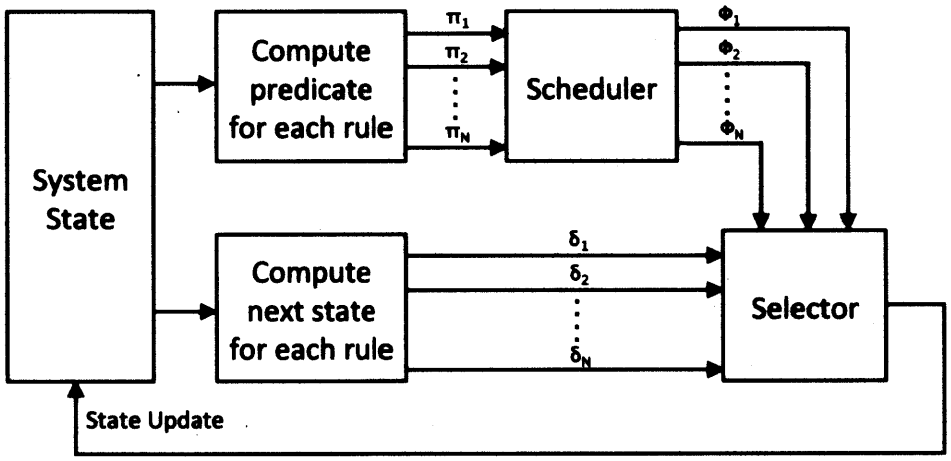


Figure 4-1: Generation of rule firing signals ϕ_i by the scheduler

4.1.2 Example of hardware generation from BSV

As an example, consider the conceptual BSV module shown in Figure 4-2, consisting of state elements x and y, and two guarded rules $r1$ and $r2$ for state updates. Functions $f1$, $f2$, $f3$, $f4$ and $f5$ used in these rules can be arbitrarily complex combinational functions, which after synthesis can result in large logic blocks. This module

```
module mkExample (InterfaceExample);

   Reg x, y; //state definitions

   rule r1 (p1(x,y));
      x <= f1(x,y);
      y <= f3(y) + f4(x,y);
   endrule

   rule r2 (p2(x,y));
      x <= f2(x,y) + f3(y);
      y <= f5(x,y);
   endrule

   method definitions;

endmodule
```

Figure 4-2: Conceptual design of a module in BSV

description generates the circuit shown in Figure 4-3. The scheduler logic that generates rule firing signals ($willFire_{ri}$) has not been shown as it is not gated. Typically the scheduler logic is much smaller than the rest of the combinational logic of the module. In this example, since both rules update registers x and y, the scheduler ensures that they can never fire in the same cycle.

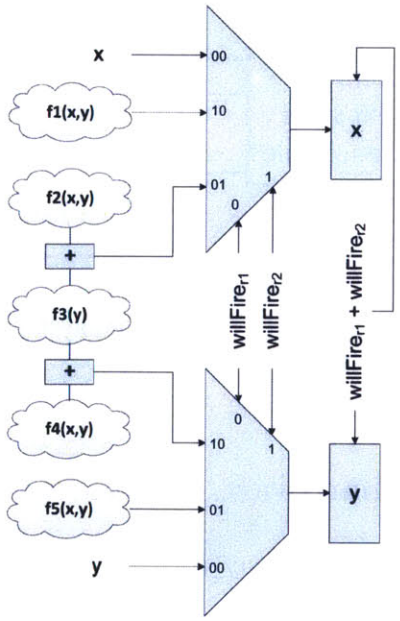


Figure 4-3: Hardware generated for example BSV module with state updates selected by the rule firing control signals

4.2 Selection of power domain logic

We parse the BSV-generated Verilog to determine the logic blocks that provide data inputs to muxes, and introduce a fine-grained power domain for each input controlled by the corresponding mux control input. Shared logic used to update multiple state elements is gated by the OR of the control signals that select this logic in each mux. The use of BSV rules to automatically generate the power domains ensures correctness by construction. Figure 4-4 shows the possible set of power domains for the example from Figure 4-2.

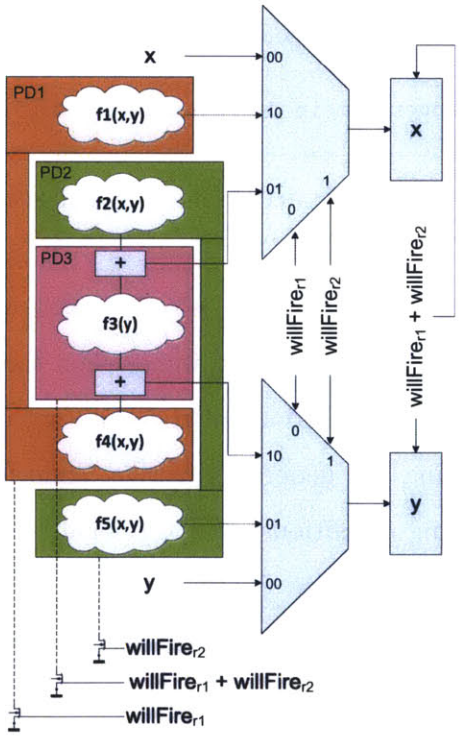


Figure 4-4: Power domain partitions and associated gating conditions for example design

It consists of three elements: $f1$ and $f4$ in one domain which is turned on only when $r1$ is selected, $f2$ and $f5$ in another domain which is turned on when $r2$ is selected, and $f3$ in a third domain which is turned on when either rule is selected for execution. The two muxes can also be considered as part of the third domain.

We summarize our algorithm for generating and selecting fine-grained power do-

```
Input: Rule-based design and compiled RTL

1. Identify Muxes that select state updates
2. Mux Control inputs are functions of rule firing
   signals Φ
3. Mux Data inputs are potential logic blocks
   with some shared logic
4. Use graph-coloring to select logic functions
   that are controlled by a unique control signal
5. Generate fine-grain UPF power domain description
6. Collect activity metrics to determine leakage
   power savings and switching cost for domains
7. Select power domains that have net power savings

Output: Power Domain definitions for logic blocks
```

Figure 4-5: Application of partitioning algorithm to rule-based designs

mains from rule-based designs in Figure 4-5. We start with an N-rule design, with each rule having a firing condition determining when the rule is active. These conditions, denoted as $\Phi_{1..N}$, are functions of the state within the design. Given these control signals and parsing the generated Verilog code for the mux control inputs, we can associate the firing conditions with the logic blocks used for state updates. This allows us to generate a Unified Power Format (UPF) [44, 71] specification that provides the place and route tools with the power domain description for each of the logic blocks and their respective gating signals. Finally, we collect activity metrics for the identified logic blocks using rule firing statistics to select appropriate power domains.

Through the use of rule-based digital design description, we can generate hardware that is amenable to the discussed graph-coloring analysis for power domain partitioning as well as collection of dynamic activity metrics for logic blocks. Such designs generate groupings of logic elements and control points, which lead to a natural description in terms of fine-grained power domains. A fundamental issue in power gating is that it should not alter the functionality of the overall design by introducing

unintended behavior, such as turning off some logic necessary for correct computation or delaying computation to a later clock cycle. Grouping logical blocks into distinct power domains, and verifying that they do not introduce changes requires significant verification effort. This is avoided when we power gate rule-based designs. It is important to note that our technique is not limited to BSV – any design process that generates well-structured RTL code with pre-defined control signals for logic blocks can be used for the method of partitioning that we have outlined.

4.3 Generating UPF/CPF description

Unified Power Format or Common Power Format are power specification industry standards developed to aid designers in specifying and verifying power intent for hardware designs. In our technique, the specification for each design can be generated automatically using the information collected from the high-level design. Instead of a centralized power controller, we specify the individual control signals for each power domain as boolean expressions of the *willFire* signals for each rule. From the earlier parsed designs, the logic elements associated with each Mux input are specified as a power domain along with appropriate gating signal.

```
...
create_power_domain -name PDTop -default

create_power_domain -name PD1 -instances {inst_A}
-shutoff_condition {!willFire_r1} -base_domains PDTop

create_power_domain -name PD2 -instances {inst_B}
-shutoff_condition {!willFire_r2} -base_domains PDTop
...
```

Figure 4-6: CPF Power specification code segment for the example design

4.4 Collecting statistical information

After specification of the domains, we have to determine the best candidates for power gating based on a power domain's statistical dynamic activity. We discuss this topic in detail in Chapter 5. This can be done easily for rule-based designs by associating activation of rules with corresponding logic blocks.

The statistical data is collected by automatically instrumenting the BSV designs for rule activity collection. For each rule in the design, a wire is added that is asserted *high* if the rule fires in that cycle. A book-keeping rule polls these wires every cycle, and maintains state for counting number of inactive cycles. Though we have used wires for communicating same-cycle activity, the same information can also be collected using additional state that is updated by the book-keeping rule. In addition, one-bit history is maintained for each rule's activity in the previous cycle. This allows counting the number of inactive-active transitions when the history shows that the rule has become active only in the current cycle. The BSV designs are then simulated using high-level testbenches that emulate expected input data rates and values. Such BSV-level simulations can run much faster that gate-level analysis, but still provide data directly corresponding to the power domains under consideration. The instrumentation does not affect the generated hardware as it is only generated for simulation using compile time macros. Figure 4-7 shows how the example BSV module is instrumented to collect the statistical activity counts for the two-rule design.

In this chapter, we showed how the partitioning algorithm is applied to hardware generated from high-level design description. Use of rule-based design allows easy identification of control signals for automation of partitioning. Such partitioning is correct by construction as it relies on the pre-existing scheduling logic generated by the Bluespec compiler. This greatly eases the design and verification effort associated with fine-grained power gating. Activity collection at the rule-level can be performed by automatic instrumentation of such designs. In the next three chapters, we explore the use and impact of rule-based power gating in detail. In Chapter 5, we use realistic

```
module mkExample (InterfaceExample);

   Reg x, y; //state definitions

   // State inserted for activity collection during simulation
   // Does not affect synthesized hardware
   `ifdef STATS_ON
     Reg r1_cnt, r2_cnt;                    // Counters for total inactivity
     Reg r1_int_cnt, r2_int_cnt;            // Counters for intervals
     Reg r1_prev, r2_prev;                  // Activity in previous cycle
     PulseWire r1_active <- mkPulseWire();  // Active r1 in current cycle
     PulseWire r2_active <- mkPulseWire();  // Active r2 in current cycle
   `endif

   rule r1 (p1(x,y));
      x <= f1(x,y);
      y <= f3(y) + f4(x,y);
      `ifdef STATS_ON
        r1_active.send();              // Indicate active r1 in current cycle
      `endif
   endrule

   rule r2 (p2(x,y));
      x <= f2(x,y) + f3(y);
      y <= f5(x,y);
      `ifdef STATS_ON
        r2_active.send();              // Indicate active r2 in current cycle
      `endif
   endrule

   `ifdef STATS_ON
     rule stats;
        if (r1_active)                   // If active in current cycle,
           r1_prev <= True;              // store active state for next cycle
           if (r1_prev == False)            // If inactive in prev cycle,
              r1_int_cnt <= r1_int_cnt + 1; // increment interval count

        else                             // If inactive in current cycle,
           r1_cnt <= r1_cnt + 1;         // increment inactivity count
           r1_prev <= False;

        if (r2_active)
           ....
     endrule
   `endif

endmodule
```

Figure 4-7: Instrumentation of the example BSV module to collect activity statistics for each rule in the design.

hardware design examples to show how the obtained statistical activity information is used to distinguish viable domains likely to generate net energy savings. In Chapter 6, more complex example designs are used to illustrate the difference in activity between coarse-grained gating at the module-level and fine-grained gating at the rule-level. Chapter 7 provides our methodology to compute breakeven thresholds for inactivity intervals and gives data on the power and performance impact of the rule-based gating.

Chapter 5

Dynamic Activity Metrics

In order to determine which power domains will provide actual leakage power savings when gated, we need to characterize the dynamic activity of the overall design. Since the energy savings arise from turning off power to inactive domains, if we power gate a domain which is mostly active during run-time it would not provide a lot of savings. Thus, the average activity of identified domains need to be computed. In addition, the energy lost during change of inactive to active state of a domain should also be accounted for to estimate net energy savings.

For this purpose, we define two dynamic data-dependent activity metrics for the power domains under consideration. There are two ways to use the metrics: first, to see whether the individual power domain should be gated and second, to see whether gating all possible domains of a design saves power. Given a representative testbench or a suite of testbenches for the overall design, we compute the following dynamic activity metrics:

Metric 1: Total inactivity (N_1) It is defined as the total number of clock cycles in which the logic block under consideration is inactive, *i.e.,* information generated by the block is not used to compute and update any state element.

Metric 2: Number of inactive-active transitions (N_2) It is defined as the total number of times the logic block becomes active from an inactive state in the previous cycle.

Use of these metrics depends on the values of two technology-dependent charac-

teristics of the power domain under consideration. Given the desired clock cycle time t, let the leakage power saved by gating the logic block be α and the energy cost of switching on the power domain from sleep state be β. Once we have computed these metrics over a realistic testbench, the net energy saved for the i^{th} power domain is given by equation 5.1. The parameter α_i is multiplied by t to get the leakage energy saved per cycle and their product with N_1 gives the total energy saved over the entire testbench. From this we subtract the energy lost during transitions, obtained by the product of β and N_2. For the complete design, the total energy saved can be computed by summing over all the gated domains as in equation 5.2.

$$E_i = tN_{1i}\alpha_i - N_{2i}\beta_i \tag{5.1}$$

$$\sum_i E_i = t\sum_i N_{1i}\alpha_i - \sum_i N_{2i}\beta_i \tag{5.2}$$

The ideal metrics for power gating a domain would be a high enough number of total inactive cycles (N_1) to compensate for the switching costs of all inactive-active transitions (N_2). Consider a logic block that is inactive for half of the total clock cycles of a test input, but each inactive interval is just one clock cycle, followed by one cycle of activity and so on. In this case, even though its N_1 metric would be quite high, the number of transitions, N_2, would also be very high and would swamp any leakage savings achieved by power gating. In this manner, dynamic metrics allow us to eliminate power domains that are unsuitable for gating due to their expected activity profiles.

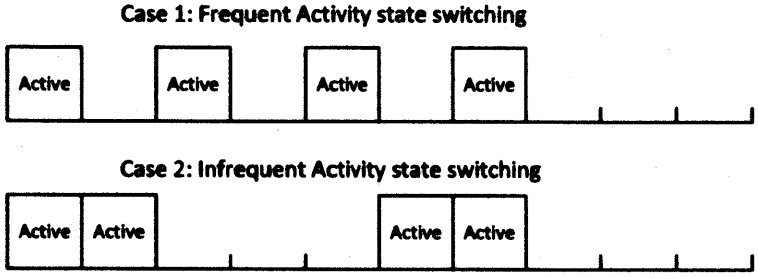


Figure 5-1: Dynamic Activity profiles

As an example, consider two cycle-by-cycle activity profile scenarios shown in

Figure 5-1 for a power domain under consideration. It shows the dynamic activity profiles over a typical period of 10 cycles that can be assumed to repeat indefinitely. In both cases, the power domain under consideration has the same average activity ratio of 40%, with a total inactivity length (N_1) of six cycles. In the first case there are four inactive-active transitions (N_2), while the second case has two such transitions. It is clear that the second activity profile would result in higher leakage energy savings on power gating. We can also quantify the average inactivity interval length (N_1/N_2) for the first case as 1.5 cycles and for the second case as 3 cycles. Greater the average inactivity interval length, greater is the net energy savings after power gating.

For the example in Figure 3-2, computing these metrics would require information about the dynamic values of the control signals (Φ_i) for realistic testbenches. In general, though this can be done for any EC circuit, doing it at the netlist level is cumbersome and resource intensive as it requires a complete design simulation for very large test inputs and collection of statistical values for each control signal.

In this work, we will demonstrate how we collect such metrics for rule-based designs. For most domains, we can have a direct relationship between the logic blocks being gated and specific rules which govern their activity. In certain cases, shared logic arising from common sub-expressions in the design description would be activated when any single one of a set of rules is active in a cycle. These relationships can also be inferred during static analysis of the rule-based design description. For simplicity, we will discuss the metrics for the various rules in each module of realistic hardware designs. The metrics are used to make one-time decisions whether to insert gating for each domain under consideration. Dynamic configuration can determine which mode a particular design is operating in, and such logic can inform dynamic decisions about when to switch-off power domains. However, most of the area and performance costs associated with power gating occur when the static decision to insert gating is made. So we focus on that point of the design process. Energy savings from inserting logic to control dynamic conditions are small, and orthogonal to our technique as such logic can be added independently.

For illustrative purposes, we have chosen two standards-compliant wireless de-

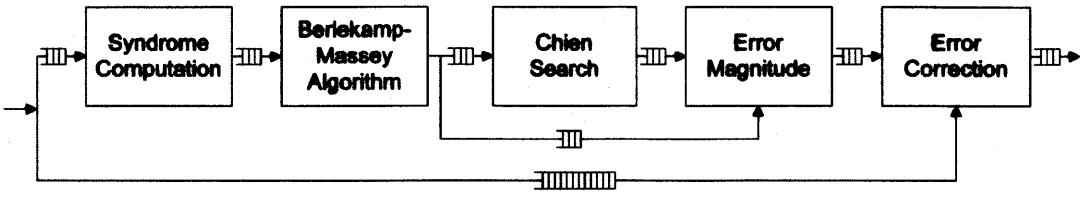


Figure 5-2: Architecture of Reed-Solomon Decoder

coders to use for our analysis. We have simulated the designs under realistic traffic patterns to analyze the dynamic activity statistics of their design components.

The first design is a Reed-Solomon decoder [4] which is used in 802.16 transceivers. This design is parameterized by the input block size, and for this study was configured to have 32 bytes of parity in each 255 byte input block. The second design is a Viterbi decoder [56], which is a component in an 802.11 transceiver. Both these designs are high performance implementations that meet performance requirements for respective wireless protocols.

5.1 Metrics for the Reed-Solomon decoder

Figure 5-2 shows the architecture of the Reed-Solomon decoder consisting of five modules, each of which performs one step of the decoding algorithm as a finite state machine (FSM). These FSMs are designed such that they might terminate early depending on the dynamic input conditions. We computed the earlier defined activity metrics for each rule in the five modules of the Reed-Solomon decoder under two different input conditions. The data shown in Table 5.1 corresponds to the maximal activity case which occurs when the number of errors in the input data is equal to the maximum correctable limit determined by the number of parity bytes in the underlying Reed-Solomon encoding. The data was collected over a testbench of 4000 cycles of simulation.

At the other end of the spectrum, Table 5.2 gives the data for the case where the input data is entirely uncorrupted with no errors. For this input, the first and third modules of the Reed-Solomon pipeline, Syndrome and Chien, still have approximately the same number of inactive cycles (N_1) as the earlier case, while the Error

Table 5.1: Reed-Solomon Activity Metrics: Input data with maximal correctable errors

Rule	N_1: **No. of Inactive cycles**	N_1%: **Percentage inactivity**	N_2: **No. of Inactive-Active transitions**	N_1/N_2: **Avg. Inactivity interval**
Syndrome Module				
r_in	24	0.6	17	1.4
s_out	3984	99.6	16	249.0
Berlekamp Module				
calc_d	3474	86.8	526	6.6
calc_lambda	3489	87.2	511	6.8
calc_lamb_2	3494	87.3	506	6.9
calc_lamb_3	3494	87.3	506	6.9
start_new	3984	99.6	16	249.0
Chien Module				
calc_loc	222	5.5	38	5.8
start_next	3985	99.6	15	265.7
Error Magnitude Module				
eval_lambda	261	6.5	233	1.1
enq_error	3767	94.2	233	16.2
deq_invalid	3986	99.6	14	284.7
process_err	714	17.8	71	10.1
bypass_int	4000	100	1	4000.0
start_next	3985	99.6	15	265.7
Error Corrector Module				
d_no_error	4000	100	1	4000.0
d_correct	711	17.7	71	10.0

Magnitude computation module is completely inactive as there are no magnitudes to be computed. Berlekamp module also has increased N_1 values, as the rules involved in the computation of the error magnitude polynomial are inactive.

In order to use this statistical data to select the viable domains for power gating, we need to come up with appropriate values of α and β for each power domain, depending on the amount of logic generated for each rule and the technology library used. Here, we will present a general discussion.

For the Syndrome module, the entire computation logic is limited to rule *r_in* which is nearly always active, and hence non-viable for gating. For the Berlekamp

Table 5.2: Reed-Solomon Activity Metrics: Input data with no errors

Rule	N_1: **No. of Inactive cycles**	N_1%: **Percentage inactivity**	N_2: **No. of Inactive-Active transitions**	N_1/N_2: **Avg. Inactivity interval**
Syndrome Module				
r_in	16	0.4	16	1.0
s_out	3984	99.6	16	249.0
Berlekamp Module				
calc_d	3472	86.8	528	6.6
calc_lambda	3488	87.2	512	6.8
calc_lamb_2	4000	100	1	4000.0
calc_lamb_3	4000	100	1	4000.0
start_new	3984	99.6	16	249.0
Chien Module				
calc_loc	32	0.8	16	2.0
start_next	3984	99.6	16	249.0
Error Magnitude Module				
eval_lambda	4000	100	1	4000.0
enq_error	4000	100	1	4000.0
deq_invalid	4000	100	1	4000.0
process_err	4000	100	1	4000.0
bypass_int	3984	99.6	16	249.0
start_next	3984	99.6	16	249.0
Error Corrector Module				
d_no_error	528	13.2	16	33.0
d_correct	4000	100	1	4000.0

module, the amount of logic in each *calc_** rule is about the same with no shared logic. Though N_1 is quite high for all the rules, N_2 (number of inactive-to-active transitions) is relatively high for rules *calc_d* and *calc_lambda* as seen in Table 5.1. Hence, the better candidates for gating are rules *calc_lambda_2* and *calc_lambda_3*. Analysis of the Chien module is similar to that of the Syndrome module. The Error Magnitude module's activity profile depends on whether the input data has errors or not, and hence expectations of the noise characteristics in the use-environment of the module would affect the selection of power domains in this case. Rules *enq_error*, *deq_invalid*, *bypass_int*, and *start_next* can be expected to be good candidates in most cases. The Error Corrector module has a bimodal activity profile where one of two

Table 5.3: Reed-Solomon Module-level Inactivity percentage

(a) Input data with maximal correctable errors

Module	N_1%: **Percentage inactivity**
Syndrome	0.2
Berlekamp	48.4
Chien	5.2
Error Magnitude	0
Error Corrector	17.8

(b) Input data with no errors

Module	N_1%: **Percentage inactivity**
Syndrome	0
Berlekamp	73.6
Chien	0.4
Error Magnitude	99.2
Error Corrector	13.2

rules fire with a very high activity rate, depending on the presence or absence of errors, making them poor candidates for gating.

It is important to note that by performing this rule-based partitioning, we are able to identify sub-modular blocks with high percentage inactivity. To illustrate this point, we also generated *coarse-grained* module-level metrics for the Reed-Solomon decoder design. For these metrics, if any rule in a module was fired in a given cycle, all the logic of the module was activated. The combinational datapath logic of a module could be switched off only if all the rules are inactive in a given cycle. These module-level metrics are shown in Table 5.3. When comparing the percentage inactivity of the modules to that of the rules contained in these modules, seen in Tables 5.1 and 5.2, it is clear that the module-level inactivity is significantly lower. In fact some modules have at least one rule active in every cycle giving a figure of 0% module-level inactivity, such as Error Magnitude in the maximal error case and Syndrome in the no-error case. Thus, partitioning at the rule-level allows significantly more opportunities for power savings than at the module-level.

5.2 Importance of macro factors on activity

The environment of the logic block, *i.e.*, how it fits into the macro-level design, is crucial in determining its actual activity level beyond the information obtained using detailed power simulations of the circuits. The activity of a wireless component inside a radio pipeline will be affected by high-level external data, such as how frequently the radio pipeline itself is active in a multi-radio design. This can be illustrated using the example of the logic blocks in the Reed-Solomon Decoder.

The Reed-Solomon algorithm uses the Galois Field arithmetic for computation. Accordingly, throughout the design we have a number of combinational GF multipliers, consisting of bit shift and XOR operations. Within the Berlekamp module, there are 32 of such 8-bit multipliers. Once syndrome polynomial input is available, the Berlekamp computation is done using these multipliers, all of which are not used simultaneously. So it is possible to switch off one set of multipliers when they are inactive.

The Berlekamp algorithm implementation does polynomial multiplication using a circular pipeline and generates a polynomial whose degree depends on the number of errors in the input block. The Chien Search module uses this polynomial to determine which locations in the input block have errors. Downstream modules like the Error Magnitude computation module, thus, only become active if there are errors in the input block and this is only determined dynamically. Within the Reed-Solomon decoder, the pipeline has been balanced between various modules for the worst case/maximum number of errors. If there are fewer errors, then some modules become intermittently idle.

The Reed-Solomon decoder is used in wireless receivers as a Forward Error Correction (FEC) decoder, shown in Figure 5-3. Depending on the structure of these receiver pipelines, the decoder is not always active. The 802.16 receiver itself is not always receiving and decoding packets and may spend a considerable amount of its time in standby/idle state.

All of the above factors have a strong impact on reducing the overall activity

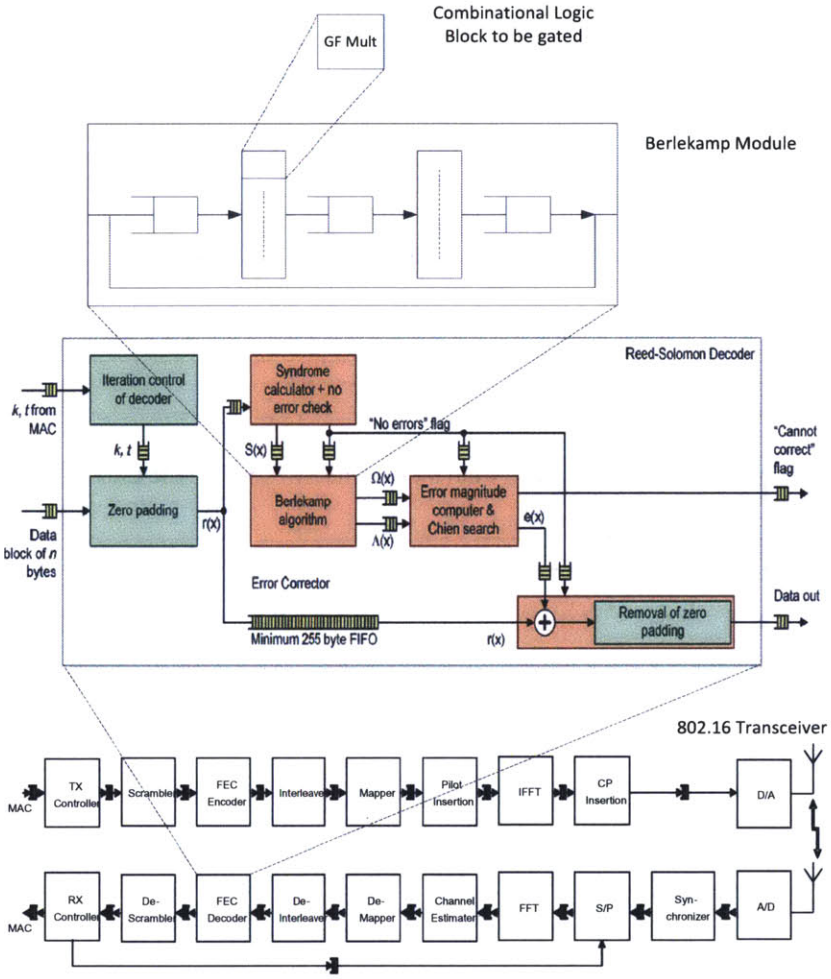


Figure 5-3: Hierarchical activity considerations for logic blocks within wireless decoders

factor of the logic blocks. It is clear that for average activity factor of the module under consideration, impact of external architecture is strictly to reduce it or keep it at same level. Metric 2, on the other hand, is more interesting because a standalone testbench continuously pushes data in, making inactivity intervals artificially short, while a real usage would have longer intervals. So we need high-level testbenches that can generate statistical data for the rate and profile of external inputs to the decoder design under consideration.

The earlier analysis for Reed-Solomon was done using a standalone testbench that continuously pumps input data into it. Depending on the structure of these receiver pipelines, the decoder is not always active. In the next subsection, we use the Viterbi

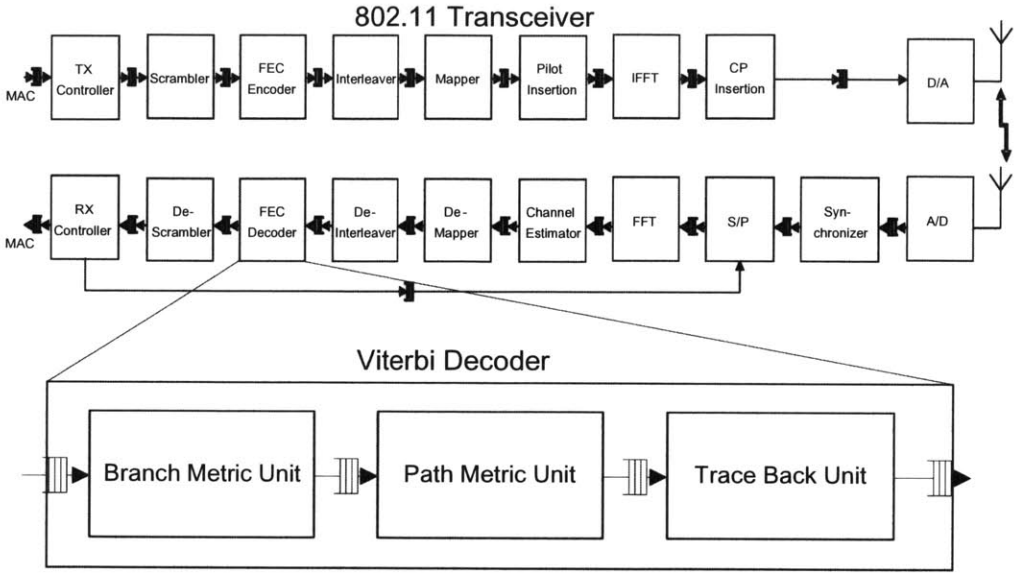


Figure 5-4: Architecture of Viterbi Decoder and testbench

decoder in a complete receiver pipeline to illustrate how external input activity affects the analysis.

5.3 Metrics for the Viterbi decoder

To obtain the statistical activity data for the Viterbi decoder, we used the AirBlue platform [55] to simulate the complete 802.11 receiver pipeline, as shown in Figure 5-4. Having a setup of the global architecture in which the decoder itself is a component, provides information on the frequency and *bursty* nature of the input to the decoder. This has an impact on the internal activity metrics - the average activity decreases and the length of inactivity intervals increases. The Viterbi decoder consists of three main modules: Branch Metric Unit (BMU), Path Metric Unit (PMU) and Trace Back Unit (TBU). Each of these modules has all its activity defined in one or two rules. Thus, the data shown in Tables 5.4(a) and 5.4(b) is able to capture all the granularity of activity with just four entries (rules *push_zeros* and *put_data* both correspond to BMU). The data was collected over a testbench of 32000 cycles of simulation taken at a steady state. For the case with input errors, we set the SNR at a low value under QAM modulation, such that the Bit Error Rate (BER) of the input data was

Table 5.4: Viterbi Activity Metrics

(a) Input data with BER = 0.5%

Rule	N_1: **No. of Inactive cycles**	N_1%: **Percentage inactivity**	N_2: **No. of Inactive-Active transitions**	N_1/N_2: **Avg. Inactivity interval**
Branch Metric Unit				
put_data	18079	56.5	156	115.9
push_zeros	30568	95.6	38	804.4
Path Metric Unit				
pmu_put	15649	48.9	156	100.3
Trace Back Unit				
tbu_put	15650	48.9	156	100.3

(b) Input data with BER = 0%

Rule	N_1: **No. of Inactive cycles**	N_1%: **Percentage inactivity**	N_2: **No. of Inactive-Active transitions**	N_1/N_2: **Avg. Inactivity interval**
Branch Metric Unit				
put_data	28376	88.7	151	187.9
push_zeros	31424	98.2	9	3491.5
Path Metric Unit				
pmu_put	27800	86.9	151	184.1
Trace Back Unit				
tbu_put	27800	86.9	151	184.1

reasonably high at 0.5%. This setup provides the Viterbi decoder with a significant amount of activity as the wireless receiver pipeline is dominated by the decoding effort. The inactivity metrics for various components of the Viterbi decoder are shown in Table 5.4(a). When compared with the large values of inactive cycles (N_1), the infrequency of inactive-active transitions (N_2) indicate that the inactivity intervals for the Viterbi decoder are quite long, much longer than those for the Reed-Solomon decoder, making it an attractive candidate for fine-grained gating even in conditions of low SNR and high activity.

The second testbench environment was set at a high SNR under BPSK modulation, giving an effective BER of zero. As shown in Table 5.4(b), under this scenario the Viterbi decoder has a higher number of inactive cycles, as other modules in the

pipeline are rate-limiting. The high value of the ratio of N_1 to N_2 emphasizes the long length of their inactivity intervals in this testbench emulating realistic traffic conditions. Based on this data, we can conclude that most of these rules would be excellent candidates for our proposed power gating scheme.

Thus, we see how the rule-based activity collection can inform decisions regarding choice of power domains to be gated in wireless decoder designs. In the decoder designs though the level of activity between modules is different, the overall streaming-based processing is relatively homogeneous. In the next chapter, we explore how such techniques can be applied in the case of much more complex designs where each module can have dramatically different activity profile.

Chapter 6

Larger and complex test cases

In this chapter, we explore two significantly larger and more complex designs than the earlier discussed wireless decoders. These test cases comprise of a million-point sparse Fourier transform design and a Reduced Instruction Set Computing (RISC) processor capable of booting Linux. Through these examples, we show how our technique scales and the benefits it provides for complex and heterogeneous hardware designs.

6.1 Million-point SFFT design

The first complex design chosen is a high-throughput implementation of a sparse Fourier transform that operates on a million (2^{20}) inputs that are frequency sparse, *i.e.*,only a few (in this case up to 500) frequency coefficients are non-zero. We first describe the design of the SFFT hardware and then provide various activity metrics for it.

6.1.1 Design overview

Processing million-point Fourier Transforms in real time can enable numerous applications ranging from GHz-wide spectrum sensing and radar signal processing to high resolution computational photography and medical imaging. Currently, million-point FFTs are not practical. Hardware implementations of such large FFTs are

prohibitively expensive in terms of high energy consumption and large area requirements. However, for most of the above applications the Fourier transform is sparse which means that only few of the output frequencies have energy and the rest are noise. Recent work [34] in the field of algorithms has shown how to compute these sparse FFTs (SFFT) in sub-linear time more efficiently than standard FFTs and using only a sub-linear number of samples.

At a high level, the SFFT algorithm has two main steps:

1. *Bucketization:* In this step, the algorithm maps the million (2^{20}) frequencies into 4096 buckets such that the value of each bucket is the sum of the values of 256 consecutive frequencies mapped to it. This is done by multiplying the input samples by a Gaussian filter and performing a 4096-point FFT. This bucketization is repeated for several iterations but in each iteration a permuted set of samples of the input is chosen. This permutation of time samples results in a permutation of the frequencies and randomizes the mapping of frequencies to buckets as described in [34].

2. *Estimation:* The algorithm then estimates the locations and values of the large frequency coefficients. To estimate the locations, the algorithm uses a voting based approach. At the output of the 4096-point FFT, it picks the buckets with the largest values. These buckets vote for the frequencies that map to them. A large frequency coefficient will get a vote in every iteration as the values of the buckets they map to are proportional to their own large value. A negligible frequency coefficient will not always get a vote due to the random mapping of frequencies to buckets. Thus, the frequencies with the most votes are the large frequency coefficients. The values of these frequencies are estimated from the values of the buckets they map to.

The SFFT algorithm enables computing a million-point Fourier transform faster than standard FFT if the output is sparse. However, published software implementations [33, 65] of SFFT algorithms are unable to achieve high input data rates, nor are they efficient from the perspectives of power, energy, unit cost or form factor. We

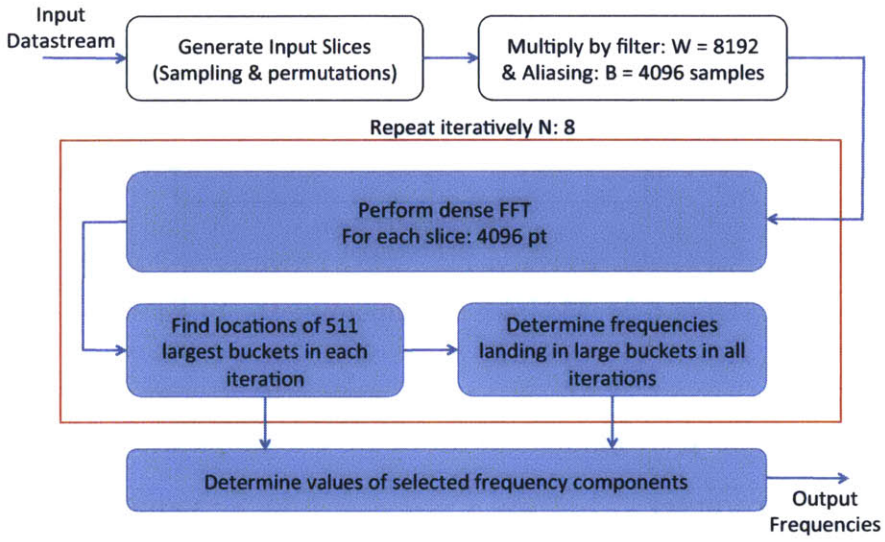


Figure 6-1: Stages of the SFFT algorithm. Core stages are highlighted in blue.

present the first hardware implementation of a million-point SFFT. We use Bluespec SystemVerilog [10], a high-level hardware design language for the design. Our design works in a streaming manner on 24-bit input samples to generate the locations and values of the largest 500 frequency coefficients in 4.49 milliseconds and hence can support input data rates of 2.23×10^8 samples per second. The SFFT algorithm version implemented in this work is robust with respect to the noise-level in the input. Our design is also reconfigurable for various sparse applications.

6.1.2 Design Architecture

The SFFT implementation consists of several modules, each implementing a distinct stage of the algorithm. Figure 6-1 shows the various stages involved in the SFFT algorithm.

In this section, we describe the implementation of four main stages of the algorithm. We have termed these stages collectively the SFFT Core, because they are responsible for the bulk of the computation and resource usage in the algorithm. Figure 6-2 shows the main modules used in our implementation of the SFFT Core and their input-output semantics. Our design has been parameterized to allow design exploration and to generate optimized results for the desired specifications. The

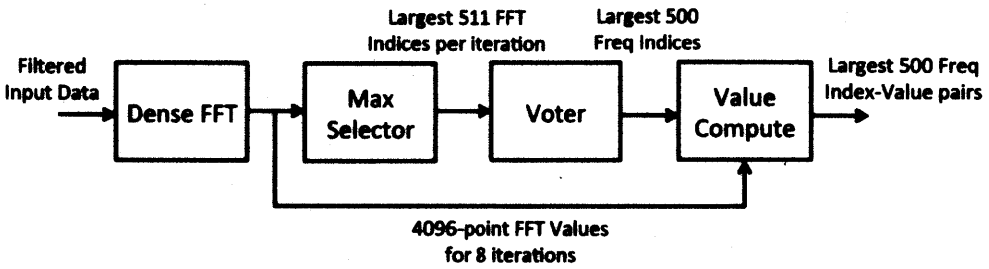


Figure 6-2: Modules implementing the SFFT Core

Table 6.1: Parameters for SFFT Core implementation

Parameter	**Value**
Input data type	Complex fixed-point
Fractional bits for fixed-point data	24
Total number of input data values	2^{20}
Maximum non-zero input frequency	500
Number of iterations in algorithm	8
Size of FFT in each iteration	4096

parameters chosen for the discussed implementation are given in Table 6.1.We chose the input data type to be complex fixed-point with 24 fractional bits for each of the real and imaginary components. The high number of fractional bits ensures that we have sufficient accuracy for various applications. The number of input samples is 2^{20}, thus each input sample has a 20-bit location index and a 64-bit value (accounting for real and imaginary sign bit, integral bit and six overflow bits). The input data is constrained to have a maximum of 500 non-zero frequency coefficients. Increasing the number of iterations and size of FFT in each iteration, increases the accuracy of the probabilistic SFFT algorithm. But it also increases the resource usage and time required for completion. We chose 8 iterations with a 4096-point FFT in each iteration, as this choice gave sufficient accuracy while still providing an achievable hardware target. Each module was targeted to achieve a minimum operating frequency of 100 MHz. We next describe the architecture of the SFFT Core modules in detail.

4096-point FFT

The SFFT algorithm requires taking a standard FFT of filtered input data slices, which we refer to as dense FFT. Our design of the dense FFT module was required to have a high throughput, low area and maximum size of the FFT possible. The larger the size of the FFT, the lower is the chance of collisions occurring due to non-zero frequency components being mapped to the same bucket. Initial attempts to use folded in-place FFT designs [25] failed, as they did not scale to a size beyond 512 points for the 24-bit input data. Instead, this implementation uses a fully pipelined streaming FFT architecture [35], utilizing a Radix-2^2 Single Delay Feedback. Each internal block of the FFT architecture is designed as shown in Figure 6-3, where N is a parameter that varies from 1 to 1024. The definition of the butterfly structures is shown in Figure 6-4.

Figure 6-5 shows how the internal blocks are instantiated with appropriate parameter values to generate the pipelined 4096-point dense FFT implementation. The use of pipelined multipliers for complex fixed-point input, and adequate buffering in FIFO queues between blocks allows the design to continuously stream data across iterations without any stalls. The twiddle factors W_i used in the computation were generated as a look-up table that each block can independently query for values. The indices for the twiddle factors are determined by the collective value of the 2-bit counters present in each block. Each block has two shift registers that map directly to FPGA shift registers. Absence of large multiplexers, which are usually present in folded in-place FFT designs, allows this implementation to be highly efficient in FPGA resource usage. The design is parameterized for the input data type, FFT size and amount of pipelining in the complex multipliers.

Max Selector

In the previous stage, by performing the 4096-point FFT on the input data we have mapped the 2^{20} input frequencies into 4096 buckets. This stage of the sparse FFT algorithm requires determining which of these buckets have a large magnitude,

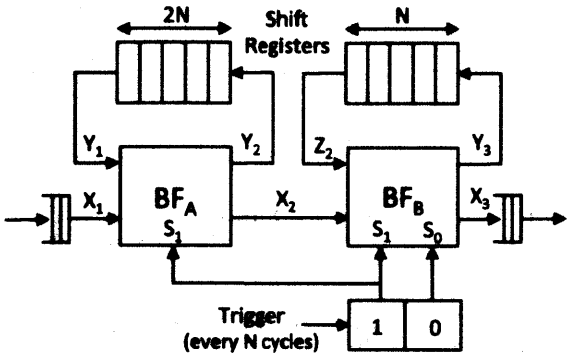


Figure 6-3: Single parameterized block of streaming FFT

```
if S1 == 1
    X2 = X1 + Y1
    Y2 = X1 - Y1
else
    X2 = Y1
    Y2 = X1
```

(a) BF_A

```
if S0 == 1
  if S1 == 1
    X3 = X2 + Z2
    Y3 = X2 - Z2
  else
    X3 = X2 - jZ2
    Y3 = X2 + jZ2
else
  X3 = Z2
  Y3 = X2
```

(b) BF_B

Figure 6-4: Definition of butterfly structures in R2^{2}SDF design

indicating that one or more of the frequencies mapped are non-zero. Selecting buckets by setting a threshold would have been sensitive to noise levels in input and hence, not robust. Sorting all the FFT outputs to generate the ordered magnitudes was observed to be highly resource intensive and time consuming, as well as overkill since the algorithm does not require them to be ordered. Instead, we implement this step by selecting the largest (but unordered) 511 magnitudes of the 4096-point FFT output for each iteration. The chosen selector architecture operates on $2^n - 1$ entries, hence the number of entries being 511. Since the input data has a maximum of 500 non-zero frequency coefficients, selecting top 511 buckets by magnitude was sufficient to collect

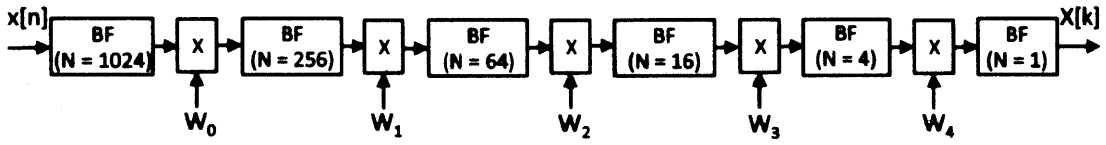


Figure 6-5: Architecture of the 4096-point dense FFT

information for all of them.

Figure 6-6 shows the implementation of the Top-511 element selector. The design is based on a pipelined heap priority queue architecture [39]. The magnitudes of the FFT outputs are tagged with their indices and inserted into a binary tree structure with 511 entries. Entries are addressed from 1 to 511, with the k^{th} entry being the parent of (and smaller in magnitude than) $2k^{th}$ and $2k + 1^{st}$ entries. Thus, the 1^{st} entry, called the root, has the smallest magnitude of all the entries in the tree. The entries are divided into stages, each stage containing indices from 2^n to $2^{n+1} - 1$. This allows write operations to be localized and prevents generation of large multiplexers during FPGA synthesis. New elements are filled into the tree in ascending order of addresses, pushing larger entries down and maintaining the root as the minimum entry.

Once the first 511 FFT outputs are inserted into the tree, further streaming inputs are compared with the root to check whether they are larger than the root. Only if they are, they replace the root element and checks are done to maintain parent children relationships in the tree. The queues between stages of the binary tree allow a pipelined design that can allow insertions into an unfilled tree every cycle, and replacements in a filled tree every alternate cycle. At the end of processing all 4096 FFT outputs, the tree contains the largest 511 magnitudes, each tagged with their corresponding location in the FFT output. The module output is a 4096-bit vector, with a high bit for each of the 511 selected FFT outputs.

Voter: Locating the top frequencies

This stage receives as input eight 4096-bit vectors with the large buckets indicated by the bits set to 1. Each of these buckets constitutes a set of 256 candidate

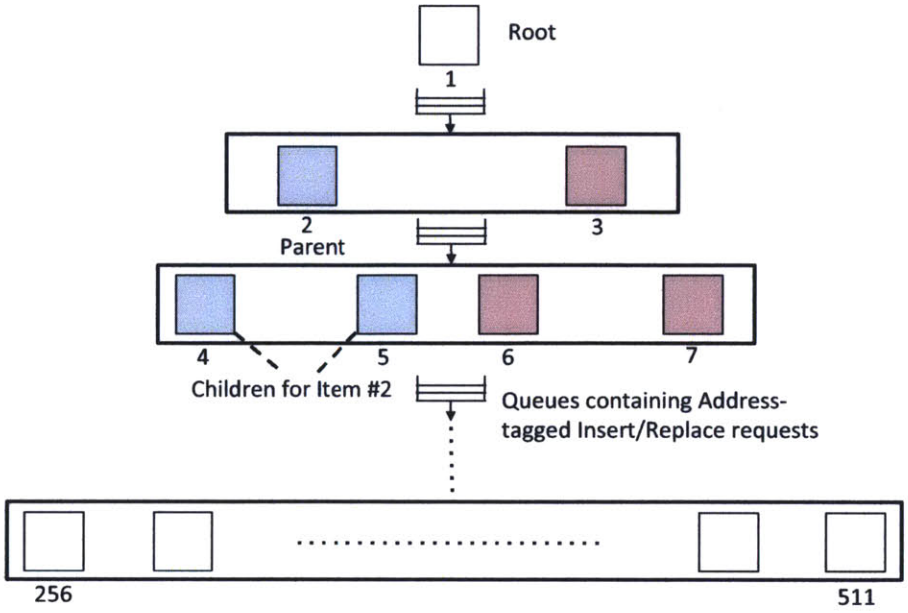


Figure 6-6: Architecture of the Max Selector module

frequencies that were mapped to this bucket using randomized permutations. This stage determines the frequency indices that have landed in top buckets in all iterations. The process can be understood as eight rounds of voting, with each iteration incrementing the votes of 511×256 distinct frequency indices and the final selection of indices with eight votes.

Implementing the stage as a naive iterative voting structure would have required book-keeping of nearly a million votes spread across a million candidate frequency locations. Implementing such a data structure in FPGA with the reading, writing and comparison of the votes for all candidates occurring within the required performance constraints is impractical. Since the values of σ chosen for each iteration are statically known, it is conceivable that a table of values can be generated that provides the static condition for each frequency index to be non-zero. But, this also runs into the issue of reading and comparing values from an extremely large data structure.

Instead, our implementation is a novel pipelined filtering process where we track candidate frequencies, instead of keeping track of votes. Candidate frequency indices are generated by the first iteration in a stepwise manner and are passed through seven filters. Each filter maps the incoming candidate frequency to the appropriate bucket

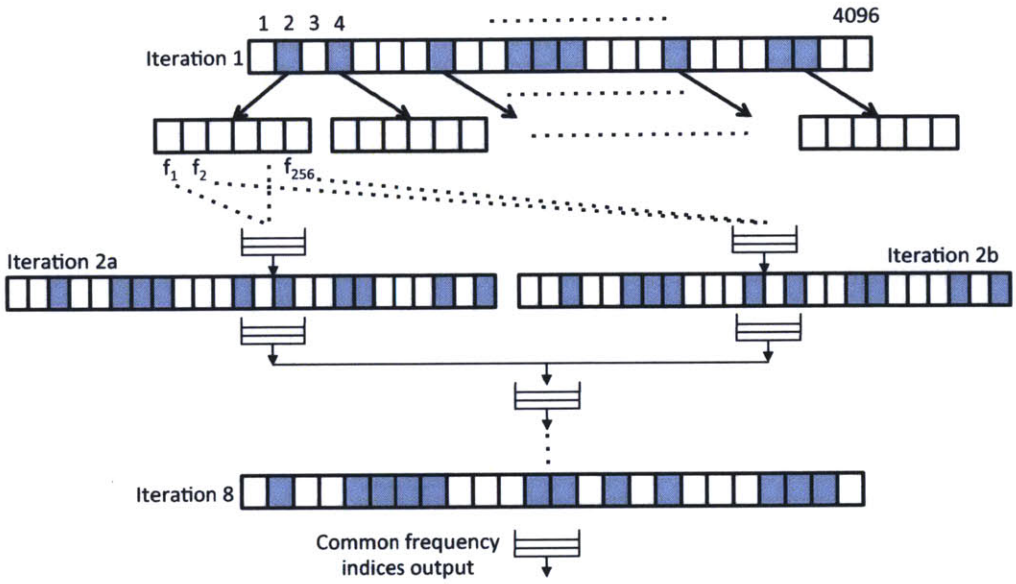


Figure 6-7: Architecture of the Voter module implemented as a series of filters. Large buckets for each iteration have been highlighted in blue.

for that iteration and checks if it lands in a top-511 bucket in the corresponding bit vector. If it does land in a large bucket, it is sent to the subsequent iteration otherwise it is discarded. The mapping functions are unique for each iteration and relate to the permutations used for generating the input data slices. At the end of the filtering process, only those frequency indices are passed through to the next stage which have been present in top buckets for all iterations. In order to further improve the throughput of this stage, we parallelized the processing of candidate frequencies by the second stage. For this, we duplicated the filter for the second iteration, seen as $2a$ and $2b$ in Figure 6-7, mapping 128 odd-indexed frequencies to $2a$ and 128 even-indexed frequencies to $2b$ for each high bucket in the first iteration. This decreased the number of cycles required to process the data by nearly 50% as most of the candidate frequencies get filtered out at the second iteration itself.

Value Compute

This stage is responsible for computing the value of the frequency components that have been determined by the previous stages. FFT outputs produced for all eight

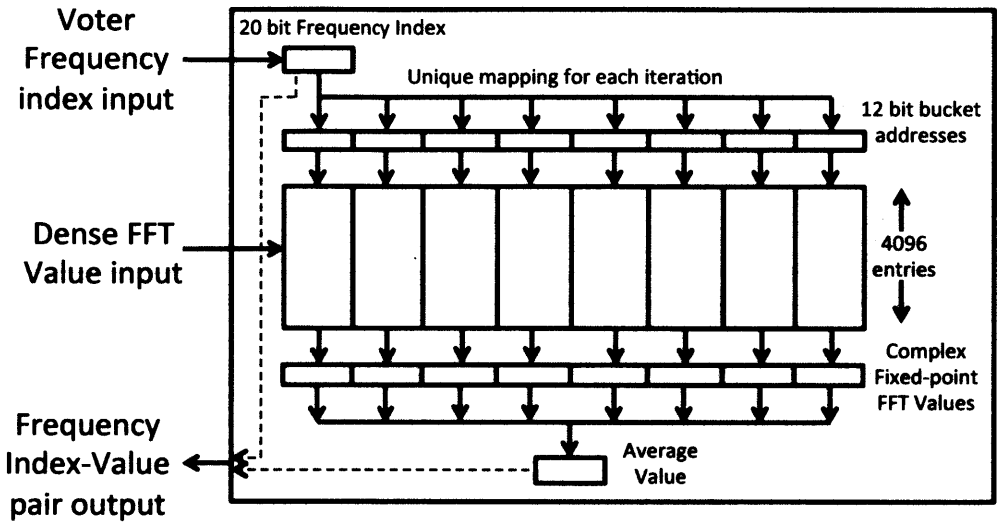


Figure 6-8: Architecture of the Value Compute module

iterations are forwarded to the Value Compute module. They are stored locally in a Block-RAM memory structure with eight memory banks, which allows simultaneous processing of read requests from all eight banks.

The frequency locations determined to be non-zero in the input, obtained from the voter module, are mapped to their respective bucket locations for each iteration. Corresponding FFT outputs are read from the memory banks for each location. These eight bucket values for each selected frequency are then averaged to obtain the final value of the frequency components. Figure 6-8 shows the architecture designed for this module.

SFFT Core

The use of high-level Bluespec interface specifications for component modules allows elegant plug and play generation of complex designs. This enabled us to quickly connect the various modules to generate the SFFT Core design. During design exploration, we had to modify the internal structure of various modules as a result of algorithmic modifications or for meeting resource and performance constraints. However, generating the overall design was straightforward once the component modules were complete. Adequate buffers were added to latency insensitive FIFO queues be-

tween modules to allow individual blocks to run at different rates without stalling the entire design. This was beneficial for the overall performance since several modules produced outputs sporadically and in a data-dependent manner.

Design Performance

The SFFT Core was designed using BSV with a focus on parametrization and latency insensitive interfaces in the architecture to enable extensive design exploration. The design was synthesized, placed and routed for Xilinx ML605 platform with the target device as XC6VLX240T FPGA and the target clock frequency as 100 MHz. The mapped design's output was compared with MATLAB results to verify correctness. Table 6.2 shows the percentage Virtex-6 FPGA resource utilization of various modules of the implementation as well as the complete SFFT Core. Different modules have varying resource requirements due to the wide variety of designs. The Dense-FFT module has the highest DSP slice utilization, due to instantiating multiple pipelined multipliers for the complex fixed-point data. The Max Selector module has the maximum LUTs slice utilization, 18% of the total, due to the logic generated for reading and writing various entries in each internal stage. The Value Compute module has the maximum BRAM utilization of 14% for storing the FFT outputs for all eight iterations. Overall, as seen in the data, the Core fits well within the resources of a single Virtex-6 FPGA with 24% Slice Registers, 48% Slice LUTs, 26% BRAMs and 16% DSP48E slice utilization.

Table 6.3 shows the latency and throughput of various modules of the implementation as measured in FPGA clock cycles with an operating clock frequency of 100 MHz. The latency of the design is defined as the number of cycles taken from providing the first input data sample to a module to receiving the last output data sample from it. The throughput is defined, under steady state conditions with continuous input supply, as the number of cycles between the first output of the first input dataset to the first output of the second dataset. We have given the per-iteration and total number of cycles for the FFT and Max Selector modules. For the FFT module, the total latency is less than 8× the single iteration's latency because the pipelined ar-

Table 6.2: FPGA Resource utilization, shown as a percentage of total resources available on Xilinx FPGA XC6VLX240T.

Design	**Regs**	**LUTs**	**BRAMs**	**DSP48Es**
Dense-FFT	3%	11%	8%	11%
Max Selector	8%	18%	0%	0%
Voter	10%	14%	3%	2%
Value Compute	1%	2%	14%	2%
SFFT Core	**24%**	**48%**	**26%**	**16%**

Table 6.3: Latency and throughput in FPGA clock cycles

Design		**Latency** (cycles)	**Throughput** (cycles)
Dense-FFT	1 iteration	13682	6826
	Total	61468	54612
Max Selector (Avg)	1 iteration	5888	5888
	Total	47104	47104
Voter (Avg)	Total	68816	68816
Value Compute	Total	33788	33788
SFFT Core (Avg)	Total	**138646**	**116024**

chitecture allows overlapping execution. The performance of the Max Selector and Voter modules is data-dependent. We have shown the average case numbers for these modules, and similarly for the entire SFFT Core design. For the SFFT Core, the total number of cycles per transform under steady state is significantly less than the sum of all individual components due to the pipelined nature of the architecture that allows overlapping computation between various modules.

We synthesized, placed and routed individual modules as well as the complete design to obtain processing times for each component. Table 6.4 shows the evaluated processing times using each module's maximum operational frequency. For the complete SFFT core, we run the entire design on a single clock frequency. The critical path of the design lies in the Voter module, specifically in the filter modules that check whether a candidate frequency falls in a high bucket for the corresponding iteration. This check requires mapping the candidate to a bucket and then comparing with a single value out of the 4096-point vector. We store this vector in Block RAMs, using

Table 6.4: Processing times in milliseconds accounting for the maximum operating frequency of the component.

Design	**Steady-state Throughput (cycles)**	**Maximum frequency (MHz)**	**Processing Time (ms)**
Dense-FFT	54612	121.2	0.45
Max Selector	47104	134.7	0.35
Voter	68816	100.1	0.69
Value Compute	33788	121.9	0.28
SFFT Core	**116024**	**100**	**1.16**

a vector size of 64 that balances the number of cycles to initialize the filter and the size of the multiplexer that selects the appropriate index out of each 64-point value. This is a significantly better solution than use of Slice Registers, as it reduces the design congestion and allows routing to be completed with desired time constraints.

The steady state throughput of our SFFT Core design is 116,024 clock cycles with a 100 MHz clock, which translates to 1.16 milliseconds per million-point sparse Fourier transform. This design is the first million-point FPGA implementation of the SFFT algorithm. The initial single-threaded software implementation [33] of the algorithm takes 190 milliseconds to complete a transform with the same parameters, while executing on an Intel Core i7-2600 CPU. A recent multi-threaded software implementation [65] takes 100 milliseconds for the same problem size, while executing on an Intel Xeon E5-2660 CPU. Though the CPU-based designs work on floating-point data, our fixed-point FPGA implementation is accurate enough for applications under consideration due to the large number of fractional bits used in the input data type. Our FPGA design is 85× faster than the latter CPU implementation, while having the benefits of operating in a single FPGA form factor and power budget as compared to that of a multi-core CPU.

6.1.3 SFFT design activity metrics

The activity metrics for each rule in the component modules of the SFFT design were computed using the average case scenario discussed earlier, over 100,000 cycles

of simulation covering one million-point SFFT computation.

It is seen that the streaming pipelined architecture of the 4096-point FFT creates an ideal profile in terms of the dynamic activity metrics for power gating shown in Table 6.5. Each sub-module block in this pipeline becomes active continuously while processing the FFT of the eight iterations in one million-point SFFT and then becomes inactive for a long duration until the start of the next SFFT. There is only a single switching of states from inactive to active in each SFFT, as seen in the N_2 metric being equal to one for all rules in the 4096-point FFT.

For the Sorter module, the number of inactive intervals is much smaller than the number of inactive cycles for each of the rules giving a very large average inactivity interval length for all of them. It is important to note that different rules would become active in different cycles independently, thus an independent fine-grained gating scheme would allow significantly more opportunities to power gate than the module gated as a whole.

For the voter module, we notice that each of the iteration filter rules has a different inactivity interval length, with monotonic increase in length as the stage indices increase. This follows from the architecture where most candidate frequencies get eliminated from the top two filter stages. Each following stage gets fewer and fewer candidates, resulting in increased inactivity and increased inactivity interval length.

The Magnitude module follows a similar behavior to the FFT module, with similar intervals for three of the component rules. The values themselves are higher than that for FFT due to total inactivity being higher. Thus the domains have higher expected power savings for typical breakeven thresholds.

We will now discuss the behavior of the SFFT design under coarse-grained gating, *i.e.*,domains encompassing all datapath logic of each module. During simulation, if any rule of a module was activated the single coarse-grained domain per module was considered active. The power domain was switched off only if all rules of the module did not fire in a given cycle. This occurs when the module was waiting for data from earlier modules in the pipeline, or when the module's output channels are completely full. Under such coarse-grained gating, the total number of inactive cycles (N_1) drops

Table 6.5: SFFT Activity Metrics

Rule	N_1: **No. of Inactive cycles**	N_1%: **Percentage inactivity**	N_2: **No. of Inactive-Active transitions**	N_1/N_2: **Avg. Inactivity interval**
4096-point FFT Module				
doStage0	75098	75.1	8300	9.05
getMultTwiddle0	75099	75.1	8300	9.05
doStage1	75101	75.1	8300	9.05
getMultTwiddle1	75102	75.1	8299	9.05
doStage2	75104	75.1	8298	9.05
getMultTwiddle2	75105	75.1	8298	9.05
doStage3	75107	75.1	8298	9.05
getMultTwiddle3	75109	75.1	8297	9.05
doStage4	75111	75.1	8297	9.05
getMultTwiddle4	75113	75.1	8296	9.05
doStage5	75115	75.1	8295	9.06
Sorter Module				
initialize	99992	99.9	8	12499.00
insertItem	47596	47.6	8	5949.50
replaceItem	54484	54.5	7	7783.43
fillOutputQ	97960	98.0	8	12245.00
Voter Module				
initialize	99999	99.9	1	99999
genBuckets	99944	99.9	56	1784.71
processBuckets	43604	43.6	1	43604
iter0ato1	92527	92.5	7473	12.38
iter0bto1	93841	93.8	6159	15.24
iter1to2	96051	96.1	3949	24.32
iter2to3	98685	98.7	1315	75.05
iter3to4	99627	99.6	893	111.56
iter4to5	99563	99.6	437	227.83
iter5to6	99563	99.6	437	227.83
fillOutputQ	99564	99.6	436	228.36
Magnitude Module				
process_req	99564	99.6	436	228.36
recv_mem_resp	99564	99.6	436	228.36
send_result	99564	99.6	436	228.36
get_fft_vals	75117	75.1	8294	9.06

Table 6.6: SFFT Module-level Inactivity percentage

Module	N_1%: Percentage inactivity
4096-point FFT	58.5
Sorter	10.1
Voter	43.6
Magnitude	73.8

significantly as compared to the values of the individual rules in these modules. This is because the condition for inactivity is a lot more restrictive for module-level gating. This is seen in the lower values of percentage inactivity for SFFT modules shown in Table 6.6. In particular, the percentage inactivity of the Sorter module is only 10.1%, while each of its individual rules has at least 45% inactivity (seen in Table 6.5). Since these rules become active in different cycles, the overall inactivity of the module drops significantly. At the same time, the breakeven threshold is much larger as the size of the domain and switching logic increases. This also increases the slowdown in clock frequency. Thus, we can see that for such designs, fine-grained gating is more appropriate and can generate significant savings at lower energy and performance costs.

6.2 RISC processor design

This design comprises of a 32-bit RISC microarchitecture that implements the MIPS I ISA. It includes a multiply unit, coprocessor 0, where MIPS I implements the data and instruction TLBs, independent instruction and data L1 caches, and a unified, N-way L2 cache. This 5-stage processor can boot the GNU/Linux kernel [1].

6.2.1 Design overview

Figure 6-9 shows the architecture of the RISC processor. The PC, F, D, E, WB and RF sub-modules are part of the Processor module, while the other components shown in Figure 6-9 are instantiated as separate modules. The architectural details

[1]Primary work on creating this design was done by Oriol Arcas Abella.

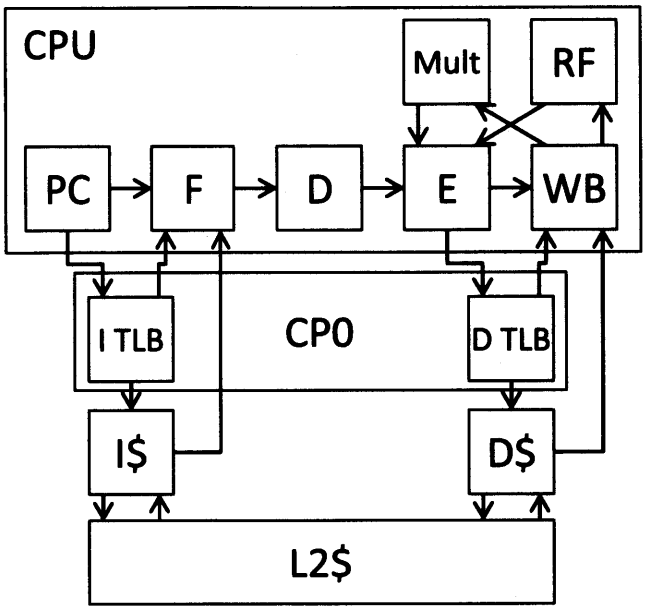


Figure 6-9: Architecture of RISC processor

are skipped here as the design has a conventional structure for a pipelined multi-stage processor. Our goal here is to show how the design activity metrics change when observed in a test-case which is quite different from the earlier special purpose hardware designs for algorithmic blocks in the wireless domains.

6.2.2 RISC processor design activity metrics

We computed the activity metrics for each rule in the component modules of the RISC processor under dynamic conditions. The data shown in Tables 6.7 and 6.8 correspond to the activity generated during a Linux boot sequence. The data was collected over a testbench of 10 million cycles of simulation.

It is observed that there is a wide range of values of the average inactivity interval in the component modules. While the total inactivity is quite high for most of the rules in all the modules, from 76% to 99%, the number of transitions is dependent on the specific module itself. The Multiplier module for example, sees infrequent activity, and has a very large inactivity interval. The Processor-CPU module itself has much shorter inactivity intervals due to frequent state switching, and thus it would be

Table 6.7: RISC Processor Activity Metrics - part 1

Rule	**N_1: No. of Inactive cycles**	**N_1%: Percentage inactivity**	**N_2: No. of Inactive-Active transitions**	**N_1/N_2: Avg. Inactivity interval**
Processor-CPU Module				
pcgen	7665660	76.6	1088024	7.05
fetch	7665660	76.6	1088023	7.05
dec	7665660	76.6	1089905	7.03
exec	7938181	79.4	1219321	6.51
writeback	7938181	79.4	1209206	6.56
update_rf	8786735	87.8	782802	11.22
update_pc	9998444	99.9	1556	6425.74
kill	9727479	97.3	137094	70.95
Multiply Module				
start_mult	9999956	99.9	44	227271.73
mult_step	9998592	99.9	44	227240.73
step_up	9998592	99.9	44	227240.73
upd_hilo	9998548	99.9	44	227239.73
Data Cache Module				
proc_req	9216250	92.2	782096	11.78
req_from_mem	9999652	99.9	348	28734.63
resp_from_mem	9872425	98.7	127575	77.39
access_mem	9999652	99.9	348	28734.63
comp_tag	9216250	92.2	782096	11.78
access_proc	9216250	92.2	782096	11.78
refill_resp	9872425	98.7	127575	77.39
update_wait	9872425	98.7	127575	77.39
st_wait	9551793	95.5	448207	21.31
evict	9999652	99.9	348	28734.63
Instruction Cache Module				
proc_req	7665660	76.7	1088024	7.05
req_from_mem	9999609	99.9	391	25574.45
resp_from_mem	9989732	99.9	10268	972.90
access_mem	9999609	99.9	391	25574.45
comp_tag	7665660	76.7	1088024	7.05
access_proc	7665660	76.7	1088024	7.05
refill_resp	9989732	99.9	10268	972.90
update_wait	9989732	99.9	10268	972.90
evict	9999609	99.9	391	25574.45

Table 6.8: RISC Processor Activity Metrics - part 2

Rule	N_1: **No. of Inactive cycles**	N_1%: **Percentage inactivity**	N_2: **No. of Inactive-Active transitions**	N_1/N_2: **Avg. Inactivity interval**
Data TLB Module				
TLB_matching	9216250	92.2	782096	11.78
translation	9216250	92.2	782096	11.78
store_paddr	9216250	92.2	782096	11.78
translate_addr	9216250	92.2	782096	11.78
write_Entries	10000000	100	1	10000000
Instruction TLB Module				
TLB_matching	7665660	76.7	1088024	7.05
translation	7665660	76.7	1088024	7.05
store_paddr	7665660	76.7	1088024	7.05
translate_addr	7665660	76.7	1088024	7.05
write_Entries	10000000	100	1	10000000
L2 Cache Module				
access_IResp	9989806	99.9	10194	979.97
access_DResp	9872498	98.7	127502	77.43
access_IReq	9989732	99.9	10268	972.90
access_DReq	9872425	98.7	127575	77.39
do_resp	9862304	98.6	137696	71.62
comp_tag	9862157	98.6	137843	71.55
replace_wait	9997046	99.9	1478	6763.90
refill_resp	9873311	98.7	126689	77.93
update_wait	9862157	98.6	137843	71.55

harder to find significant savings in this module. This is along expected lines as it is the central module generating work for the other modules of the system. Cache modules have different values for requests and responses as the latter get bunched together when being served from a cache line. Thus it would be better to power gate the response logic while keeping request logic ungated. Similarly, for each module, we can select appropriate sub-modular blocks with high total inactivity and high average inactivity interval lengths to obtain the viable fine-grained power domains for each module.

Table 6.9 shows the module-level inactivity percentage for the RISC processor design. It is seen that the coarse-grain module-level domains would have lower in-

Table 6.9: RISC Processor Module-level Inactivity percentage

Module	N_1%: **Percentage inactivity**
Processor-CPU	70.4
Multiply	99.9
Data Cache	82.9
Instruction Cache	65.6
Data TLB	84.3
Instruction TLB	65.8
L2 Cache	91.8

activity than individual rule-level domains discussed earlier. While computing these metrics, we removed the contribution of a few book-keeping rules in these modules that were frequently active. This was done to avoid skewing the module-level inactivity percentage even lower and to maintain a fair comparison to the rule-level metrics shown.

In this chapter, we discussed complex designs where the level of activity differs widely at a modular level as well as at the rule-level within modules. We observed that coarse-grained modular activity was significantly higher than the activity of the individual rules. This demonstrated the benefit of pursuing fine-grained gating where individual blocks within a module can be turned off even if there is activity in other blocks of the module. In the next chapter, we will discuss the power and performance impact of such gating.

Chapter 7

Breakeven threshold and impact of fine-grained gating

In this chapter, we discuss the overall impact of our technique on the net power consumption and performance of the Reed-Solomon decoder, Viterbi decoder and SFFT accelerator designs, all of which are examples of digital signal processing under strict energy and processing time constraints. We evaluate the breakeven threshold for inactivity of the fine-grained power domains under consideration, and compare it to the discussed activity metrics in the last two chapters.

7.1 Leakage power consumption

We begin the analysis by looking at the total power consumption of the three designs and the fraction consumed by leakage. Table 7.1 shows a breakdown of the power consumption. It is seen that the fraction of total power consumed as leakage in decoder designs is quite significant: 44% in Reed-Solomon decoder and 41% in Viterbi decoder. For the larger SFFT design, the amount of leakage power consumed over representative testbench is large, though the percentage fraction is smaller (34%) due to higher rate of average activity to meet performance constraints. The data was generated by simulating the extracted place and routed netlists with realistic testbenches. This process also generates the breakdown of the leakage power for each

Table 7.1: Power consumption breakdown of Reed-Solomon decoder, Viterbi decoder and SFFT design. The synthesis was done using Nangate 45nm library with the operating frequency set to 100MHz. Power consumption data was obtained using simulation of the placed and routed designs.

Design	Component	Power (mW)
Reed-Solomon decoder	Dynamic	8.62 (56%)
	Leakage	**6.74 (44%)**
	Total	15.37
Viterbi decoder	Dynamic	7.96 (59%)
	Leakage	**5.37 (41%)**
	Total	13.33
Sparse FFT design	Dynamic	305.9 (66%)
	Leakage	**157.6 (34%)**
	Total	463.5

logic element of the design.

The fraction of leakage power increases even further depending on a number of factors such as operating temperature, technology node and relative external activity. This provides a strong motivation to explore methods for reducing leakage power in such designs. Next, we need to determine the number of clock cycles for which a typical fine-grained logic block needs to remain inactive, in order to recoup the energy lost in switching on the power domain and compare it to the dynamic activity metrics obtained from simulation.

7.2 Breakeven threshold of inactivity interval

To determine the switching energy cost per transition and the breakeven threshold of inactivity, we performed circuit-level SPICE simulations of typical logic blocks with power switches and isolation logic using NCSU 45nm library and Nangate Open Cell library. We chose three sizes for the power domains and appropriately sized the power switches to balance the voltage drop and turn-on time. Table 7.2 shows a brief summary of the analysis determining the leakage power saved by gating typical logic blocks, breakeven time to generate net savings and the impact on output signal propagation due to insertion of power switches and isolation cells.

Table 7.2: Computing breakeven threshold of inactivity interval, and the performance impact for gating typical logic blocks

(a) Characterization for logic block consisting of eight 2-input NAND gates with x2 drive strength

Property tested	**Value**
Leakage power of ungated logic block: a	432.60 nW
Leakage power of the block after gating with a 4/1 high V_{th} PMOS: b	0.21 nW
Leakage power saved by power gating the block: $\alpha = (a - b)$	432.39 nW
Energy lost in turning on the power switch and power domain: β	20.8 fJ
Breakeven time for net power savings in a single inactive interval: β/α	48.1 ns
Breakeven threshold in clock cycles at 100 MHz	5 cycles
Increase in output propagation delay (as measured by 50% value)	0.21 ns
Increase in output rise time (as measured by 10%-90% delay)	1.50 ns

(b) Characterization for logic block consisting of sixteen 2-input NAND gates with x2 drive strength

Property tested	**Value**
Leakage power of ungated logic block: a	879.5 nW
Leakage power of the block after gating with a 8/1 high V_{th} PMOS: b	0.38 nW
Leakage power saved by power gating the block: $\alpha = (a - b)$	879.12 nW
Energy lost in turning on the power switch and power domain: β	45.6 fJ
Breakeven time for net power savings in a single inactive interval: β/α	51.9 ns
Breakeven threshold in clock cycles at 100 MHz	6 cycles
Increase in output propagation delay (as measured by 50% value)	0.23 ns
Increase in output rise time (as measured by 10%-90% delay)	1.61 ns

(c) Characterization for logic block consisting of thirty-two 2-input NAND gates with x2 drive strength

Property tested	**Value**
Leakage power of ungated logic block: a	1738.40 nW
Leakage power of the block after gating with a 8/1 high V_{th} PMOS: b	0.51 nW
Leakage power saved by power gating the block: $\alpha = (a - b)$	1737.89 nW
Energy lost in turning on the power switch and power domain: β	68.3 fJ
Breakeven time for net power savings in a single inactive interval: β/α	39.3 ns
Breakeven threshold in clock cycles at 100 MHz	4 cycles
Increase in output propagation delay (as measured by 50% value)	0.27 ns
Increase in output rise time (as measured by 10%-90% delay)	1.69 ns

The leakage power saved by gating was obtained by comparing the power dissipation with and without the power switch. The energy cost of turn-on is the total energy drawn from the voltage supply, and it is evaluated by integrating the supply current over the turn-on time. As seen in the data, by dividing the switching energy cost (β) with the leakage power saved by gating (α), we arrive at the breakeven time period. The values of the breakeven time period for the three sizes of domains are: 48.1 ns, 51.9 ns and 39.3 ns. This indicates that the minimum length of inactivity interval required to compensate for the energy lost in switching off the domain is 4-6 cycles at a clock frequency of 100 MHz. This reflects well on our previous analysis of the example designs which had several logic blocks having much longer average inactivity intervals, thus providing significant power savings. For each technology library used, designers can similarly determine values of α and β for typical logic blocks and use them with the dynamic activity metrics computed for the rule-based design to select appropriate logic blocks to be power gated.

7.3 Performance Impact

For our gating methodology, the power domains need to be turned on or off within a clock cycle. This places a requirement on the clock cycle time to be long enough to accommodate the time required to turn on a power domain and complete logic computation. For the fine-grained power domains under consideration in the design testcases, the impact of gating is seen in Table 7.2 as an increase in propagation delay of 0.31 ns (for the output to reach 50% of the max value) and a 1.5 ns increase in output rise time (for the output to rise from 10% to 90% of the max value). After adding this delay to the prior critical path of the designs, we still had sufficient margin to keep the clock frequency at 100 MHz to allow such gating and maintain the performance requirements of wireless standards compliance.

It is possible that for designs with a high clock frequency requirement, insertion of such gating might necessitate increasing the cycle time. Such performance impact can be mitigated by selective addition or removal of power gating from the overall

architecture. Since power gating is defined in a fine-grained manner, it is possible to eliminate gating from the logic on the critical path of the clock. It is also expected that some critical logic would be mostly active and hence would not be a viable candidate for fine-grained power gating anyway.

7.4 Qualifying domains for the design testcases

In the designs under consideration, most rules have small logic blocks which fall within the size range discussed in the earlier analysis. Accordingly, we can consider the breakeven threshold for such blocks to be within the same order of 4-6 cycles. For our analysis, we kept the threshold at 6 cycles of inactivity after which power-gating the domain, on average, would reduce energy consumption. With this threshold, we can determine which modules and sub-modular blocks in each design are good candidates for fine-grained power gating.

For the Reed-Solomon design, only 3 rules (out of the total of 17 rules in the complete design) have a smaller average inactivity interval than this threshold (in both input data scenarios). After accounting for the logic corresponding to these 3 rules, 90% of the datapath logic in Reed-Solomon design can be power gated. For the Viterbi design, though the rules are larger with greater logic per rule body, the average inactivity interval for all rules is greater than 100 cycles under all conditions. Thus, all of the datapath combinational logic for this design would qualify for gating under our technique.

For the SFFT design, we can see that all the rules have the average inactivity interval greater than 9 cycles. Thus, we can include all the datapath logic for power gating for this design as well. A similar case holds true for the RISC processor design, where all rules have their average inactivity interval length greater than the 6 cycles threshold and qualify for fine-grained power gating. In these designs, the effect of the gating on the critical path of designs was within acceptable bounds with the increased delay being absorbed in the existing slack in cycle time at 100 MHz.

7.5 Energy Savings

7.5.1 Cycle by cycle profile

In the first cycle when a power domain turns off, the effective supply voltage V_{DDeff} slowly decays due to the residual leakage currents. The net leakage currents drawn from the voltage supply are immediately reduced by a large percentage (greater than 99% in the data shown in Figure 7.2) due to the presence of high threshold power switch. The energy savings are dictated by the reduced leakage current, which is roughly equal to the initial value in the first cycle of inactivity and decreasing slightly in subsequent inactive cycles as the V_{DDeff} decreases. The activation energy cost is spent in the first active cycle of the power domain after being inactive in previous cycles, with the net savings determined by the total inactivity and the number of inactive-active transitions.

7.5.2 Net reduction in power consumption

The overall leakage power reduction achieved by our technique can be estimated in the following manner. Leakage power dissipation has these main components as driving factors: 1. inactive combinational logic used for state updates, 2. the state elements, and 3. the clock tree and control logic. Our technique targets the elimination of leakage power consumed by inactive combinational logic. We categorized the various leakage power components manually by examining the digital power simulation output of the designs under various activity inputs. Based on this analysis, we estimate that 40% of the total leakage power is due to the first component, of which 90% of the domains qualify for power gating after accounting for activity metrics and breakeven thresholds. Over the course of the testbenches, the switching costs of gating the domains reduce the net savings to about 90% of their leakage power consumption. Given that the decoder designs had up to 44% power as leakage dissipation (Table 7.1), we estimate that our technique can save up to 14% ($= 0.44 \times 0.9 \times 0.40 \times 0.9$) of the total power consumption without any power-saving design burden on the user.

In this chapter, we characterized the impact of our technique on individual fine-grained domains as well as the complete wireless designs under consideration. We demonstrated the use of the obtained activity metrics to compare against computed breakeven thresholds for obtaining viable power domains from a rule-based design. In the next chapter, we present a discussion of C-based design methodology and a comparison of hardware quality vis-a-vis rule-based design.

Chapter 8

Comparison of high-level languages for hardware design

The increasing use of high-level synthesis (HLS) for hardware design provides opportunities and challenges in the domain of power-efficient design. By raising the level of abstraction, HLS has the potential to make the designer's intent explicit for use in automation of power domain partitioning. Hardware design can be expressed at a high-level with the centers of activity explicitly described. As described in this thesis, such description can lead to a natural translation to fine-grained power domain specification of the designs.

The questions to be explored in this chapter are:

1. Which high-level hardware design language can give us a direct relationship between specification and fine-grained domains?
   The two design methodologies that we will compare are software-based HLS and rule-based HLS.

2. Is there a penalty in terms of resource consumption or performance compromise to the use of HLS for hardware design?

3. To what extent can the earlier described rule-based power gating techniques be applicable in software-based design?

8.1 Software-based design tools

DSP community perceives several advantages in using a C-based design methodology [72, 31] — having a concise source code allows faster design and simulation, technology-dependent physical design is isolated from the source and using an untimed design description allows high-level exploration by raising the level of abstraction. Several EDA vendors provide tools for this purpose [51, 73, 13, 29, 41].

C-based tools fall into two distinct categories - those that adhere to pure C/C++ semantics like Catapult-C [51], PICO [73] and C-to-Silicon Compiler [13], and those that deviate from the pure sequential semantics by allowing new constructs, like SpecC [29], SystemC [57] and BachC [41], (see [26] for a detailed discussion of this topic). In this study, we used Catapult-C as the C-based tool, which synthesizes hardware directly from standard C/C++ and allows annotations and user specified settings for greater customization. Such annotations are most effective in those parts of the source code that have static loop bounds and statically determinable data dependencies.

8.1.1 Relationship between C-based design and generated hardware

Given a C-based design, the tools generate hardware for completing the computation, maintaining an input-output correspondence with the C description. As we will discuss later in this chapter, the tools utilize user provided annotations for defining the level of parallelism, pipelining and resource sharing desired, and will optimize for the given area and performance constraints to direct synthesis. From the generated hardware it should be possible to create fine-grained power domains for the design, as each C function could result in several hardware computational blocks not all of which might be simultaneously active.

However, the generated domains would lack a direct relationship with the original high-level specification because the compiler code transformations to generate the hardware FSMs (Finite-State-Machines) are not visible at the source level. In

addition, the timing information about operation scheduling is not present in the sequential C description. Though a high-level C execution can provide information on how many times a given function is invoked under a particular test run, the following information, which is essential for determining the activity metrics of the domains, is unavailable:

1. The number of clock cycles used by the design to run the test bench.
2. The number of clock cycles each block is active for computation triggered by the given input.
3. Relative cycle-by-cycle scheduling of operations in the design which directs exactly when each block becomes active and inactive.

In absence of such a connection, all activity information for the possible domains would have to be collected at the RTL design level. This would lead to the conventional increase in design and verification effort associated with power gating of RTL designs. Any automation techniques used would only be applied for the RTL, bypassing the high-level design and negating benefits of using the latter.

For rule-based designs, the rule-level activity information either directly corresponds to the activity of a computational block exclusively active when a rule fires, or it can be used in a boolean function to compute the net activity of a shared block between multiple rules. In this manner the exact activity information for the power domains under consideration can be determined completely from a high-level simulation. This ensures that the automation techniques can utilize the high-level design information for reducing design and verification costs associated with fine-grained power gating.

There is an additional concern about the hardware quality generated using high-level design, whether raising the design abstractions causes an increase in area or decrease in design performance. Such inefficiencies in the generated hardware design would swamp any power savings obtained by the power gating. In the rest of this chapter, we look closely at such hardware inefficiencies. We give examples where it

is essential to exploit parallelism, the extent of which depends on run-time parameters. It is difficult for the user to restructure some of these source codes to allow the C-based tool to infer the desired hardware structure. These hardware structures can be designed using any HDL; we used a high-level rule-based HDL, Bluespec SystemVerilog [9], which makes it easy to express the necessary architectural elements to achieve the desired performance. To explore and elaborate on these issues, we will use a specific application, Reed-Solomon decoding, as a design example. In particular, we will explore whether the HLS design methodologies can achieve performance targets under given area constraints.

8.2 The Application: Reed-Solomon Decoder

Reed-Solomon codes [53] are a class of error correction codes frequently used in wireless protocols. In this chapter, we present the design of a Reed-Solomon decoder for an 802.16 protocol receiver [38]. The target operating frequency for the FPGA implementation of our designs was set to 100 MHz. To achieve the 802.16 target throughput of 134.4 Mbps at this frequency, the design needs to accept a new 255 byte input block every 1520 cycles. During the design process, our goal was also to see if the number of cycles can be reduced even further because the "extra performance" can be used to decrease voltage or frequency for low power implementations.

8.2.1 Decoding Process

Reed-Solomon decoding algorithm [79] consists of five steps:

1. Syndrome computation by evaluating the received polynomial at various roots of the underlying Galois Field (GF) primitive polynomial

2. Error locator polynomial and error evaluator polynomial computation through the Berlekamp-Massey algorithm using the syndrome

3. Error location computation using Chien search which gives the roots of the error locator polynomial

4. Error magnitude computation using Forney's algorithm

5. Error correction by subtracting the computed errors from the received polynomial

Each input block is decoded independently of other blocks. A Reed-Solomon encoded data block consists of k information symbols and $2t$ parity symbols for a total of n $(= k + 2t)$ symbols. The decoding process is able to correct a maximum of t errors.

8.3 Generating hardware from C/C++

8.3.1 The Initial Design

The decoding algorithm was written in a subset of C++ used by the tool for compiling into hardware. Each stage of the Reed-Solomon decoder was represented by a separate function and a top-level function invokes these functions sequentially. The different functions share data using array pointers passed as arguments. High-level synthesis tools can automatically generate a finite state machine (FSM) associated with each C/C++ function once the target platform (Xilinx Virtex II FPGA) and the target frequency (100 MHz) has been specified. For our Reed-Solomon code, with n as 255 and t as a parameter with a maximum value of 16, the tool generated a hardware design that required 7.565 million cycles per input block, for the worst case error scenario. The high cycle count was due to the fact that the tool produced an FSM for each computation loop that exactly mimicked its sequential execution. We next discuss how we reduced this cycle count by three orders of magnitude.

8.3.2 Loop unrolling to increase parallelism

C-based design tools exploit computational loops to extract fine-grain parallelism [32]. Loop unrolling can increase the amount of parallelism in a computation and data-dependency analysis within and across loops can show the opportunities for pipelined execution. For example, the algorithm for syndrome calculations consists of two

nested for-loops. For a typical value of $t = 16$, the inner-loop computes 32 syndromes sequentially. All of these can be computed in parallel if the inner-loop is unfolded. Most C-based design tools can automatically identify the loops that can be unrolled. By adding annotations to the source code, the user can specify which of these identified loops need to be unrolled and how many times they should be unrolled. For unrolling, we first selected the for-loops corresponding to the Galois Field Multiplication, which is used extensively throughout the design. Next, the inner for-loop of Syndrome computation was unrolled. The inner for-loop of the Chien search was also unrolled. To perform unrolling we had to replace the dynamic parameters being used as loop bounds by their static upper bounds. These unrolling steps cumulatively lead to an improvement of two orders of magnitude in the throughput, achieving 19,020 cycles per input block. Still, this was only 7% of the target data throughput.

8.3.3 Expressing producer-consumer relationships

To further improve the throughput, two consecutive stages in the decoder need to be able to exploit fine-grain producer-consumer parallelism. For example, once the Chien search module determines a particular error location, that location can be forwarded immediately to the Forney's algorithm module for computation of the error magnitude, without waiting for the rest of error locations. Such functions are naturally suited for pipelined implementations. But this idea is hard to express in sequential C source descriptions, and automatic detection of such opportunities is practically impossible.

For simple loop structures, the compiler can infer that both the producer and consumer operate on data symbols in-order. It can use this information to process the data in a fine-grained manner, without waiting for the entire block to be available. Consider the code segment shown in Figure 8-1.

For such simple producer-consumer pairs, the C-based tool appropriately generates streaming hardware in the form shown in Figure 8-2 and passes one byte at a time, as opposed to passing the entire data structure, to achieve maximum overlapped execution of the producer and consumer processes.

```
void producer(char input[255], char intermediate[255])
{
  for (int i=0; i<255; i++)
    intermediate[i]=input[i]+i;
}
void consumer(char intermediate[255], char output[255])
{
  for (int i=0; i<255; i++)
    output[i]=intermediate[i]-i;
}
```

Figure 8-1: Simple streaming example - source code

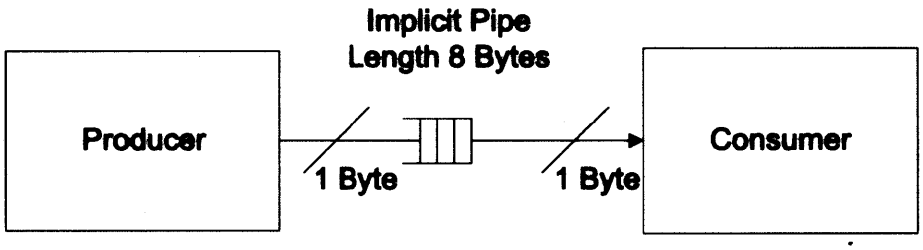


Figure 8-2: Simple streaming example - hardware output

However, the presence of dynamic parameters in for-loop bounds can obfuscate the sharing of streamed data and makes it difficult to apply static dataflow optimizations [48]. For example, consider the code segment shown in Figure 8-3, where the length of the intermediate array produced and the producer loop iterations which produce its values are dynamically determined based on the input.

The hardware generated by the C-based tool for this code is shown in Figure 8-4. The compiler generates a large RAM for sharing one instance of the *intermediate* array between the modules. Furthermore, to ensure the program semantics, the compiler does not permit the two modules to access the array simultaneously, preventing overlapped execution of the two modules. It is conceivable that a clever compiler could detect that the production and consumption of data-elements is in order and then set up a pipelined producer-consumer structure properly. However, we expect such analysis for real codes to be quite difficult and brittle in practice.

Some C-based tools support an alternative buffering mechanism called ping-pong memory which uses a double buffering technique, to allow some overlapping execution,

```
void producer(char input[255], char length,
              char intermediate[255], char *count)
{
  *count = 0;
  for (int i=0; i<length; i++)
    if (input[i]==0)
      intermediate[(*count)++]=input[i]+i;
}
void consumer(char intermediate[255], char *count,
              char output[255])
{
  for (int i=0; i<*count; i++)
    output[i]=intermediate[i]-i;
}
```

Figure 8-3: Complex streaming example - Source code

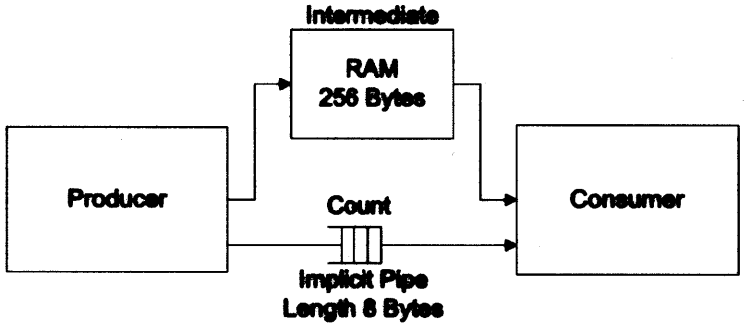


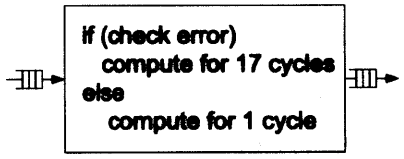
Figure 8-4: Complex streaming example - Hardware output

but at the cost of extra hardware resources. Using this buffer, our design's throughput improved to 16,638 cycles per data block.

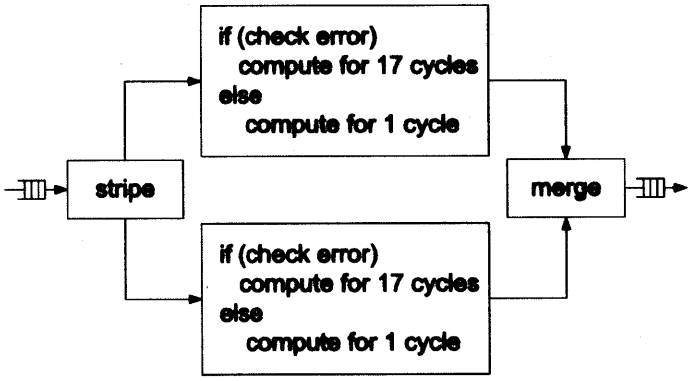
8.3.4 Issues in Streaming Conditionals

The compilers are generally unable to infer streaming architectures if the data blocks are accessed conditionally by the producer or consumer. For example, in Forney's algorithm the operation of the main for-loop is determined by a conditional check whether the location is in error or not. The input data structure contains k (= 223) symbols out of which at most t (= 16) symbols can be in error. Let us further assume it takes 17 cycles to process a symbol in error and only one to process a symbol not in error. The processing of symbols is independent of each other but

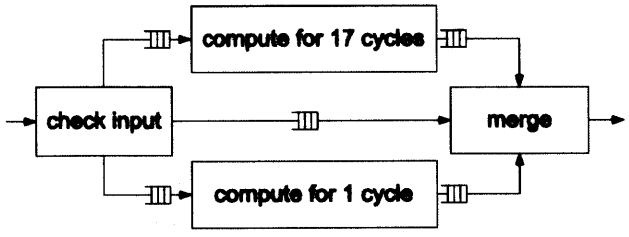
the output stream must maintain the order of the input. If the compiler is unable to detect this independence, it will process these symbols in-order sequentially, taking as much as $17t + (k - t) = k + 16t = 479$ cycles (see Figure 8-5(a)). On the other hand, if the compiler can detect the independence of conditional loop iterations and we ask the tool to unroll it 2 times, we get the structure shown in Figure 8-5(b). If the errors are distributed evenly, such a structure may double the throughput at the cost of doubling the hardware. The preferred structure for this computation is the pipeline shown in Figure 8-5(c), which should take $max(17t, k - t) = 272$ cycles to process all k symbols. Notice Figure 8-5(c) takes considerably less area than Figure 8-5(b) because it does not duplicate the error handling hardware.



(a) Original structure



(b) Unrolled structure



(c) Modified structure

Figure 8-5: Forney's Algorithm Implementation

The C-based tool was not able to generate the structure shown in Figure 8-5(c). We think it will be difficult for any C-based synthesis tool to infer such a conditional structure. First, it is always difficult to detect if the iterations of an outer loop can be done in parallel. Second, static scheduling techniques rely on the fact that different branches take equal amount of time, while we are trying to exploit the imbalance in branches.

To further improve the performance and synthesis results, we made use of common guidelines [72] for code refinement. Adding hierarchy to Berlekamp computations and making its complex loop bounds static by removing the dynamic variables from the loop bounds, required algorithmic modifications to ensure data consistency. By doing so, we could unroll the Berlekamp module to obtain a throughput of 2073 cycles per block. However, as seen in Section 8.5, even this design could only achieve 66.7% of the target throughput and the synthesized hardware required considerably more FPGA resources than the other designs.

8.3.5 Fine-grained processing

Further optimizations require expressing module functions in a fine-grained manner, i.e. operating on a symbol-by-symbol basis. This leads to considerable complexity as modules higher in hierarchy have to keep track of individual symbol accesses within a block. The modular design would need to be flattened completely, so that a global FSM can be made aware of fine-grained parallelism across the design. The abstractions provided by high-level sequential languages are at odds with these types of concurrent hardware structures and make it difficult for algorithm designers to express the intended structures in C/C++. Others have identified the same tension [26]. This is the reason for the inefficiency in generated hardware which we encountered during our study. The transaction granularity on which the functions operate is a trade-off between performance and implementation effort. Coarse-grain interfaces where each function call processes a complete array is easier for software programmers but fine-grain interface gives the C compiler a better chance to exploit fine-grained parallelism.

8.4 Implementation in rule-based design languages

We next compare the implementation using a rule-based design language, Bluespec SystemVerilog. Bluespec encourages the designer to consider the decoding algorithm in terms of concurrently operating modules, each corresponding to one major functional block. Modules communicate with each other through bounded First-In-First-Out (FIFO) buffers as shown in Figure 8-6. Each module's interface simply consists of methods to enqueue and dequeue data with underlying Bluespec semantics taking care of control logic for handling full and empty FIFOs. It is straightforward to encode desired architectural mechanisms and perform design exploration to search for an optimal hardware configuration. Bluespec supports polymorphism, which allows expression of parameterized module interfaces to vary granularity of data communication between modules. The pipeline in Figure 5-2 is latency insensitive in the sense that its functional correctness does not depend upon the size of FIFOs or the number of cycles each module takes to produce an output or consume an input. This provides great flexibility in tuning any module for better performance without affecting the correctness of the whole pipeline.

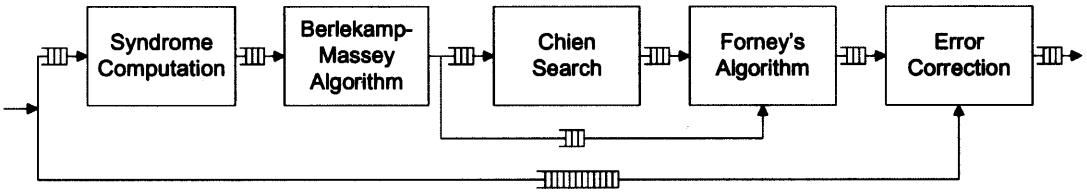


Figure 8-6: Bluespec interface for the decoder

8.4.1 Initial design

In Bluespec design, one instantiates the state elements, e.g., registers, memories, and FIFOs, and describes the behavior using atomic rules which specify how the values of the state elements can be changed every cycle. The FSM, with its Muxes and control signals, is generated automatically by the compiler. For example, for Syndrome computation, the input and output of the module are buffered by two FIFOs, *r_in_q* and *s_out_q* and it has three registers: *syn* for storing the temporary value of

the syndrome, and i and j for loop bookkeeping. The entire FSM is represented by a single rule called *compute_syndrome* in the module as shown in Figure 8-7. This rule models the semantics of the two nested for-loops in the algorithm. The GF arithmetic operations, *gf_mult* and *gf_add*, are purely combinational library functions.

```
rule compute_syndrome (True);
  let new_syn = syn;
  let product = gf_mult(new_syn[j],alpha(j+1));
  new_syn[j]  = gf_add(r_in_q.first(), product);
  if (j + 1 >= 2*t)
    j <= 0; r_in_q.deq();
    if (i + 1 == n)
      s_out_q.enq(new_syn);
      syn <= replicate(0);
      i <= 0;
    else
      i <= i + 1;
  else
    syn <= new_syn;
    j <= j + 1;
endrule
```

Figure 8-7: Initial version of compute-syndrome rule

We implemented each of the five modules using this approach. This initial design had a throughput of 8,161 cycles per data block. This was 17% of the target data throughput. It should be noted that even in this early implementation, computations in different modules can occur concurrently on different bytes of a single data block boosting the performance.

8.4.2 Design Refinements

Bluespec requires users to express explicitly the level of parallelism they want to achieve, which can be parameterized similar to the degree of loop unrolling in C-based tools. We illustrate this using the Syndrome Computation module. This module requires $2t$ GF Mults and $2t$ GF Adds per input symbol, which can be performed in parallel. The implementation shown in Figure 8-8 completes **par** iterations per cycle.

We unrolled the computations of the other modules using this technique, which allowed the design to process a block every 483 cycles. At this point, the design

```
rule compute_syndrome (True);
  let new_syn = syn;
  for (Byte p = 0; p < par; p = p + 1)
    let product = gf_mult(in_q.first,alpha(i+p+1));
    new_syn[i+p] = gf_add(new_syn[i+p], product);
  if (j + par >= 2*t)
    j <= 0; in_q.deq();
    if (i + 1 == n)
      out_q.enq(new_syn); syn <= replicate(0); i <= 0;
    else
      i <= i + 1;
  else
    syn <= new_syn; j <= j + par;
endrule
```

Figure 8-8: Parameterized parallel version of the compute-syndrome rule

throughput was already 315% of the target performance. It was possible to boost the performance even further by using some of the insight into algorithmic structures discussed in Section 8.3. For example, at this point in the design cycle we found that the Forney's algorithm module was the bottleneck, which could be resolved by using a split conditional streaming structure shown in Figure 8-5(c). This structure can be described in BSV using individual rules triggering independently for each of the steps shown as a box in Figure 8-5(c). This design allowed the Forney's Algorithm module to process an input block every 272 cycles. The sizes of FIFO buffers in the system also have a large impact on the overall system throughput and area. It is trivial to adjust the sizes of the FIFOs with the BSV library. Exploration of various sizes through testbench simulations allowed fine-tuning of the overall system to get a system throughput of 276 cycles per input block, which was 5.5x of the target throughput, as seen in Section 8.5.

8.5 Area and performance results

At the end of the design process, the RTL outputs of C and Bluespec design flows were used to obtain performance and hardware synthesis metrics for comparison. Both the RTL designs were synthesized for Xilinx Virtex-II Pro FPGA using Xilinx ISE v8.2.03i. The Xilinx IP core for Reed Solomon decoder, v5.1 [80], was used for

comparison. The designs were simulated to obtain performance metrics. Table 8.1 summarizes the results[1]. The C-based design achieved only 23% of the Xilinx IP's data rate while using 201% of the latter's equivalent gate count, while the Bluespec design achieved 178% of the IP's data rate with 90% of its equivalent gate count.

Table 8.1: Comparison of Source code size, FPGA resources and performance

Design	**C-based tool**	**Bluespec**	**Xilinx**
Lines of Source Code	1046	1759	3756*
LUTs	29549	5863	2067
FFs	8324	3162	1386
Block RAMs	5	3	4
Equivalent Gate Count	596,730	267,741	297,409
Frequency (MHz)	91.2	108.5	145.3
Throughput (Cyc/Blk)	2073	276	660
Data rate (Mbps)	89.7	701.3	392.8

Using the Reed-Solomon decoder as an example, we have shown that even for DSP algorithms with relatively simple modular structure, architectural issues dominate in determining the quality of hardware generated. Identifying the right microarchitecture requires exploring the design space, *i.e.*, a design needs to be tuned after we have the first working design. Examples of design explorations include pipelining at the right level of granularity, splitting streaming conditionals to exploit computationally unbalanced branches, sizing of buffers and caches, and associated caching policies. The desired hardware structures can always be expressed in an HDL like Verilog, but it takes considerable effort to do design exploration. HDLs like Bluespec bring many advantages of software languages in the hardware domain by providing high-level language abstractions for handling intricate controls and allowing design exploration through parametrization.

C-based design flow offers many advantages for algorithmic designs — the designer works in a familiar language and often starts with an executable specification. The C-based synthesis tools can synthesize good hardware when the source code is analyzable for parallelism and resource demands. The compiler's ability to infer appropriate

[1]* we use a publicly available comparable Verilog RTL [76] for source code size comparison as the source code was not available for the Xilinx IP

dataflow and parallelism, and the granularity of communication, decreases as the data-dependent control behavior in the program increases. It is difficult for the user to remove all such dynamic control parameters from the algorithm, as seen in the case of Forney's algorithm, and this leads to inefficient hardware. For our case study, we were not able to go beyond 66.7% of the target performance with the C-based tool. It is not clear to us if even a complete reworking of the algorithm would have yielded the target performance.

Thus, it is clear that rule-based design languages like Bluespec can achieve desired performance and area targets as well as hand-coded Verilog RTL. The cost of high-level design for such implementations is minimal, while the benefits of HLS such as concise code and higher abstraction increase designer productivity. In the next section, we explore possibilities in applying various power gating techniques in C-based design, in particular partitioning of designs into distinct activity regions and quantifying the activity metrics of such designs.

8.6 Application of power gating techniques in C-based design

In this section, we briefly explore the possibilities of applying the earlier mentioned power gating techniques to C-based hardware design. In this chapter, we have shown the inefficiencies associated with such a design methodology. However having chosen a C-based design, the application of our techniques can save leakage power in this methodology as well. Our technique fundamentally relies on the knowledge of scheduling of various logic blocks in the design, and pre-determined control signals related to the high-level design. The C-based hardware compiler does have this information and so it would be possible to insert power gating for logical blocks corresponding to various computational functions in the C-based design description.

One point to note here is that our rule-based power gating scheme ensures that there is no loss in performance by gating of the design. This requires that no logic

block which was scheduled to be used in a given cycle of operation in the ungated design, can be switched off in the gated design. Thus, an equivalent implementation of power gating in the C-based design methodology also needs to ensure that the scheduling of various functional blocks is maintained as in the ungated design.

In such designs, there could be a need for additional logic required for generating the control signals of the power switches. The hardware generated from the C-based description need not have such scheduling signals already present in the form that would correspond to the fine-grained gating blocks generated for each implemented function or method. In view of compiler optimizations for area efficiency or performance improvement such as loop merging or loop unrolling, it could lead to changes in the activity factors seen for the generated power domains. Our technique for use of dynamic activity metrics for determining viable domains can only be applicable for C-based design on clear availability of the relationship between high-level specification and the domains under consideration. It is possible that this information can be provided by the C-based synthesis tool. By generating breakeven thresholds for such domains, we can similarly select viable domains for net power savings.

In this manner, the power gating techniques introduced in our work could be applied to hardware design from C-based specifications as an extension of the techniques for rule-based design languages. A complete implementation would require building a C-to-hardware compiler to gain access to the compiler's operation scheduling, which is beyond the scope of this work.

Chapter 9

Conclusions

This chapter presents the conclusions for the thesis. In Section 9.1, we provide a summary of the work presented and the main contributions of the thesis. We conclude by proposing future extensions and research directions for the discussed techniques in Section 9.2, allowing them to have greater applicability in a wide variety of hardware designs.

9.1 Thesis Summary

With technology scaling, the fraction of leakage power in the total power consumption of ASIC designs is increasing. Static power management through power gating is an essential part of the hardware design process. This thesis presents techniques that allow granularity of gating to happen at a sub-modular block-level which can provide more opportunities to switch-off inactive logic. Design and verification effort associated with power gating is an important impediment to wide application in heterogeneous hardware designs. We use rule-based design information to automate this process.

We have proposed techniques to automate: 1) the partitioning of a high-level design into independent fine-grained power domains with associated control signals, and 2) the collection of dynamic activity information for selecting viable domains from the point of view of leakage savings and switching energy costs. The control signals used

for gating are identified from pre-existing scheduling logic in the design and do not require insertion of a centralized power management unit. For determining viability, we use two technology-independent statistical activity metrics: *total inactivity* and *frequency of inactive-active transitions.* These activity metrics are collected automatically during the normal test phase of the design flow by insertion of profiling logic into the high-level design. For each rule under consideration, average length of its inactivity intervals is computed and compared with the appropriate breakeven thresholds. The power-gating control signals are correct-by-construction and do not involve any hardware overhead because they are generated using high-level information already present in rule-based designs.

We demonstrated the use of these activity metrics using hardware design examples of protocol-compliant wireless decoders - Reed-Solomon and Viterbi, million-point sparse Fourier transform accelerator and RISC processor. The designs were simulated in multiple testbench environments to illustrate how such information is collected, analyzed and used to identify the viable power domains for gating. We have shown how the use of our technique can reduce the leakage power consumption of inactive datapath logic by up to 90% and the total power consumption of wireless designs by up to 14%. An important aspect of our technique is that it can be used in conjunction with a global power management scheme, accruing additional power savings. Even when a global power domain is turned on, the use of our technique allows switching off blocks within this domain automatically based on their activity condition. Our technique dramatically decreases the verification effort of fine-grained power gating of heterogeneous hardware designs by creating domains that are correct by construction.

In addition to the rule-based methodology, this work provides analysis of how such gating techniques could be applied in low-level RTL designs or high-level C-based designs as well. We described some of the difficulties in identifying the appropriate control signals in RTL designs, and issues with efficient hardware specification in C-based synthesis. We provided reasoning for why rule-based designs are helpful in preserving high-level design information that is essential for automating fine-grained power gating. We next summarize the steps required for incorporating the techniques

introduced in this work into conventional design flow.

Incorporating techniques into design flow

Specific design and activity information is required to allow tools to identify relevant control signals, and to partition designs into domains that can be switched off using the control signals. Such information can be easily obtained using high-level design. Hardware designers can either use highly structured and modular RTL coding styles or use high-level rule-based design which produces such RTL automatically. Given such a design description, our partitioning technique can produce power specifications that identify prospective fine-grained power domains in the design. Our profiling technique automatically adds activity collection logic to the high-level design for generating activity metrics through testbenches with expected input data. The designer uses these metrics to select domains expected to save net energy accounting for domain switching costs. The selection of domains is a static post-synthesis decision, based on expected dynamic activity and breakeven thresholds. Designers can also remove power gating from critical regions of the design if the margin for increased clock cycle time is overshot. Finally, the generated customized power specification is provided to commercial tools to insert the gating into the design layout. This is followed by customary post-layout validation for signal integrity. The main benefit provided by the techniques is removing the burden of design effort in creating the domains and associated control signals, as well as reducing the verification effort of collecting relevant activity information to ensure that the gated domains save net energy.

9.2 Future extensions

We have presented an initial framework for automating fine-grained power gating. Using this work as a foundation, several extensions are possible through which the power gating issues could be solved under different scenarios and requirements. We next describe a few of the possible extensions.

9.2.1 Automatic gating of local state

The technique for determining appropriate power domains by collecting dynamic activity information applies to logic as well as state elements. There is a significant fraction of leakage power consumed by state elements like registers. However, the loss of information on switching-off state elements, and the difficulty of detecting and specifying exactly when it is functionally correct to do so, is a barrier to the application of this technique. Verifying that all future readers of a state have obtained the required information and the value stored in the state element will not be needed by any block internal or external to the module is a complex problem. However, it could be possible to turn off some local state elements inside a module. For example, consider a divider implementation that takes multiple cycles to complete an operation. Any state that is used in this implementation to store running accumulator values or pipeline states, is strictly local to the divide operation and can be turned off when the operation is completed.

One way to implement this is to use multi-cycle rule expression, an enhancement to BSV [43], which generates state elements that are only needed when the operation is triggered. By using the signals indicating active rule computation, we can keep the local state active only when needed. Another way could be by compiler analysis of state within a module to determine what state is localized for operations and will not be read externally. This analysis has to conservatively determine the intervals between the last access of a state element and the next update, which can be used as the inactivity interval for gating.

9.2.2 Allowing turn on over multiple cycles

Currently, our technique requires the power domain to turn on within a clock cycle, thus limiting the maximum clock frequency of the design. This requirement arises due to the control signals being updated every cycle to indicate which rules are scheduled for execution. The way to overcome the clock speed limitation is to allow domains to turn on over multiple cycles.

One way to implement this while preserving original functionality requires change in rule scheduling to stall rules that read state updates from a rule being activated over multiple cycles. Once the logic has been triggered to be activated, the state update would be registered only after a defined number of cycles and then the stalled rules can be scheduled. Another method is to conservatively generate look ahead turn-on signals for rules that might get activated in a few cycles. Though a conservative turn-on technique would reduce the leakage savings, there can still be significant reduction in the power consumption of designs containing rules with long inactivity intervals. Development of a modified compilation scheme that generates scheduling logic to account for multi-cycle activation of rules is one of the future extensions of our work.

9.2.3 Extending to module level gating

Power gating designs at the module-level coarse granularity has some advantages from the viewpoint of post-synthesis implementation. Under such a scheme, BSV modules would be synthesized into separate Verilog modules and can be expressed as independent power domains for the layout tools. To automatically generate distributed gating signals for such domains, in the absence of a centralized controller, we require information about each module's logic activity as well as state usage. We identify the following conditions for turning on and off a power domain that consists of a single module with *methods* for interfacing with the external environment. To turn the power domain on from an off state, any one of the IO method *enable* signals should be high. To turn the power domain off from an on state, all of the following need to be true: all method *enable* signals should be low, all rule *willFire* signals should be low and none of the state elements would be read by any external module. Figure 9-1 shows how such module-level gating condition would depend on internal rule firing signals as well as on signals indicating pending input-output requests, for a module containing no externally read state.

In the general case of module-level gating, we have to ensure that there is no loss of internal state that is yet to be read by an external method. One naive way of verifying this would be to check the *ready* signals of output methods. For example,

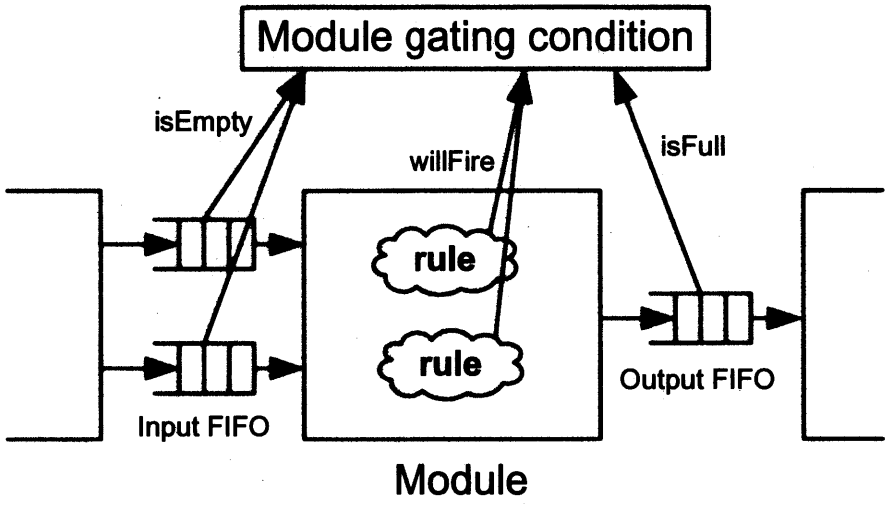


Figure 9-1: Generating module-level gating condition for modules containing no externally read state

if a module has a FIFO buffer in which the output is stored for an external module to read from, after the computation is completed all rules can stop firing while the output methods become ready. However, this is not universal, and there can be designs where module registers need to be read externally and the methods to read such state might be always ready. We propose the use of a *state done* signal that can be combined optionally with one or more methods or it can be an independent signal designed by the user. Here, the burden of ensuring that there is no pending state that can be lost by power gating lies with the designer, making the correctness issue moot. In this scenario, some of the gating logic, namely the rule *willFire* signal generation, is contained within the power domain being gated. However, this logic is only needed for turning off the domain and hence it will be valid when used.

Another possibility is to consider additional primitive modules, such as FIFO buffers themselves, being available for power gating at the modular level. Such BSV primitive modules would have additional control signals that can provide information required for generating the gating condition. For some FIFO buffers, BSV provides a *not-Empty* signal indicating presence of at least one enqueued data element. If this

signal is false, it is clear that the buffer module can be power gated without loss of data. Use of such state-holding modules which provide information about the state through externally accessible methods, can allow for gating at a modular granularity.

9.2.4 Integrated power management

This work can lead to an integrated power management which tackles both dynamic and static power dissipation in an automated manner. The analysis that we perform to identify inactive computational blocks can also be extended to identify state elements that are not updated in a given cycle. This information can be used for clock gating of such state elements for reducing dynamic power. For state elements which are updated in a given cycle and do get clocked, we still can save static power by power gating of blocks that do not contribute to the new state computation.

A further improvement can be addition of dynamic decision making to the distributed power controller, for selecting whether to turn off the power domains depending on the expected length of inactivity intervals. For digital signal processing blocks used in wireless applications, frequently the bit error rates in the input data can be determined early in the wireless pipeline. Since the activity rates and interval lengths of computational blocks depends on these error rates, under certain high BER scenarios the sleep signals of power domains can be disabled dynamically to avoid paying the activation energy costs and increasing the net savings achieved. Similar techniques can be applicable in micro-processors, where instruction speculation can help in dynamically deciding whether functional units like floating point arithmetic blocks can be turned off. The combination of such features for reducing static and dynamic power dissipation can provide an integrated solution for power management of digital designs.

In this work, we have provided methods for applying distributed power management to combat increasing leakage power in hardware designs. With the proliferation of wireless, hand-held electronic devices in the current era, such techniques are an essential component for designers working with strict power budgets. Though our

focus was towards automating fine-grained gating of the inactive combinational logic, the ideas introduced in this work are applicable in a variety of power management contexts. By the implementation of the discussed extensions, automatic steps can be used in the design flow to get close to the ideal scenario of zero power dissipation for zero work.

Bibliography

[1] Abhinav Agarwal and Arvind. Leveraging rule-based designs for automatic power domain partitioning. In *Computer-Aided Design (ICCAD), 2013 IEEE/ACM International Conference on*, pages 326–333, Nov 2013.

[2] Abhinav Agarwal, Nirav Dave, Kermin Fleming, Asif Khan, Myron King, Man Cheuk Ng, and Muralidaran Vijayaraghavan. Implementing a fast cartesian-polar matrix interpolator. In *Formal Methods and Models for Co-Design, 2009. MEMOCODE '09. 7th IEEE/ACM International Conference on*, pages 73–76, July 2009.

[3] Abhinav Agarwal, Haitham Hassanieh, Omid Abari, Ezz Hamed, Dina Katabi, and Arvind. High-throughput implementation of a million-point sparse fourier transform. In *Field Programmable Logic and Applications (FPL), 2014 24th International Conference on*, Sept 2014.

[4] Abhinav Agarwal, Man Cheuk Ng, and Arvind. A Comparative Evaluation of High-Level Hardware Synthesis Using Reed-Solomon Decoder. *Embedded Systems Letters, IEEE*, 2(3):72 –76, September 2010.

[5] Arvind, Rishiyur S. Nikhil, Daniel L. Rosenband, and Nirav Dave. High-level Synthesis: An Essential Ingredient for Designing Complex ASICs. In *Proceedings of the 2004 IEEE/ACM International Conference on Computer-Aided Design*, ICCAD '04, San Jose, CA, 2004.

[6] Joshua Auerbach, David F. Bacon, Perry Cheng, and Rodric Rabbah. Lime: a Java-compatible and synthesizable language for heterogeneous architectures. In

Proceedings of the ACM international conference on Object oriented programming systems languages and applications, OOPSLA '10, pages 89–108, New York, NY, USA, 2010. ACM.

[7] Jonathan Bachrach, Huy Vo, Brian Richards, Yunsup Lee, Andrew Waterman, Rimas Avižienis, John Wawrzynek, and Krste Asanović. Chisel: Constructing Hardware in a Scala Embedded Language. In *Proceedings of the 49th Annual Design Automation Conference*, DAC '12, pages 1216–1225, New York, NY, USA, 2012. ACM.

[8] Nilanjan Banerjee, Arijit Raychowdhury, Swarup Bhunia, Hamid Mahmoodi Meimand, and Kaushik Roy. Novel low-overhead operand isolation techniques for low-power datapath synthesis. In *Computer Design: VLSI in Computers and Processors, 2005. ICCD 2005. Proceedings. 2005 IEEE International Conference on*, pages 206–211, Oct 2005.

[9] Bluespec Inc. Bluespec SystemVerilog Language. `www.bluespec.com`.

[10] Bluespec, Inc., Waltham, MA. *Bluespec SystemVerilog Reference Guide*, November 2009.

[11] L. Bolzani, A Calimera, A Macii, E. Macii, and M. Poncino. Enabling concurrent clock and power gating in an industrial design flow. In *Design, Automation Test in Europe Conference Exhibition, 2009. DATE '09.*, pages 334–339, April 2009.

[12] Jens Brandt, Klaus Schneider, Sumit Ahuja, and Sandeep K. Shukla. The model checking view to clock gating and operand isolation. In *Application of Concurrency to System Design (ACSD), 2010 10th International Conference on*, pages 181–190, June 2010.

[13] Cadence. C-to-Silicon Compiler. `www.cadence.com`.

[14] Benton H. Calhoun, Frank A. Honore, and Anantha Chandrakasan. Design methodology for fine-grained leakage control in mtcmos. In *Proceedings of the*

2003 international symposium on Low power electronics and design, ISLPED '03, pages 104–109, New York, NY, USA, 2003. ACM.

[15] Calypto Design Systems. Catapult C. `www.calypto.com`.

[16] Andrew Canis, Jongsok Choi, Mark Aldham, Victor Zhang, Ahmed Kammoona, Jason H Anderson, Stephen Brown, and Tomasz Czajkowski. LegUp: high-level synthesis for FPGA-based processor/accelerator systems. In *Proceedings of the 19th International Symposium on Field Programmable Gate Arrays (FPGA)*. ACM, 2011.

[17] Bhaskar Chatterjee, Manoj Sachdev, Steven Hsu, Ram Krishnamurthy, and Shekhar Borkar. Effectiveness and scaling trends of leakage control techniques for sub-130nm cmos technologies. In *ISLPED '03: Proceedings of the 2003 international symposium on Low power electronics and design*, pages 122–127, New York, NY, USA, 2003. ACM.

[18] Anupam Chattopadhyay, B. Geukes, David Kammler, Ernst Martin Witte, Oliver Schliebusch, Harold Ishebabi, Rainer Leupers, Gerd Ascheid, and Heinrich Meyr. Automatic ADL-based Operand Isolation for Embedded Processors. In *Design, Automation and Test in Europe, 2006. DATE '06. Proceedings*, volume 1, pages 1–6, March 2006.

[19] Deming Chen, Jason Cong, Yiping Fan, Guoling Han, Wei Jiang, and Zhiru Zhang. xPilot: A Platform-Based Behavioral Synthesis System. *SRC TechCon*, 5, 2005.

[20] Mu-Yen Chen, Da-Ren Chen, and Shu-Ming Hsieh. A Blocking-Aware Scheduling for Real-Time Task Synchronization Using a Leakage-Controlled Method. *International Journal of Distributed Sensor Networks (IJDSN)*, 2014.

[21] David Chinnery, Kurt Keutzer, Jerry Frenkil, and Srini Venkatraman. Power gating design automation. In *Closing the Power Gap Between ASIC and Custom: Tools and Techniques for Low Power Design*, pages 251–280. Springer US, 2007.

[22] De-Shiuan Chiou, Da-Cheng Juan, Yu-Ting Chen, and Shih-Chieh Chang. Fine-grained sleep transistor sizing algorithm for leakage power minimization. In *Proceedings of the 44th annual Design Automation Conference*, DAC '07, pages 81–86, New York, NY, USA, 2007. ACM.

[23] Deniz Dal and Nazanin Mansouri. Power optimization with power islands synthesis. *Computer-Aided Design of Integrated Circuits and Systems, IEEE Transactions on*, 28(7):1025–1037, July 2009.

[24] Nirav Dave. Designing a reorder buffer in bluespec. In *Formal Methods and Models for Co-Design, 2004. MEMOCODE '04. Proceedings. Second ACM and IEEE International Conference on*, pages 93–102, June 2004.

[25] Nirav Dave, Michael Pellauer, Steve Gerding, and Arvind. 802.11a transmitter: a case study in microarchitectural exploration. In *Formal Methods and Models for Co-Design, 2006. MEMOCODE '06. Proceedings. Fourth ACM and IEEE International Conference on*, pages 59–68, July 2006.

[26] S. A. Edwards. Challenges of Synthesizing Hardware from C-Like Languages. *IEEE Design and Test of Computers*, 23(5), May 2006.

[27] Thomas Esposito, Mieszko Lis, Ravi Nanavati, Joseph Stoy, and Jacob Schwartz. System and method for scheduling TRS rules. United States Patent US 133051-0001, February 2005.

[28] Kermin Fleming, Chun-Chieh Lin, Nirav Dave, Arvind, Gopal Raghavan, and Jamey Hicks. H.264 Decoder: A Case Study in Multiple Design Points. In *MEMOCODE*, pages 165–174, 2008.

[29] D.D. Gajski, J. Zhu, R. Domer, A. Gerstlauer, and S. Zhao. *SpecC: Specification Language and Methodology.* Kluwer, 2000.

[30] Steve Gunther, Anant Deval, Ted Burton, and Rajesh Kumar. Energy-efficient computing: Power management system on the Nehalem family of processors. *Intel Technology Journal*, 14(3):50, September 2010.

[31] Y. Guo, D. McCain, J. R. Cavallaro, and A. Takach. Rapid Prototyping and SoC Design of 3G/4G Wireless Systems Using an HLS Methodology. *EURASIP Journal on Embedded Systems*, 2006(1):18–18, 2006.

[32] Sumit Gupta, Nikhil Dutt, Rajesh Gupta, and Alexander Nicolau. Loop Shifting and Compaction for the High-Level Synthesis of Designs with Complex Control Flow. In *Proceedings of DATE*, 2004.

[33] Haitham Hassanieh, Piotr Indyk, Dina Katabi, and Eric Price. SFFT Sparse Fast Fourier Transform [online]. `http://groups.csail.mit.edu/netmit/sFFT/code.html`.

[34] Haitham Hassanieh, Piotr Indyk, Dina Katabi, and Eric Price. Simple and practical algorithm for sparse fourier transform. In *Proceedings of the 23rd ACM-SIAM Symposium on Discrete Algorithms, SODA*, pages 1183–1194, 2012.

[35] Shousheng He and Mats Torkelson. A New Approach to Pipeline FFT Processor. In *Proceedings of the 10th International Parallel Processing Symposium*, IPPS '96, pages 766–770, Washington, DC, USA, 1996. IEEE Computer Society.

[36] James C. Hoe and Arvind. Synthesis of operation-centric hardware descriptions. In *Computer Aided Design, 2000. ICCAD-2000. IEEE/ACM International Conference on*, pages 511–518, Nov 2000.

[37] Nathan J. Ickes. *A Microppower DSP for Sensor Applications.* PhD thesis, MIT, Cambridge, MA, 2008.

[38] IEEE. *IEEE standard 802.16. Air Interface for Fixed Broadband Wireless Access Systems*, 2004.

[39] A. Ioannou and M.G.H. Katevenis. Pipelined heap (priority queue) management for advanced scheduling in high-speed networks. *Networking, IEEE/ACM Transactions on*, 15(2):450–461, 2007.

[40] Nikhil Jayakumar, Suganth Paul, Rajesh Garg, Kanupriya Gulati, and SunilP. Khatri. Optimum reverse body biasing for leakage minimization. In *Minimizing and Exploiting Leakage in VLSI Design*, pages 91–100. Springer US, 2010.

[41] Takashi Kambe, Akihisa Yamada, Koichi Nishida, Kazuhisa Okada, Mitsuhisa Ohnishi, Andrew Kay, Paul Boca, Vince Zammit, and Toshio Nomura. A C-based synthesis system, Bach, and its application. In *ASP-DAC*, pages 151–155. ACM, 2001.

[42] J. Kao, S. Narendra, and A Chandrakasan. Mtcmos hierarchical sizing based on mutual exclusive discharge patterns. In *Design Automation Conference, 1998. Proceedings*, pages 495–500, June 1998.

[43] Michal Karczmarek and Arvind. Synthesis from multi-cycle atomic actions as a solution to the timing closure problem. In *Proceedings of the 2008 IEEE/ACM International Conference on Computer-Aided Design*, ICCAD '08, pages 24–31, Piscataway, NJ, USA, 2008. IEEE Press.

[44] Michael Keating, David Flynn, Rob Aitken, Alan Gibbons, and Kaijian Shi. *Low Power Methodology Manual: For System-on-Chip Design.* Springer Publishing Company, Incorporated, 2007.

[45] Asif Khan, Muralidaran Vijayaraghavan, Silas Boyd-Wickizer, and Arvind. Fast and cycle-accurate modeling of a multicore processor. In *Proceedings of the 2012 IEEE International Symposium on Performance Analysis of Systems & Software, New Brunswick, NJ, USA, April 1-3, 2012*, pages 178–187, 2012.

[46] Nam Sung Kim, Todd Austin, David Blaauw, Trevor Mudge, Krisztián Flautner, Jie S. Hu, Mary Jane Irwin, Mahmut Kandemir, and Vijaykrishnan Narayanan. Leakage current: Moore's law meets static power. *Computer*, 36(12):68–75, December 2003.

[47] R. Kumar and G. Hinton. A family of 45nm IA processors. In *Solid-State Circuits Conference - Digest of Technical Papers, 2009. ISSCC 2009. IEEE International*, pages 58–59, Feb 2009.

[48] Edward A. Lee and David G. Messerschmitt. Static Scheduling of Synchronous Data Flow Programs for Digital Signal Processing. *IEEE Transactions on Computers*, 36(1):24–35, 1987.

[49] Lawrence Leinweber and Swarup Bhunia. Fine-grained supply gating through hypergraph partitioning and shannon decomposition for active power reduction. In *Proceedings of the conference on Design, automation and test in Europe*, DATE '08, pages 373–378, New York, NY, USA, 2008. ACM.

[50] H. Matsutani, M. Koibuchi, D. Ikebuchi, K. Usami, H. Nakamura, and H. Amano. Ultra fine-grained run-time power gating of on-chip routers for cmps. In *Networks-on-Chip (NOCS), 2010 Fourth ACM/IEEE International Symposium on*, pages 61–68, May 2010.

[51] Mentor Graphics. Catapult-C. `www.mentor.com`.

[52] Michael Meredith. High-Level SystemC Synthesis with Forte's Cynthesizer. In *High-Level Synthesis – From Algorithm to Digital Circuit* , pages 75–97. Springer Netherlands, 2008.

[53] Todd K. Moon. *Error Correction Coding-mathematical methods and Algorithms.* Wiley-Interscience, New York, 2005.

[54] Siva G. Narendra and Anantha Chandrakasan. *Leakage in Nanometer CMOS Technologies (Series on Integrated Circuits and Systems).* Springer-Verlag New York, Inc., Secaucus, NJ, USA, 2005.

[55] Man Cheuk Ng, Kermin Elliott Fleming, Mythili Vutukuru, Samuel Gross, Arvind, and Hari Balakrishnan. Airblue: A System for Cross-layer Wireless Protocol Development. In *Proceedings of the 6th ACM/IEEE Symposium on*

Architectures for Networking and Communications Systems, ANCS '10, pages 4:1–4:11, New York, NY, USA, 2010. ACM.

[56] Man Cheuk Ng, Muralidaran Vijayaraghavan, Gopal Raghavan, Nirav Dave, Jamey Hicks, and Arvind. From WiFI to WiMAX: Techniques for IP Reuse Across Different OFDM Protocols. In *Proceedings of MEMOCODE'07*, Nice, France, 2007.

[57] Open SystemC Initiative. SystemC language. `www.systemc.org`.

[58] Navid Paydavosi, Tanvir Hasan Morshed, Darsen D. Lu, Wenwei (Morgan) Yang, Mohan V. Dunga, Xuemei (Jane) Xi, Jin He, Weidong Liu, Kanyu, M. Cao, Xiaodong Jin, Jeff J. Ou, Mansun Chan, Ali M. Niknejad, and Chenming Hu. Bsim 4v4.8.0 mosfet model - user's manual. Technical report, Department of Electrical Engineering and Computer Sciences, University of California, Berkeley, 2013.

[59] S. Potluri, N. Chandrachoodan, and V. Kamakoti. Post-synthesis circuit techniques for runtime leakage reduction. In *VLSI (ISVLSI), 2011 IEEE Computer Society Annual Symposium on*, pages 319–320, July 2011.

[60] Power Forward. *A Practical guide to Low-Power design*, 2009.

[61] Sven Rosinger. *RT-Level Power-Gating Models optimizing Dynamic Leakage-Management.* PhD thesis, Carl von Ossietzky Universitt Oldenburg, Oldenburg, Germany, 2012.

[62] Sven Rosinger, Domenik Helms, and Wolfgang Nebel. Rtl power modeling and estimation of sleep transistor based power gating. *J. Embedded Comput.*, 3(3):189–196, August 2009.

[63] Efraim Rotem, Alon Naveh, Avinash Ananthakrishnan, Doron Rajwan, and Eliezer Weissmann. Power-management architecture of the intel microarchitecture code-named sandy bridge. *IEEE Micro*, 32(2):20–27, 2012.

[64] Toshiyuki Saito. NEC electronics: Integrating power awareness in SoC design with CPF. In *A Practical guide to Low-Power design.* Power Forward, 2009.

[65] Jorn Schumacher. High performance Sparse Fast Fourier Transform. Master's thesis, ETH, Zurich, Switzerland, May 2013.

[66] Greg Semeraro, Grigorios Magklis, Rajeev Balasubramonian, David H. Albonesi, Sandhya Dwarkadas, and Michael L. Scott. Energy-efficient processor design using multiple clock domains with dynamic voltage and frequency scaling. In *Proceedings of the 8th International Symposium on High-Performance Computer Architecture*, HPCA '02, pages 29–, Washington, DC, USA, 2002. IEEE Computer Society.

[67] Eitan N. Shauly. Cmos leakage and power reduction in transistors and circuits: Process and layout considerations. *Journal of Low Power Electronics and Applications*, 2(1):1–29, 2012.

[68] Youngsoo Shin, Jun Seomun, Kyu-Myung Choi, and Takayasu Sakurai. Power gating: Circuits, design methodologies, and best practice for standard-cell VLSI designs. *ACM Trans. Des. Autom. Electron. Syst.*, 15:28:1–28:37, October 2010.

[69] Gaurav Singh and Sandeep K. Shukla. Low-power hardware synthesis from TRS-based specifications. In *Formal Methods and Models for Co-Design, 2006. MEMOCODE '06. Proceedings. Fourth ACM and IEEE International Conference on*, pages 49–58, July 2006.

[70] Gaurav Singh and Sandeep Kumar Shukla. *Low Power Hardware Synthesis from Concurrent Action-Oriented Specifications.* Springer, 2010.

[71] Phillip Stanley-Marbell. What is IEEE P1801 (Unified Power Format)? *SIGDA Newsl.*, 37(19):1:1–1:1, October 2007.

[72] Greg Stitt, Frank Vahid, and Walid Najjar. A code refinement methodology for performance-improved synthesis from c. In *Computer-Aided Design, 2006. ICCAD'06. IEEE/ACM International Conference on*, pages 716–723. IEEE, 2006.

[73] Synfora. PICO Platform. `www.synfora.com`.

[74] Kimiyoshi Usami and Naoaki Ohkubo. A design approach for fine-grained run-time power gating using locally extracted sleep signals. In *24th International Conference on Computer Design*, 2006.

[75] S.R. Vangal, J. Howard, G. Ruhl, S. Dighe, H. Wilson, J. Tschanz, D. Finan, A Singh, T. Jacob, S. Jain, V. Erraguntla, C. Roberts, Y. Hoskote, N. Borkar, and S. Borkar. An 80-tile sub-100-w teraflops processor in 65-nm cmos. *Solid-State Circuits, IEEE Journal of*, 43(1):29–41, Jan 2008.

[76] Varkon Semiconductors. Reed-Solomon Decoder. `www.opencores.org/project,reed_solomon_decoder`.

[77] Villarreal, Jason and Park, Adrian and Najjar, Walid and Halstead, Robert. Designing modular hardware accelerators in C with ROCCC 2.0. In *18th IEEE Annual International Symposium on Field-Programmable Custom Computing Machines (FCCM)*, pages 127–134. IEEE, 2010.

[78] M. Ware, K. Rajamani, M. Floyd, B. Brock, J.C. Rubio, F. Rawson, and J.B. Carter. Architecting for power management: The IBM POWER7 approach. In *High Performance Computer Architecture (HPCA), 2010 IEEE 16th International Symposium on*, pages 1–11, Jan 2010.

[79] Stephen B. Wicker and Vijay Bhargava. *Reed-Solomon Codes and Their Applications*. IEEE Press, New York, 1994.

[80] Xilinx. CORE Reed Solomon Decoder IP v5.1. `www.xilinx.com/ipcenter/coregen/coregen_iplist_71i_ip2.htm`.

[81] Tong Xu, Peng Li, and Boyuan Yan. Decoupling for power gating: Sources of power noise and design strategies. In *Design Automation Conference (DAC), 2011 48th ACM/EDAC/IEEE*, pages 1002–1007. IEEE, 2011.

[82] Jing Yang and Yong-bin Kim. Self adaptive body biasing scheme for leakage power reduction in nanoscale cmos circuit. In *Proceedings of the Great Lakes Symposium on VLSI*, GLSVLSI '12, pages 111–116, New York, NY, USA, 2012. ACM.