System-Level Design of Power Efficient FSMD Architectures

by

Nainesh Agarwal

BEng, University of Victoria, 1998
MASc, University of Waterloo, 2000

A Dissertation Submitted in Partial Fulfillment of the
Requirements for the Degree of

Doctor of Philosophy

in the Electrical and Computer Engineering

University of Victoria

System-Level Design of Power Efficient FSMD Architectures

by

Nainesh Agarwal

BEng, University of Victoria, 1998
MASc, University of Waterloo, 2000

Supervisory Committee

Dr. Nikitas Dimopoulos, Supervisor (Dept. of Elec. and Comp. Engineering)

Dr. Amirali Baniasadi, Departmental Member (Dept. of Elec. and Comp. Engineering)

Dr. Wu-Sheng Lu, Departmental Member (Dept. of Elec. and Comp. Engineering)

Dr. Micaela Serra, Outside Member (Dept. of Computer Science)

Supervisory Committee

Dr. Nikitas Dimopoulos, Supervisor (Dept. of Elec. and Comp. Engineering)

Dr. Amirali Baniasadi, Departmental Member (Dept. of Elec. and Comp. Engineering)

Dr. Wu-Sheng Lu, Departmental Member (Dept. of Elec. and Comp. Engineering)

Dr. Micaela Serra, Outside Member (Dept. of Computer Science)

Abstract

Power dissipation in CMOS circuits is of growing concern as the computational requirements of portable, battery operated devices increases. The ability to easily develop application specific circuits, rather than program general-purpose architectures can provide tremendous power savings. To this end, we present a design platform for rapidly developing power efficient hardware architectures starting at a system level. This high level VLSI design platform, called CoDeL, allows hardware description at the algorithm level, and thus dramatically reduces design time and power dissipation. We compare the CoDeL platform to a modern DSP and find that the CoDeL platform produces designs with somewhat slower run times but dramatically lower power dissipation.

The CoDeL compiler produces an FSMD (Finite State Machine with Datapath) implementation of the circuit. This regular structure can be exploited to further reduce power through various techniques.

To reduce dynamic power dissipation in the resulting architecture, the CoDeL compiler automatically inserts clock gating for registers. Power analysis shows that CoDeL's automated, high-level clock gating provides considerably more power savings than existing automated clock gating tools.

To reduce static power, we use the CoDeL platform to analyze the potential and performance impact of power gating individual registers. We propose a static gating method, with very low area overhead, which uses the information available to the CoDeL compiler to predict, at compile time, when the registers can be powered off and powered on. Static branch prediction is used to more intelligently traverse the finite state machine description of the circuit to discover gating opportunities. Using simulation and estimation, we find that CoDeL with backward branch prediction gives the best overall combination of gating potential and performance. Compared to a dynamic time-based technique, this method gives dramatically more power savings, without any additional performance loss.

Finally, we propose techniques to efficiently partition a FSMD using Integer Linear Programming and a simulated annealing approach. The FSMD is split into two or more simpler communicating processors. These separate processors can then be clock gated or power gated to achieve considerable power savings since only one processor is active at any given time. Implementation and estimation shows that significant power savings can be expected, when the original machine is partitioned into two or more submachines.

Table of Contents

List of Tables

List of Figures

Acknowledgements

It has been a long, crazy journey. I have seen triumphs and disappointments. I have been overjoyed and thoroughly crushed. There are several individuals that have been an integral part of this journey. These are people who have provided the support and encouragement needed to allow me to continue, as best I can, without giving up. It is a pleasure to thank and acknowledge these individuals here.

First and foremost, I would like to thank the enduring guidance, mentorship and friendship of my supervisor, Dr. Nikitas Dimopoulos. His ability to provide the right amount of freedom and impetus is astounding. I am indebted to Dr. Dimopoulos and will forever aspire to imitate his qualities as a fantastic supervisor and a great human being.

For their extremely valuable comments and suggestions, a special thanks to my wonderful committee members: Dr. Wu-Sheng Lu, Dr. Amirali Baniasadi, and Dr. Micaela Serra. Also, my sincere thanks to Dr. Jarmo Takala for agreeing to become the external examiner on such an incredibly short notice, and for taking the time to carefully go through the manuscript and provide such thorough comments.

For their constant readiness to help and provide words of wisdom and encouragement I would like to thank my lab colleagues Farshad Khunjush, Daniel Vanderster, Rafael Parra-Hernandez and Darshika Perera.

Perhaps the most important contribution in my journey has been that of my family. For their undying, selfless love and support I wish to thank my parents. It is their confidence and belief in me that made it possible for me to embark on this journey, and it is the way they gently held my hand throughout that has allowed me to complete it.

From their far-away home in Indianapolis, my little sister, Ruchi, and her husband, Krishan, were always there to hear my stories of joy and complaints of failures. They always had the right things to say. Thank you so much.

From even farther away in India, my parent's-in-law have always held and shown an incredible confidence in my abilities. Thank you for believing in me and allowing me to marry your beloved daughter and take her miles away from you.

Finally, I thank my wife, Sayukta. Mere words can not express all you have done for me. I thank you for giving me two lovely daughters, Sejal and Niyati. My entire journey was possible only because you were beside me as a pillar of strength. I look forward to many splendid journeys in our future!

To my parents,

and

my love, Sayukta

Chapter 1

Introduction

Engineers participate in the activities which make the resources of nature available in a form beneficial to man and provide systems which will perform optimally and economically.

- L. M. K. Boelter

The encroachment of portable computing and communication devices into our lives is apparent today. Most of us employ one or more of these devices on a regular basis to enrich our lives and increase productivity. Examples of such devices includes cellular phones, personal digital assistants, digital cameras, and portable media players. This is just the beginning. New portable devices are being developed which promise increasingly sophisticated capabilities and features. Examples of highly sophisticated applications utilized now and in the future include speech recognition [73, 50] and full frame video encoding and decoding [58, 61].

As the processing algorithms become more complex and the computational requirements increase, the power dissipation rises. This is a major hurdle for portable devices which are battery powered. Batteries are a limited energy source that need to be recharged or replaced once drained. In most laptop computers, for example, batteries last no longer than about four to six hours. Even in highly portable devices such as cellular phones and media players, batteries don't last more than a few

hours. With high intensity applications such as video capture, display and transmission, battery life can be as short as about two hours. Thus, long battery life is a key criterion in the effective design of a portable device. Power dissipation also results in dissipated heat requiring effective cooling and packaging which can be costly. Lowering power and heat dissipation can also result in greater circuit density and longer component lives.

Reduction in power dissipation is an important problem that has received wide attention by several researchers. There are several techniques that have been devised that are available to the low-level circuit designer to use to reduce power and energy in the circuits. These include, dynamic voltage scaling (DVS) [25, 85], dynamic frequency scaling [57], clock gating [18, 54], dual voltage threshold transistors [49], and power gating [68, 41]. These techniques will be highlighted in chapter 3. These techniques, however, rely on manual intervention by circuit designers which are difficult to implement in practice and can lead to long design and verification cycles. Therefore, we focus our attention on techniques that can be fully automated leading to dramatically reduced design times.

Another important evolution occurring these days is the emergence of system-level design languages (SLDLs). In the late 1980s, chip designers started moving away from schematic-capture-based design methodologies to the emerging Hardware Description Languages (HDLs). This moved the designers to a higher level of abstraction, which was necessary to overcome the rapidly increasing hardware complexity. In the early 1990s, the Register Transfer Level (RTL) design era had begun with the introduction of HDLs such as VHDL (VHSIC Hardware Description Language) [43], Verilog [44] and others.

Today a similar evolution is taking place as RTL hardware design is again proving to be too low an abstraction level for designing complex, multi-million gate systems. The answer SLDLs where the entire system, including the software and the hardware,

can be described using a singular language platform. As this happens, VHDL and Verilog will become analogous to assembly level languages. Their explicit use will be limited to performance-critical sections of the system. Some examples of SLDLs include SystemVerilog [77], SystemC [20], HandelC [47], Impulse C [45], Catapult C [60] and CoDeL [75, 1, 3].

Most system level hardware languages, including HandelC, Impulse C and CoDeL, implement the design descriptions as a Finite State Machine with Datapath (FSMD) type of circuit. An FSMD is a hardware system architecture, which comprises of a finite state machine, which controls the flow of program logic, and the datapath, which stores the data elements and performs the desired operations.

1.1 Goals and Contributions

The problem we are trying to solve, in this work, is to reduce power in CMOS circuits in a fully automated manner. To this end, we have developed methods that can be automatically implemented by a system-level design compiler requiring little to no user intervention.

To explore the various low power techniques, we have extended a system-level design language, called CoDeL (Controller Description Language), and enhanced its compiler and development environment [1, 3, 2]. CoDeL allows system description at the algorithmic level through rapid design and implementation of hardware modules without understanding the intricacies of hardware description languages such as VHDL and Verilog. In fact, CoDeL compiles to create synthesizable VHDL code that can be simulated and synthesized using standard VHDL tools. CoDeL implements hardware descriptions as FSMD (Finite State Machine with Datapath) architectures. Thus, it is these classes of structured circuits that we target for power reduction.

CoDeL provides us with an excellent environment to develop power efficient designs. The unique advantage of CoDeL is that it provides us with static behavioral

information of the target design. This information is then analyzed and used to guide the application of power saving methods, such as clock gating and power gating.

To reduce power in CMOS circuits at the micro-architectural level, we have developed and analyzed techniques that can target a circuit's data storage components, called registers, and can be fully automated.

The continuously switching clock is a major source of power dissipation in a synchronous circuit. We have developed a clock gating technique that is implemented at compile time, which turns off the clock signal to registers when they are not needed. This static clock gating mechanism results in dramatic power savings [5, 4, 7, 8, 12]. This gating technique has been fully automated into the CoDeL compiler to reduce the design time of implementing clock gated circuits. We have also developed a power savings estimation framework that is able to quickly and accurately identify the expected power savings from clock gating at compile time.

Another popular approach to reduce power is power gating which turns off the power supply to idle circuit components. We explore a unique power gating approach, which targets individual registers, that can be automatically implemented at compile time [9, 6]. We show that this gating mechanism can provide significant power savings at minimal performance loss.

Macro-architectural power reduction techniques try to isolate groups of circuit components that can then be collectively clock gated or power gated when they are unused. We have developed models that optimally partition a FSMD circuit architecture into two or more communicating subcircuits, which can then be individually clock gated or power gated, to realize substantial power savings [10, 11, 13].

It should be noted that the power reduction techniques we have presented here have been explored through the use of CoDeL. However, these techniques can be implemented in any hardware design platform to provide automated power reduction.

1.2 Overview

In chapters 2 and 3 we provide a background to the topics we have covered in this thesis. We also try to motivate our approaches by presenting outstanding problems and some related solutions. Chapter 2 presents an introduction to system-level design languages and shows where our design platform, CoDeL, fits in this set of existing platforms. In chapter 3 we discuss power dissipation and review various methods to reduce it.

Chapter 4 presents CoDeL, our implementation of a system-level design platform. First, the capabilities of the design platform are discussed. Then, we present a comparison of CoDeL to a traditional DSP design platform. We examine CoDeL's code complexity, run time performance, and energy usage.

In chapter 5 we present a description and analysis of our automated clock gating framework. Chapter 6 describes our proposed micro-architectural power gating approach that can be used to reduce power dissipation.

Chapter 7 proposes optimal circuit partitioning strategies using Integer Linear Programming and simulated annealing, which can dramatically reduce power dissipation.

Chapter 8 concludes this dissertation with a summary of the presented work and provides suggestions for future research directions.

Chapter 2

System-Level Design

Speak properly, and in as few words as you can,
but always plainly; for the end of speech is not
ostentation, but to be understood.
- William Penn

2.1 Hardware Description Languages

A hardware description language (HDL) is any language from a class of computer languages used for formal description of electronic circuits. An HDL provides the ability to describe the temporal and spatial behavior of a circuit. Unlike a software programming language, it has explicit methods for expressing time and concurrency, which are essential in hardware.

In a broad sense, HDLs are used to design two types of hardware systems. First, they are used to design a dedicated integrated circuit, referred to as an Application Specific Integrated Circuit (ASIC). Second, they are used to target programmable logic devices (PLDs). The most common PLDs in use today are Field Programmable Gate Arrays (FPGAs). Using this approach, the hardware design can be coded, compiled and implemented on the PLD repetitively.

Currently, the hardware industry is dominated by two HDLs: VHDL [43] and Verilog [44]. Although, both languages provide somewhat differing features the de-

sign methodology using either language is virtually identical. These languages are normally used to design systems at the Register Transfer Level (RTL). They allow the description of the following attributes of a design:

- Control flow
- Iteration
- Hierarchy
- Registers of variable widths, bit vectors, and bit fields
- Explicitly specified sequential and parallel operations
- Arithmetic and logic operations

An HDL compiler is composed of two main stages: *synthesis* and *implementation*.

In *synthesis*, first the RTL description is converted into a structural description consisting of registers and combinational logic. Second, *logic optimization* is used to optimize the network of gates used to implement the required logic functions. This network of logic gates, called a *netlist*, and the various design constraints are an output of the synthesis phase.

The *implementation* phase performs the necessary steps to implement the design on the target device. It converts the logical design into a physical description. The first step is *translation*, where the input logic description and the constraints are merged to produce an intermediate logic description of the design. In the second step, called *mapping*, the logical description is mapped to structural components of the target device. The third step, *layout*, is divided into two phases: *placement* and *routing*. In *placement*, the various components are placed onto the device such that the area or cycle time is minimized. In *routing*, the various placed components are interconnected with wires.

Since HDLs, such as VHDL and Verilog, have been in existence since the early 1990s, and have received widespread industry support, there are highly effective design tools currently available to support these languages.

By the late 1990s, an evolution had begun toward higher level hardware design as RTL proved to be too low an abstraction level for designing the increasingly complex systems. What was needed was a language that could describe the entire system, including the hardware and software. This was the beginning of System-Level Design Languages.

2.2 System-Level Design Languages

The ultimate goal of a System-Level Design Language (SLDL) is to provide a single language platform to allow specification, analysis, design, and verification of an entire electronic system. The SLDL should allow the system developer to start at a high level system description and, through refinement steps, reach implementation. The design process normally begins with an abstract specification model and ends with an accurate implementation of the real system. The advantage of such a top-down approach is that all necessary design decisions can be made at an abstraction level where irrelevant details can be excluded. This allows designers to work with a model of minimum complexity.

The goals of SLDLs can be specified as follows.

- *Executability* - This is important for simulation. Simulation allows validation of the system specification as well as verification of design models throughout the design process.

- *Synthesizability* - The entire language should be synthesizable in order to obtain an implementation composed of software or hardware components.

- *Modularity* - The language should allow structural and behavioral hierarchy, so

the system can be decomposed into hierarchical components of reduced complexity. In addition, modularity is needed to separate computation from communication.

- *Completeness* - Concepts typically found in software and embedded systems should be supported. These include concurrency, synchronization, exception handling, timing, and explicit state transitions.

No existing SLDL supports all these goals completely. Each SLDL has its areas of strengths and weakness.

The current popular set of SLDLs can be divided into three categories. The first category consists of languages which extend existing HDLs. This category includes SystemVerilog [77], which is a set of extensions to the IEEE Verilog standard [44], and SpecCharts [64, 81], which extends VHDL [43].

The extensions in SystemVerilog and SpecCharts support additional data types similar to higher level languages, such as C, new procedural blocks to more clearly represent the intended design logic, and the notion of interfaces to group and abstract related ports and/or signals into a user declared module.

The second category includes languages which have extended or built upon the existing software languages. This set consists of JHDL [17], which extends the Java language, and, SpecC [33], SystemC [20], HandelC [47], Impulse C [45], and Catapult Synthesis [60] which extend the C/C++ programming language.

The third category consists of newly created languages. This includes the language Rosetta [15].

To allow us to effectively explore automated mechanisms to reduce power dissipation starting at the system level, we have developed a System-Level Design Language, CoDeL. CoDeL falls in the third category of newly created languages, although it shares several syntactic elements from the C programming language.

CoDeL (Controller Description Language) [75, 1, 3] is a rapid hardware design platform that allows circuit description at the algorithmic level. CoDeL compiles to create synthesizable VHDL code that can be simulated and synthesized using standard VHDL tools. CoDeL implements a design as a Finite State Machine with Datapath (FSMD) architecture. Thus, it has sufficient and detailed information on the usage of registers and functional units allowing various power reduction techniques. The CoDeL design platform is discussed further in chapter 4.

The objective of all SLDLs is to provide a higher level of abstraction for system development. Most provide a syntax and set of libraries to support features such as behavioral and structural hierarchy, concurrency, communication, synchronization, state transitions, exception handling and timing. In most cases, the control flow needs to be specified explicitly. It is only with HandelC, Impulse C and CoDeL that the abstraction level is even higher. These tools allow the designer to work at the algorithmic level. The compiler is able to automatically extract the control flow from the algorithmic description. Further, most hardware languages are concurrent and sequentiality is a special case. In HandelC, Impulse C and CoDeL, however, sequentiality is the default control flow. Parallelism must be explicitly specified in HandelC, while in CoDeL and ImpulseC, the compiler automatically parallelizes non-dependent assignment statements. Unlike HandelC and ImpulseC, CoDeL abstracts module interaction through ports and protocols, and has intrinsic support for fixed point computation making it highly effective in DSP applications. Further, an important distinction between CoDeL and ImpulseC and HandelC is that while CoDeL targets ASICs and FPGAs, ImpulseC and HandelC produce designs that can be targeted only to FPGAs.

An important feature missing from all current SLDLs, except CoDeL, is an inherent awareness of power dissipation. With the proposed power extensions of CoDeL that allow clock gating, power gating and partitioning, CoDeL takes an important

leap in the design of power efficient systems. Our CoDeL design platform is covered in more detail in chapter 4.

The discussion of SLDLs presented here is a very high level description of the general concept. There are several important concepts that have not been presented here for brevity. These include hardware/software co-design [31, 52] and high-level synthesis [74]. This is a very exciting and active field of research, which promises to revolutionize how computation systems are designed and developed in the near future.

-

Chapter 3

Low-Power Design

Microprocessor design has traditionally focused on
dynamic power consumption as a limiting factor in system integration.
As feature sizes shrink below 0.1 micron, static power is posing
new low-power design challenges.
- Kim et. al. [51]

3.1 Power and Energy

Power and energy are closely related terms which are often used interchangeably. However, their differences are important to understand. Power is the time rate of consumption of energy. Thus we have

$$P = \frac{\mathrm{d}E}{\mathrm{d}t}. \tag{3.1}$$

In the context of circuits, we require electrical energy to do some *work*, i.e. some computation. This energy used is dissipated as heat and electromagnetic radiation. It is common to rate a circuit by the amount of energy used to perform a task or the amount of power dissipated. Both these quantities are important and provide insight into different characteristics of the circuit component under examination.

Energy is an important indicator of battery life. Batteries store a finite amount of

energy, which allows some finite amount of work to be performed. Thus, to maximize battery life we would like to perform tasks by minimizing the amount of energy used.

If a given task is performed quickly, the power dissipation will be high, while the same task performed slowly will result in low power dissipation. In the ideal case where everything in these two environments is the same, the amount of work done, and consequently the energy used, is the same. In a practical environment, however, this is not the case. The physical characteristics of the circuits used to perform the computation at the different speeds dictate the power and energy requirements. First, there are parasitic effects leading to different energy requirements depending on the total run time of the computation. An example of such an effect is leakage current in the diffusion regions of the transistors (see section 3.2.1) which is a continuous effect leading to greater wasted energy as the run time is increased. Second, circuits designed to operate at different speeds usually employ dissimilar circuit components, component layouts and routing. These differences will lead to different power characteristics, ultimately leading to a different energy profile of the computation.

In many cases, the task demands a particular run time constraint. A particular example is real-time communication systems where a data stream needs to be handled at a fixed rate. In such cases, where the run times of the tasks are constrained, examining a circuit's energy is equivalent to examining the power dissipation.

The power dissipation of a circuit is an important indicator of heat and reliability. A higher power dissipation leads to more generated heat, resulting in costly cooling techniques and poor component reliability. In this case, minimizing energy is not the final objective. Rather, the power dissipation needs to be lowered. However, reducing energy is still an important design objective as it will result in a lower power dissipation.

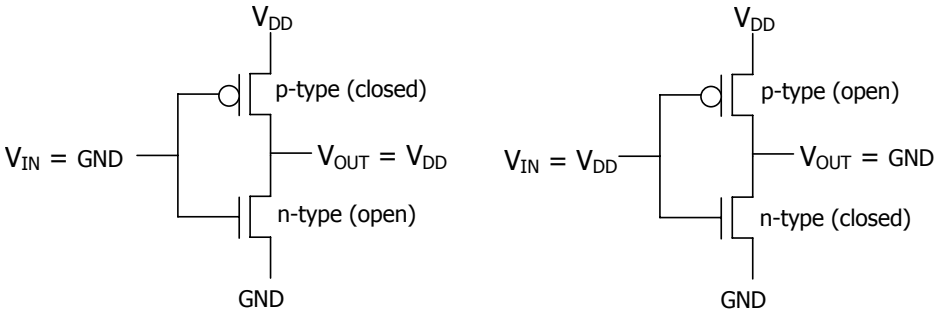


Figure 3.1: CMOS inverter

3.2 Power Dissipation

Today's integrated circuits are most commonly implemented using CMOS (Complementary Metal Oxide Silicon) technology. Power dissipation in digital CMOS circuits can be largely divided into two main categories:

- *Static dissipation*, where power is dissipated while the circuit is in steady state and not switching digital states.
- *Dynamic dissipation*, where power is dissipated due to changes in the digital state of the circuit.

3.2.1 Static Dissipation

In an ideal complementary CMOS gate, one of the transistors is always "OFF". As a result, no current flows into the gate terminal, and thus there is no DC path from V_{DD} to V_{SS}. This results in zero power dissipation. A CMOS inverter circuit is shown in figure 3.1. However, real CMOS transistors suffer from current leakage in the standby state due to two main causes.

First, there is some reverse bias leakage current at the junctions between the diffusion regions and the substrate. Second, when the gate-to-source voltage, V_{gs}, is less than the threshold voltage, V_t, a sub-threshold current passes through the transistor resulting in static power dissipation.

The leakage current in a CMOS transistor can be modeled by the following diode equation [84]

$$i_o = i_s \left(e^{\frac{qV_{DD}}{kT}} - 1 \right), \tag{3.2}$$

where i_s is the reverse saturation current, V_{DD} is the supply voltage, q is the electronic charge (1.602×10^{-19} C), k is Boltzmann's constant (1.38×10^{-23} J/K), and T is the temperature. The total static power dissipation of a circuit, P_{static}, is then simply the sum of the leakage power dissipated in each device, given by

$$P_{static} = \sum_{1}^{n} i_o V_{DD}, \tag{3.3}$$

where n is the number of devices, and V_{DD} is the supply voltage.

3.2.2 Dynamic Dissipation

When a CMOS transistor changes state, both the n- and p- transistors are "ON" for a brief moment. This results in a short current pulse. This is called *short-circuit* dissipation, and is usually not significant.

The current required to charge and discharge the output capacitive load is the dominant contributor to dynamic power dissipation. This is given by [84]

$$P_{dynamic} = \alpha C_L V_{DD}^2 f, \tag{3.4}$$

where α is the percentage switching activity of the circuit, C_L is the total capacitive load driven by the gate outputs, V_{DD} is the supply voltage, and f is the circuit frequency.

3.3 Reducing Dynamic Power Dissipation

Traditionally, the dynamic dissipation in integrated circuits has been quite dominant when compared to the static power dissipation, and has therefore received much

attention. To this effect, equation 3.4 has suggested the parameter values that should be reduced to lower power dissipation. We now examine some ideas that have been used by designers and architects to reduce these parameters.

3.3.1 Reducing Supply Voltage

We can see from equation 3.4 that the dynamic power of a system varies quadratically with the supply voltage V_{DD}. This means that if the supply voltage can be halved, the dynamic power can be reduced by a factor of 4! Therefore, reducing the supply voltage is one of the most effective methods of reducing the dynamic power.

However, the supply voltage cannot be reduced arbitrarily. There are adverse effects of reducing supply voltage which need to be considered. For a MOS transistor operating in the saturation region, as is normally the case, the drain-to-source current, i_{ds} is given by [84]

$$i_{ds} = \frac{\beta}{2} \left(V_{DD} - V_t \right)^2 , \tag{3.5}$$

where β is the transistor gain factor and incorporates several process dependent characteristics such as doping density, gate-oxide thickness, and the device geometry. We can see that as the supply voltage is reduced, the drain-to-source current is also reduced. This means less current is available for charging and discharging output capacitances leading to longer rise and fall times. This causes the circuit delays to rise significantly leading to poor performance. Therefore, the accepted design rule is to operate a circuit at the lowest possible supply voltage that meets the performance requirements.

One method to reduce power dissipation is to operate different parts of a chip at its own optimal voltage circuit [80, 27]. In this approach, multiple supply lines are routed to different subcircuits on a chip. The transfer signals from one voltage domain to another, level-shifter circuitry is needed. The overhead and complexity of this approach has limited its usefulness. An important variation on this method is

dynamic voltage scaling (DVS), where the supply voltage is altered in real-time as the performance requirements of a circuit change over time [25, 85]. One important application where DVS is extremely effective is in the microprocessor circuits of battery-operated portable computers [23]. In these systems, the supply voltage can be dynamically adjusted by the operating system based on the amount of work being executed.

3.3.2 Reducing Circuit Frequency

Given performance constraints circuit frequency can not be reduced arbitrarily. Further, as discussed earlier, circuit frequency is very closely connected to supply voltage. Thus the desired performance determines the required circuit frequency which in turn allows the optimal supply voltage to be used to minimize power dissipation.

However, the frequency requirements of a circuit may vary over time. In this case, dynamic frequency scaling may be used [57].

3.3.3 Reducing Capacitive Load

In a MOS transistor the gate capacitance is proportional to the area of the gate [84]. Thus, reducing the size of the transistor, called *technology scaling*, decreases the capacitance, and, hence, decreases dynamic power dissipation.

One approach to lowering capacitive load is to use custom circuits to perform the required computational tasks, rather than general purpose circuits. General purpose circuits are necessarily larger and more complex as they are designed to handle a wide array of different operations. Also, general purpose circuits need to be able to handle the largest possible data sizes in a large set of possible applications. Application specific circuits are constructed to contain only the required components and interconnections, making them much more energy efficient.

Driving global signals across a chip and accessing large, centralized memories, register banks and functional units are power-consuming tasks that must be avoided.

A solution to this problem is partitioning a design so that the locality of reference present in a given algorithm is preserved. This reduces the amount of power-hungry chip wide interactions. We have devised a method of optimally partitioning a circuit given an FSMD description into minimally interacting subcircuits [11].

3.3.4 Reducing Switching Activity

An important criterion to reducing power dissipation is to eliminate switching in any unused elements of the circuit. In many cases, changes to register values (writes) are completely unnecessary and thus wasteful of energy. If we can prevent these unnecessary state changes, we can lower power dissipation. A useful method for eliminating these unwanted state changes is to disable the clock signal to these devices. This is accomplished relatively easily using *gated clocks*.

For synchronous circuits, early results have shown that the continuously switching clock signal can account for as much as 45% of the system power [65]. Our tests using modern CMOS technologies suggest that the switching clock can contribute up to 60% of the total dynamic power dissipation [4]. Thus, reduction in the power used by the clock signal is key in reducing total power dissipation. *Gated clocks* can be used to reduce the clock switching in the clock tree and to the leaf registers and flip-flops, where feasible.

Clock Gating

Clock gating is an important technique in reducing dynamic power dissipation in CMOS circuits and has been explored by several researchers [18, 65, 71, 24, 16, 83, 28]. Figure 3.2 shows a simple clock gating mechanism.

Although the idea of clock gating is not new it is quite difficult to determine when gating can be applied. Some rely on the designer explicitly adding clock gates where needed [65], while others apply clock gates based on limited information about the underlying control logic of an architecture described at the RTL level [18, 71, 16].

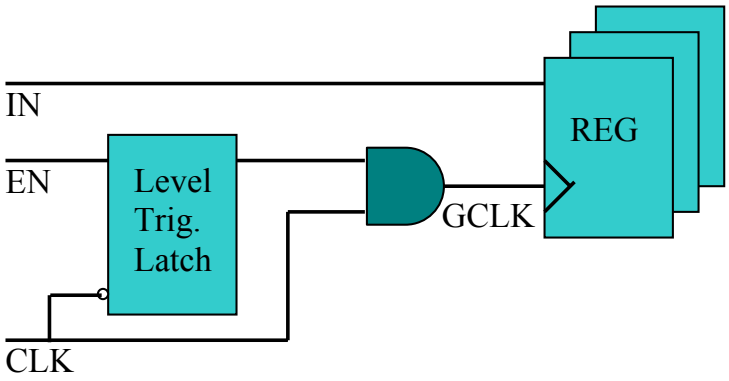


Figure 3.2: Clock gating circuit

There are techniques that automatically clock individual flip-flops [54] whose area requirements are too high with little power savings. Some try to clock gate large portions of the circuit, which require the entire set of circuit components to be static over a period of time to be activated [83, 28]. Other techniques rely on extracting idle states in finite state machines [71]. Due to limited information available at the time clock gating is applied, these techniques are unable to capture several instances where gating can be applied.

Our approach relies on a system-level compiler that has all the necessary state and register information available. It uses this information to disable devices whose states do not change and devices whose state changes are not necessary. Therefore, it has the ability to generate an efficient gating sequence for registers. Details of our clock gating mechanism is presented in chapter 5.

3.4 Reducing Static Power Dissipation

Although, reducing the size of the transistor decreases dynamic power dissipation, as the CMOS technology is scaled below 100nm, an exponential growth in subthreshold leakage current is seen [39, 21, 51]. As this trend continues, the leakage current is becoming a dominant source of total power dissipation in CMOS circuits [32, 26, 51, 53]. Therefore, much attention is given to the reduction of leakage, or static, power

in modern VLSI circuits.

Static power reduction techniques used can be broadly categorized into two categories: static methods and dynamic methods.

3.4.1 Static Methods

Static methods to control leakage attempt to intervene only during the design phase of the project. There are then no further mechanisms that are employed during the operation of this circuit. One common method is the use of dual voltage threshold transistors in the design. To maintain speed performance, the critical paths of the design use high-performance, high-leakage low V_T transistors, while the non-critical parts of the design use the high V_T low-leakage transistors [49]. These techniques, however, can become quite difficult to apply as the number of critical paths increases.

3.4.2 Dynamic Methods

Dynamic methods seek to implement constructs within the circuit to detect when components are "idle". When this idle mode is detected, the circuit enters a low-leakage or "standby" mode. There have been several approaches to reduce leakage at the circuit level using dynamic methods. These include body-bias control [29], dual-threshold domino circuits [48], input vector control [46], and power gating [68, 41].

3.4.3 Power Gating

Power gating relies on the detection of idle periods in the circuit. During these idle periods, the supply voltage can be switched off to the appropriate circuit component to conserve leakage power. At the end of the idle period, the supply voltage is restored to resume normal operation. Most power gating approaches rely on trying to predict idle periods for either storage structures (SRAMs) [30] or functional units [72, 41]. In [41], micro-architectural techniques for power gating functional units are presented. A "sleep" signal is applied to a power gating transistor to turn off the power supply voltage to the circuit block when a long idle time is detected. The "sleep" signal is

de-asserted and the voltage is restored once the circuit blocked needs to be used.

Here we propose methods that use a combination of static and dynamic techniques. We use static analysis to identify portions of the hardware that are not used as the execution trace progresses. These portions can then be switched off during periods of inactivity and switched back on when needed. We apply the proposed methods to effectively reduce the leakage power in individual registers. Our focus is to detect and implement power gating constructs at a high level of design, and eventually make it fully automated.

3.5 FSMD Partitioning

Partitioning is one technique used to facilitate logic isolation in FSMD circuits [42]. Normally the isolated circuit components are switched off (power gated) [56] or clock gated [42, 35] to conserve static or dynamic power, respectively. Two methods are generally employed for the partitioning of these sequential circuits. The first method relies on disabling parts of the FSM (Finite State Machine) controller. Here, the controller is partitioned into two or more mutually-exclusive FSMs. Each partition is then selectively clock gated [35] or power gated [56]. Thus, only one FSM is active at any given time, while the others are idle and their clocks are stopped, or their power is gated off. The second method tries to discover idle periods in one or more datapath components of the circuit. These components can then be clock gated or power gated. In [41], idle periods in the ALU are discovered and for these periods the ALU is power gated. In [4], individual registers are clock gated, while in [6], individual registers are power gated.

Although gating parts of either the controller or the datapath has been shown to be highly effective in reducing power, further savings can be achieved if both the controller and the datapath are considered together. This was proposed in [42], where a simple heuristic was used in a branch and bound method to partition the

FSMD. Further, the method was more suited for a clock gating environment. We use a more thorough and detailed model and hope to achieve better power reduction. Also, our model is well suited for a power gating environment where static power is of significant concern.

We formulate FSMD partitioning as both an an Integer Linear Programming (ILP) problem [10] and a non-linear programming problem which we solve using the Simulated Annealing (SA) algorithm [11]. Our objective is to maximize the isolation of circuit components by minimizing the communication between the partitioned FSMDs. This maximizes the number of components that can be put to sleep thus reducing the overall power dissipation.

Chapter 4

CoDeL

Language is the dress of thought.

- Samuel Johnson

CoDeL (Controller Description Language) [76, 75, 1, 3] is a system-level hardware design platform that targets the specification and design at the behavioral level. CoDeL is a procedural language in which the order of the statements implicitly represents the sequence of activities. It extracts the data and control flow from the program automatically, assigns the necessary hardware blocks and exploits inherent parallelism. It is similar to the C programming language and is therefore easy to learn. CoDeL introduces the concept of object-oriented hardware design and provides primitives, data structures and constructs for manipulating objects at the behavioral/RTL level. It includes a library of I/O protocols that simplify (sub)system interaction. The CoDeL compiler produces synthesizable VHDL code which can be targeted to any technology including FPGA or ASIC. Details of the language syntax can be found in appendix A.

We have extended the CoDeL compiler to automatically implement clock gating to lower dynamic power dissipation in CMOS circuits [5, 4, 7]. The power gating [9, 6] and partitioning [10, 11] mechanisms are not fully automated yet. However, the compiler provides information which can be used to evaluate these techniques

and implement them manually.

4.1 CoDeL Compiler

The CoDeL compiler is written in C and it compiles a source CoDeL program to produce synthesizable VHDL code. The compiler consists of the recursive, descent based [14, 40] with single token lookahead, parser and the VHDL builder.

The VHDL builder implements the data path as a RTL description, where data is stored in registers, operations are effected by a combinational circuit(s) and the results are stored back in a register. The control path is extracted automatically and sequences which operations are to take place and when the results are to be stored in the registers. Optimization includes automatic parallelization of non-dependent assignment statements. As such, some assignments can be scheduled in parallel during the same machine state and hence reduce the state count significantly. This feature schedules multiple register assignments simultaneously and hence improves the efficiency and speed of the synthesized hardware.

4.2 Performance Evaluation

4.2.1 Evaluation Framework

To evaluate the CoDeL platform we have used a set of kernel circuits from the DSP-stone benchmark [86]. The DSPstone benchmark has been shown to be effective in measuring the performance of DSP compilers and processors. This benchmark consists of three suites:

- **Application benchmarks** - A complex application. Specifically, the ADPCM transcoder is used.
- **DSP Kernel Benchmarks** - Set of 14 code fragments commonly used in DSP algorithms. These include FIR/IIR filters, FFTs, etc.

Table 4.1: DSPstone benchmark circuits - DSP kernel suite

Kernels	
real_update	n_real_updates
complex_update	n_complex_updates
dot_product	convolution
mat1x3	matrix
fir	fir2dim
iir_one_biquad	iir_n_biquads
lms	

- **C Kernel Benchmarks** - Set of typical C statements, such as loops, function calls, etc.

In our study we have chosen the DSP kernel benchmarks as they are the most representative set of operations which will typically be implemented using CoDeL in a DSP environment[1]. This DSP specific kernel suite consists of the 13 code fragments shown in table 4.1.

In presenting the DSPstone benchmark, the authors propose a methodology whereby the performance between hand-written assembly code and the assembly code generated by the compiler is compared. We do not have hand-coded HDL implementations of the DSPstone kernels. Rather we use the kernels to compare the performance of the architectures produced by the CoDeL compiler, with the compiler implementations on a modern DSP.

All kernels from the DSP benchmark suite are implemented using CoDeL and compiled to generate synthesizable VHDL. For data storage, a dual port memory is implemented in VHDL for simulation. In all cases, the memory is considered to be the fastest possible (zero wait state). Further, the power contribution of the memory is not considered in our power analysis since the memory module is constructed as a

[1]Although implementation of a large application, such as an ADPCM transcoder, is technically possible using CoDeL, it is rather cumbersome at this point due to the lack of an effective debugging and simulation tool at the design level. Thus, we have have not attempted such an implementation.

Table 4.2: C, CoDeL, VHDL code complexity

	Lines of Code		
Kernel	**C**	**CoDeL**	**VHDL**
real_update	39	39	206
n_real_updates	39	61	349
complex_update	41	52	301
n_complex_updates	50	94	522
dot_product	34	38	215
mat1x3	61	95	407
matrix	66	90	417
convolution	42	74	314
fir	54	85	431
fir2dim	94	160	1175
iir_one_biquad	37	76	351
iir_n_biquads	62	105	547
lms	71	128	637

black box used only for simulation and does not accurately reflect the specifications of a memory module that may be used in practice. This is reasonable since even DSPs normally do not report power usage with memory under consideration. However, the power reported for DSPs generally includes the dissipation in the caches. We have attempted to make an accurate comparison by accounting for the power dissipation in the cache for the DSP.

We compare the performance of CoDeL designed circuits to the TMS320C6416 DSP developed by Texas Instruments [79]. The TMS320C6416 is one of the highest performance DSP chips available today and provides a good comparison. To get execution results we have used a C compiler and a cycle accurate simulator provided by Texas Instruments as part of their Code Composer Studio 3.1 development environment.

Table 4.3: CoDeL vs. DSP (Raw Results)

	CoDeL (1.2 V)			**TMS320C6416 (1.4 V)**			**Energy**
	625 MHz			**600 MHz**			**Ratio**
Kernel	**Cycles**	**Power** (μW)	**Energy** (pJ)	**Cycles**	**Energy** (Incl. Cache) ($\times 10^3$ pJ)	**Energy** (Excl. Cache) ($\times 10^3$ pJ)	**TMS/ CoDeL**
real_update	5	534.8	4.3	21	13.7	9.6	2233.4
n_real_updates	114	658.0	120.0	73	47.5	33.2	276.7
complex_update	8	1324.8	17.0	39	25.4	17.7	1046.5
n_complex_updates	225	1250.9	450.3	163	106.0	74.2	164.7
dot_product	5	819.2	6.6	26	16.9	11.8	1805.0
mat1x3	49	531.5	41.7	37	24.1	16.8	404.0
matrix	4751	735.5	5590.6	3560	2314.0	1619.8	289.7
convolution	63	318.6	32.1	49	31.9	22.3	694.3
fir	99	521.6	82.6	94	61.1	42.8	517.7
fir2dim	565	914.3	826.5	390	253.5	177.5	214.7
iir_one_biquad	8	388.1	5.0	45	29.3	20.5	4121.7
iir_n_biquads	73	1087.4	127.0	70	45.5	31.9	250.8
lms	229	1165.9	427.2	143	93.0	65.1	152.3
Total	6194	10250.4	7730.8	4710	3061.5	2143.1	
Geometric Mean							532.0

4.2.2 Results

Code Complexity

Table 4.2 shows code complexity, as measured by the number of lines of code in the various language environments. Compared to C we see that CoDeL requires about 60% more lines of code, which is reasonable and does not pose too much of a hurdle in describing hardware architectures. Examining VHDL code complexity we find that the designs use about 5 times as many lines of code compared to CoDeL. In chapter 5 we introduce automated clock gating in CoDeL to reduce dynamic power. Comparing the VHDL description for these clock gated designs to CoDeL we find that nearly 10 times the number of lines of code are needed for VHDL. This shows that CoDeL significantly reduces the complexity of describing efficient hardware architectures.

DSP Comparison

One of the issues we are exploring is the viability of directly synthesizing a particular algorithm in silicon. We compare, therefore, the CoDeL implementation of the DSPstone benchmark suite to that of implementing the same using a modern DSP.

Table 4.4: Energy Delay Product (EDP)

Kernel	**CoDeL** ($\mu s \cdot pJ$)	**DSP** ($\mu s \cdot pJ$)	**Ratio (DSP/CoDeL)**
real_update	0.03	334.4	9771.1
n_real_updates	21.89	4041.2	184.6
complex_update	0.22	1153.4	5314.1
n_complex_updates	162.11	20148.2	124.3
dot_product	0.05	512.6	9777.3
mat1x3	3.27	1038.2	317.8
matrix	42497.48	9610813.3	226.2
convolution	3.24	1820.8	562.5
fir	13.09	6700.6	512.0
fir2dim	747.16	115342.5	154.4
iir_one_biquad	0.06	1535.6	24150.9
iir_n_biquads	14.83	3715.8	250.5
lms	156.52	15507.2	99.1
Geo Mean			746.4

In tables 4.3 and 4.4 we provide comparative execution and energy results for CoDeL and the TMS320C6416 DSP processor. It is important to note that the CoDeL results presented here do not employ any power optimization techniques such as clock or power gating. Employing these techniques would make the power dissipation numbers even lower.

For evaluation, we have used the 600 MHz version of the TMS320C6416 DSP here to provide an accurate comparison with our 625 MHz CoDeL circuits. However, this DSP is available in speeds up to 1 GHz [79]. The CoDeL designs are synthesized using the high performance 90nm TSMC VLSI technology using a 625 MHz clock. This is the fastest possible clock speed for all our kernel architectures without any hand optimization of the generated VHDL code. The TMS320C6416 DSP also uses a 90nm VLSI process technology.

Compared to the DSP, we see that CoDeL is less cycle efficient in most cases and on average[2]. We find that CoDeL performs better on kernels that do not employ

[2]The CoDeL cycle counts we present here are not affected by the clock gating mechanisms used.

loops. The kernels that utilize loops suffer in CoDeL. The performance of the kernels in CoDeL can be further improved using the following techniques. One or more of these methods are employed by most modern DSPs. First, the lack of the commonly used MAC (multiply-accumulate) instruction in CoDeL causes it to take two cycles while the DSPs are able to perform it in one cycle. In the 13 kernels we have examined, we find that using the MAC instruction could save 2% to 50% clock cycles. In total, we find that 384 out of 6194 (6%) cycles could be saved. Second, CoDeL does not employ many major compiler optimizations that are present in the mature compilers used for DSPs. These include pipelining, loop optimizations, and branch prediction.

In table 4.3, we also present the total power (dynamic + static) dissipated by the CoDeL designed circuits when CoDeL and Synopsys clock gating is used. Using this power we can compute the required energy as $Energy = Power \times Time$, where $Time$ is the number of cycles times the clock period. For the TMS320C6416 DSP running at 600 MHz, the reported average power is 390 mW based on 60% CPU utilization [79]. This reported DSP power includes a 32kB L1 cache, along with the CPU. We have not included any caching environment in our power results. Thus, we now provide an estimate for just the CPU of the DSP, discounting the L1 cache.

In [38], the power dissipation of a 32kB data cache is determined to be 118 μW/MHz using a DSP benchmark. Further 11.3 μW/MHz is measured for a memory management unit. For the TMS320C6416 DSP running at 600 MHz, this provides a power dissipation estimate of 78 mW. This is 20% of the total power of 390 mW. This is typical of cache power and similar percentages are available in literature. For example, in commercial processors such as Pentium Pro [59], Alpha 21264 [37] and StrongARM SA-110 [62], caches consume 33%, 16% and 43% of the total chip power, respectively.

Using this information we conservatively estimate the cache power on the TMS320C6416 DSP to be 30% of the reported 390 mW. Using this power estimate, the total en-

ergy required for the kernels on the TMS320C6416 core, excluding the cache, can be obtained by reducing the TMS320C6416 entries in table 3 by 30%, resulting in a total energy requirement of 2143×10^3 pJ. Comparing this energy requirement for the DSP with the CoDeL implemented kernels, we see that the total energy required by the DSP is about 277 times greater than the customized circuits developed by the CoDeL platform.

As an estimate of the expected energy energy savings per kernel, we also examine the geometric mean of the CoDeL vs. DSP energy ratios. This shows that the energy required on a DSP is 532 times that of a customized circuit described using CoDeL.

In table 4.4 we see a comparison of the Energy Delay Product for designs implemented in CoDeL and on a commercial DSP. We see that CoDeL provides significant energy advantages even when the execution delay is taken into consideration.

4.3 Summary

We have presented a high level VLSI development framework, CoDeL, for the design of customized hardware architectures. This framework supports the use of fixed point data structures for efficient description of DSP style algorithms.

In comparing the CoDeL platform to a DSP processor, we find that CoDeL allows algorithm description with somewhat higher code complexity than the C language normally used for DSPs, but provides dramatically lower energy consumption. The execution performance of CoDeL circuits compared to a modern DSP (TMS320C6416) is lower but this is largely due to better loop optimization in the C compiler used for the DSP. We expect that further development of the CoDeL compiler would result in better performing systems.

Chapter 5

Clock Gating

The clock tree is a good target for power reduction
in processor designs because it switches all the time,
consuming power every cycle.
- Garrett et. al. [36]

We have developed extensions to the CoDeL compiler which implement automated clock gating to lower dynamic power dissipation in CMOS circuits [5, 4, 7]. To estimate these power savings from clock gating, we have developed an analysis framework, which allows quick and accurate power savings estimation based on the CoDeL description at the behavioral level. This estimation framework is built into the CoDeL compiler and the estimates are output upon design compilation.

CoDeL uses a sequential machine to determine the sequence of operations and data transfers in and out of registers. Because of this sequential machine, we know the exact time of the events, and we can anticipate them. Although we do not know how many writes will happen, the state information allows us to know when they will happen. This allows us to build the appropriate gating logic and open the gate at these states.

The compiler gathers information on register reads and writes in each state of the finite state machine. We express reads as r_i^s, where a read is performed for register

i in state s. Similarly, writes to register i in state s are referred to as w_i^s. The set of all registers written in state s is w^s, while the set of all registers read in state s is r^s. Let the total number of registers be N and the total number of states be M, $i \in [1, N]$, and $s \in [1, M]$.

Using state transition information and the set of reads and writes in each state we can determine the register writes which are necessary and which are useless. The following rules determine that a particular write (w_i^s) is useless.

- Multiple writes without any read in between means all but the last write are useless.
- All writes after the last read of a further-unused register are useless.

Using these rules the set of writes w^s is minimized to include only those register writes that are necessary. We call this minimized set $\tilde{w}^s$. It should be noted that it is not possible to discover all useless writes through a pure static analysis of the state machine. A run-time mechanism is needed to discover all such useless writes. We do not explore any dynamic, run-time mechanism due to the associated area overhead. We rely on a purely static approach that can be automated at compile time.

We then implement the clock gating mechanism. Since the register outputs are valid when the clock is not active, all reads of a register, r_i, can be performed while the clock is gated. It is only during a register write, w_i, that the clock needs to be activated to latch the updated input signal into the register. Let the complete set of states be S. For a register i let the set of write states be $S_i^w \subseteq S$. Then, for each register, i, a clock gate, $g_i \in \{0, 1\}$, is created, which enables the clock to the register only during its write states $s \in S_i^w$, and disables the clock in all other states $s \notin S_i^w$. Thus, we have $g_i = s \in S_i^w$.

Since CoDeL implements designs as a Moore finite state machine, the clock gates for the registers are simply a function of the current state. Thus, simple combinational

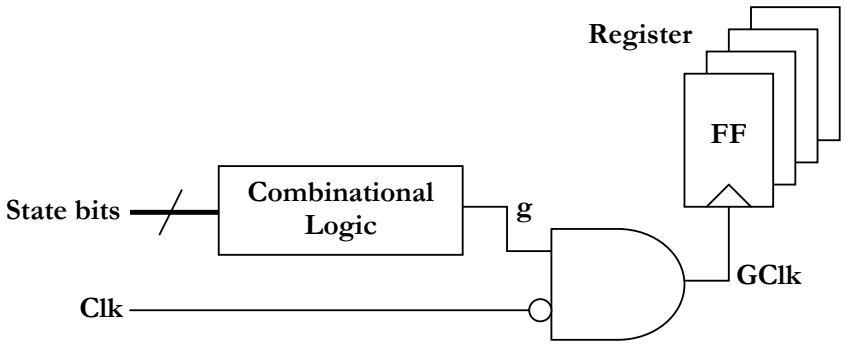


Figure 5.1: Clock gating circuit

logic can be used to set up a clock gate. In the case of a Mealy machine, the clock gate for some registers becomes a function of the current state and the inputs, making the gating logic more complex. This has not been explored in this thesis. It should be noted that the registers encoding the state, i.e. the control path memory elements, are not clock gated. To ensure the state value stabilizes, and setup and hold times are met for the register inputs, we use the falling edge of the clock to clock the gated registers. Thus, the clock signal for register i is given by

$$\text{gclk}_i = \overline{\text{clk}} \ \ \text{AND} \ \ g_i.$$

The minimum number of bits for a register which should be clock gated is left as a configurable parameter, ξ, which is an input to the CoDeL compiler. Thus, a register r is clock gated only if its word length $|r|$ is greater than or equal to ξ.

A principal block diagram of the clock gating mechanism is presented in figure 5.1. Figure 5.2 provides a timing diagram for a clock gated register gated on in state `x`. The next section provides an example of how clock gating is performed for an FSMD circuit designed using CoDeL.

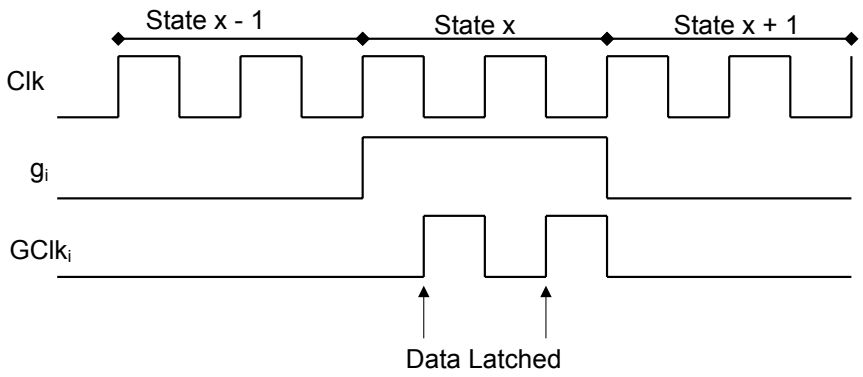


Figure 5.2: Clock gating timing

Table 5.1: FSMD implementation of the *real_update* kernel ($d = c + a * b$)

State(s)	Activity
0...2	Wait for `Start` signal; Indicate busy with `Ready` signal.
3...6	Update memory with values for `a`, `b`, `c` and `d`.
7	Start profiling.
8	Get a value for `a` and `b` from memory.
9	Get a value for `c` from memory.
10	Calculate $a * b$. Assign value to `d`.
11	Calculate $c + d$. Assign value to `d`.
12	Write `d` to memory.
13	Stop profiling.

5.1 Example

Table 5.1 presents an outline of the resulting finite state machine implementation of the simplest kernel, *real_update*, as an example of a circuit compiled using CoDeL. The CoDeL source code listing can be found in section B.2 We see that the complete implementation requires 14 states. However, states zero to seven, and state 13 perform only setup operations and are therefore not included in the profiling. This is also the case for the reported DSP results implemented in C.

Simulation of the CoDeL compiled design shows that each state from eight to 12 is visited once per clock cycle. Thus, to perform the desired kernel functionality five clock cycles are needed.

The gated elements in this circuit are the registers `a`, `b`, `c` and `d`, as well as the data latches used for the output ports interfacing to memory and a fixed point unit (FXU). Considering just the data registers, we find that, according to table 5.1, the value of `a` and `b` is updated in state eight, `c` is updated in state 9, and `d` is updated in states 10 and 11. This provides us with the states when the clock needs to be enabled for these registers. For all other states, the clock is gated off.

5.2 Power Savings Estimation Framework

We now present a framework to estimate the expected dynamic power savings from automated clock gating using CoDeL. The dynamic power savings obtained can be divided into two parts. In the first part, we examine the saved power due to the removal of useless switching, while in the second part we examine the savings due to the reduction of clock fanning.

5.2.1 Useless Switching

For the entire state machine the total number of bits that are potentially written to, W, can be calculated as

$$W = \sum_{s=1}^{M} \sum_{w_i^s \in w^s} |w_i^s| \,,$$

where M is the number of states, w^s is the unoptimized set of written registers in state s, and $|w_i^s|$ represents the word length of register w_i^s. The optimized total number of written bits, $\tilde{W}$, needs to include all those non-gated registers, whose word length is less than the threshold ξ. It can be represented as

$$\tilde{W} = \sum_{s=1}^{M} \left[\sum_{w_i^s \in \tilde{w}^s} |w_i^s| + \sum_{w_i^s \in w^s, w_i^s \notin \tilde{w}^s, |w_i^s| < \xi} |w_i^s| \right].$$

Not taking into account the clock gating overhead, the fraction of clock power saved due to the removal of useless switching, P_s, is proportional to the fraction of writes saved.

$$P_s = 1 - \frac{\phi \tilde{W}}{\phi W} = 1 - \frac{\tilde{W}}{W} \tag{5.1}$$

where the factor ϕ represents the fraction of bits that change value, on average, when a register's value changes.

5.2.2 Clock Switching

The total number of clocked register bit states is given by

$$C = M \times \sum_{i=1}^{N} |r_i| \,,$$

where r_i is the ith register, and there are a total of N registers.

After clock gating, the number of clocked register bit states is given by

$$\tilde{C} = \sum_{s=1}^{M} \left[\sum_{w_i^s \in \tilde{w}^s, \left|w_i^s\right| \geq \xi} |w_i^s| + \sum_{|r_i|<\xi} |r_i| \right].$$

Not taking into account the clock gating overhead, the fraction of power saved due to the reduction in clock switching, P_c, is proportional to the fraction of clock cycles saved, and is given by

$$P_c = 1 - \frac{\tilde{C}}{C}. \tag{5.2}$$

5.2.3 Clock Gating Overhead

We call the additional power requirement for clock gating P_g, which is a monotonically increasing function of the number of clock gated bits and the frequency of changes in the state of these gates. We can approximate this overhead by summing the additional switching activity and the additional clocking requirement.

The proportion of additional switching activity, ρ_s, is

$$\rho_s = \frac{\sum_{s=1}^{M} \sum_{i=1}^{N} \begin{cases} 1, \text{if } w_i^s \in \tilde{w}^s, |w_i^s| \geq \xi \\ 0, \text{otherwise} \end{cases}}{\phi W} \tag{5.3}$$

where, as before, the factor ϕ represents the average fraction of bits of a register that change value.

The proportion of additional clocking overhead, ρ_c, is given by

$$\rho_c = \frac{M \cdot \sum_{i=1}^{N} \{1 \text{ if } |r_i| \geq \xi; 0 \text{ otherwise}\}}{C}. \tag{5.4}$$

The overall gating overhead can now be stated as

$$P_g = \alpha_s \rho_s + \alpha_c \rho_c, \tag{5.5}$$

where α_s is the fraction of dynamic power dissipation attributable to register switching activity and α_c is the fraction of dissipation attributable to clocking.

5.2.4 Total Power Saved

The total power saved P is the sum of savings due to the removal of useless switching, P_s, and the saving due to reduction in clock switching, P_c. We also need to take into account the clock gating overhead, P_g. The total power saving is then given by

$$P = \alpha_s P_s + \alpha_c P_c - P_g, \tag{5.6}$$

where, as before, α_s is the fraction of dynamic power dissipation attributable to register switching activity and α_c is the fraction of dissipation attributable to clocking.

An estimate of α_s and α_c can be obtained by examining the proportion of estimated switching activity due to register updates, W, and register clocking, C.

$$\alpha_s = \frac{\phi W}{\phi W + C} \tag{5.7}$$

$$\alpha_c = \frac{C}{\phi W + C} = 1 - \alpha_s. \tag{5.8}$$

5.3 Evaluation

To evaluate the CoDeL platform we have used the set of DSP kernel benchmark circuits from the DSPstone benchmark [86] and a set of application circuits. The DSP kernel suite consists of the code fragments shown in table 4.1. The application circuits examined consist of the following integer algorithms commonly found in DSP applications.

- A simple 16-bit counter.
- A 5/3 Discrete Wavelet Transform (DWT) using the lifting technique [34,

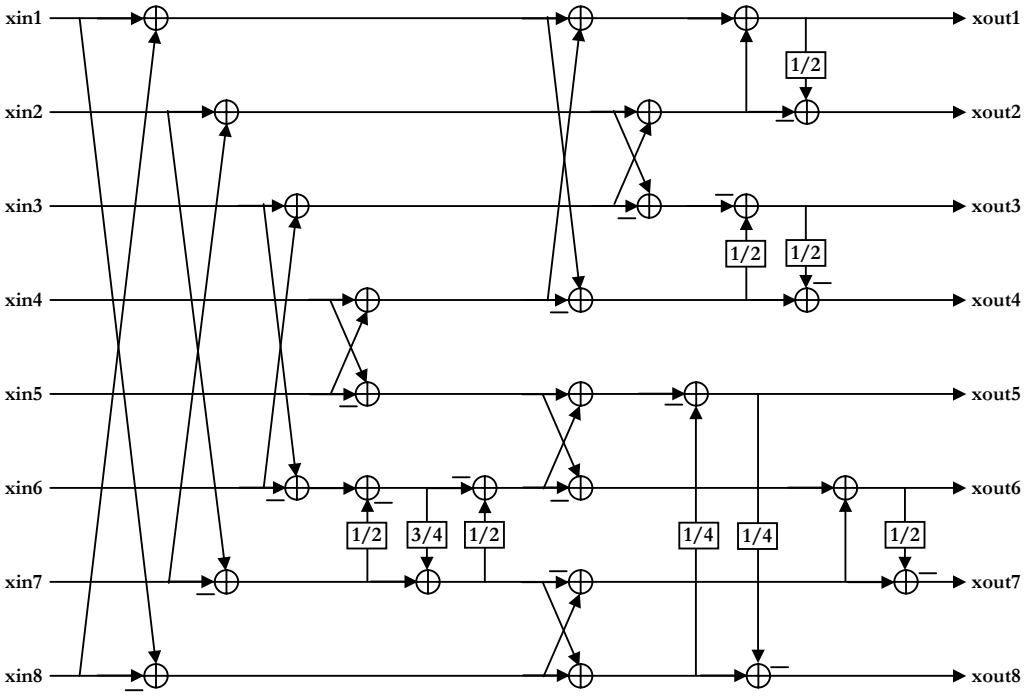


Figure 5.3: Structure of the 8-point multiplierless DCT approximation

78]. The 5/3 DWT is used to perform lossless compression of images in the JPEG2000 standard [70].

- A multiplierless approximation to the eight-point Discrete Cosine Transform (DCT) [55]. The DCT forms the heart of the JPEG and MPEG standards [66, 61]. From [55] we use the C7 DCT based on Chen's factorization. The implemented structure of this transform is presented in figure 5.3.

- An integer transform used in the H.264 (MPEG4 Part 10) standard [58]. H.264 is an important, new video compression standard suitable for very high data compression. The implemented structure of this transform is presented in figure 5.4.

All kernels from the DSP benchmark suite and the application circuits are implemented using CoDeL and compiled to generate synthesizable VHDL. These circuits are synthesized using the Synopsys Design Compiler using three TSMC 90nm CMOS

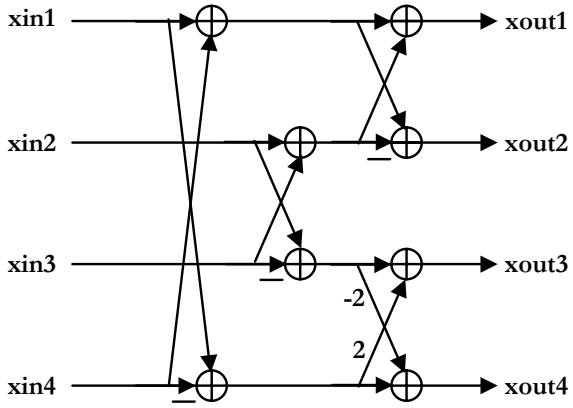


Figure 5.4: Structure of the H.264 Integer Transform

technologies: general purpose, high performance, and low power.

5.4 Benchmarking Results

5.4.1 DSPstone Dynamic Power

All power results presented here are for the DSPstone kernel modules implemented using CoDeL. The results for the application circuits are presented in section 5.4.4. In all cases, except the DWT application circuit, randomized, statistical data is used for simulation based results. For the DWT, we perform the transform on 128×128 grayscale images of *Lena* and *Boats* [1].

We present a comparison of four scenarios: NCG uses no clock gating, CCG uses automated CoDeL clock gating, SCG uses automated clock gating using Synopsys, and SCCG uses clock gating using both CoDeL and Synopsys. In the case of SCCG, CoDeL applies clock gating for the registers first during compilation of the CoDeL description, which produces a VHDL description. Then Synopsys clock gating is applied during VHDL synthesis. In all cases, all registers wider than two bits are clock gated. Further, all results are presented for the gate level design, before placement and routing. To obtain the power dissipation in the circuits, Synopsys' Power

[1]The standard *Lena* and *Boats* images are 512×512 (http://www.cs.tut.fi/~foi/GCF-BM3D/). We used IrfanView, which uses the Lanczos resampling algorithm, to reduce these to 128×128.

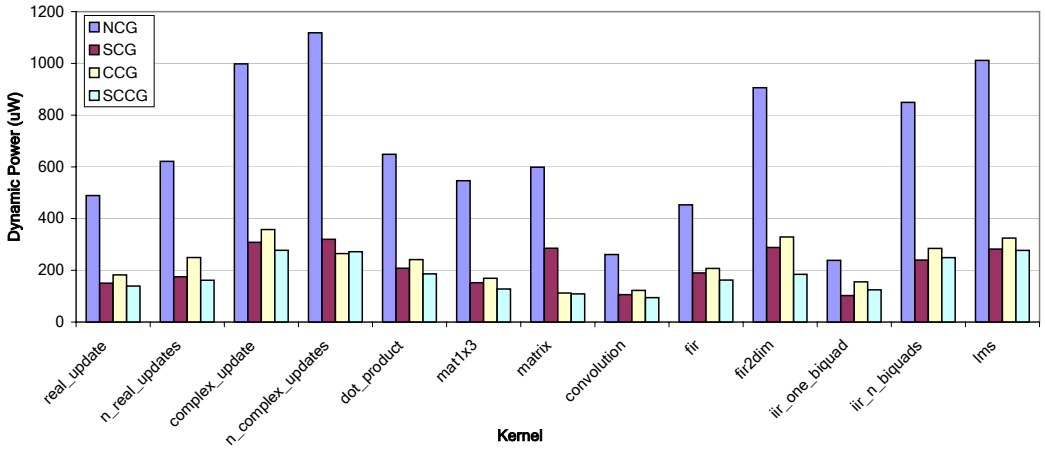


Figure 5.5: Power dissipation - 400 MHz - General purpose 90nm

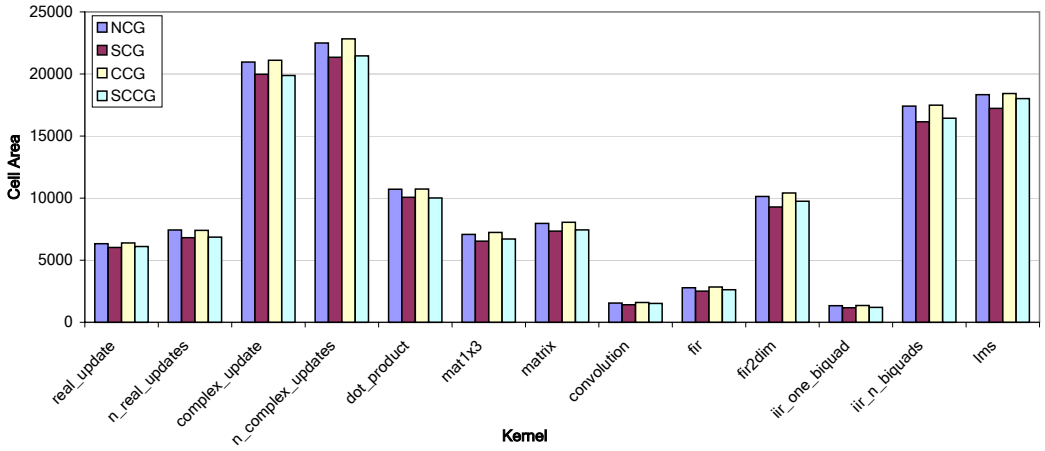


Figure 5.6: Cell Area - 400 MHz - General purpose 90nm

Compiler was used with switching activity annotated through simulation.

Figure 5.5 shows the dynamic power dissipation for the kernels using the various gating methods. These results use the general purpose 90nm TSMC CMOS technology for synthesis. We find that in most cases clock gating performed just by CoDeL (CCG) is less effective at reducing dynamic power dissipation than clock gating performed just by the Synopsys Power Compiler (SCG). However, in virtually all cases where both CoDeL and Synopsys clock gating (SCCG) is used, the power savings are maximized.

Figure 5.6 shows area results for the various designs. We see that the CoDeL gated designs (CCG) exhibit the largest area requirement, which is slightly higher,

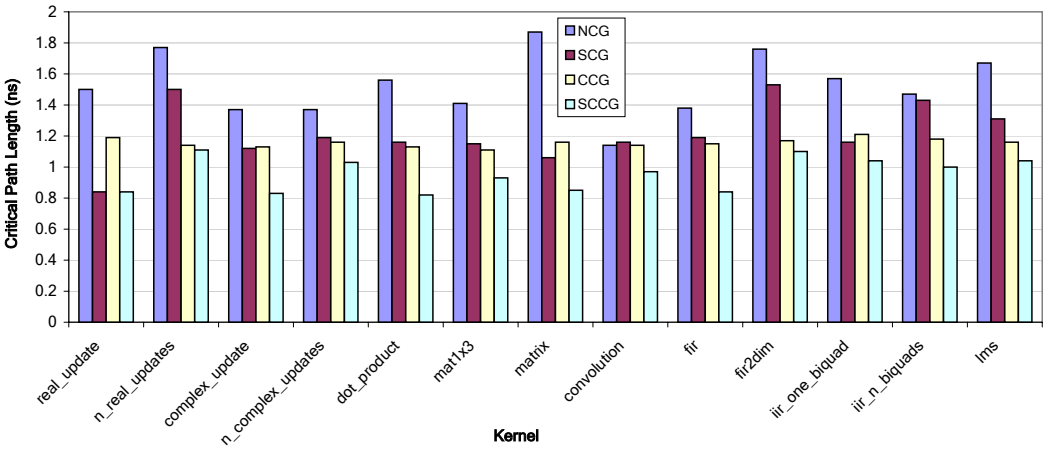


Figure 5.7: Critical Path Length - 400 MHz - General purpose 90nm

in most cases, than the non gated designs (NCG) (1.3% higher on average). The Synopsys gated designs (SCG) use the least cell area, while the CoDeL and Synopsys gated designs (SCCG) show somewhat higher area usage. On average, compared to NCG designs, SCG exhibits 7.5% lower area, while SCCG exhibits 5.3% lower area usage. The lower area with Synopsys clock gating is because this gating method allows Synopsys to use more area efficient cells to implement the sequential elements, namely the flip flops. The increase in area using CoDeL clock gating, as compared to using no clock gating, can be attributed to the increase in combinational logic required to generate the clock enable signal.

Examining the results of figures 5.5 and 5.6, we find that CoDeL clock gating effectively reduces clock switching activity, while using Synopsys clock gating results in the use of more area and power efficient cells resulting in overall power and area reduction. It is because of this phenomenon that the combination of Synopsys and CoDeL clock gating provides the most favorable results.

Figure 5.7 shows the critical path lengths for the various designs. We see that the non gated designs (NCG) exhibit the longest paths while the CoDeL and Synopsys gated designs (SCCG) show the shortest critical paths. The reason for this result lies in the optimizing behavior of the Synopsys compiler. As synthesis proceeds, the

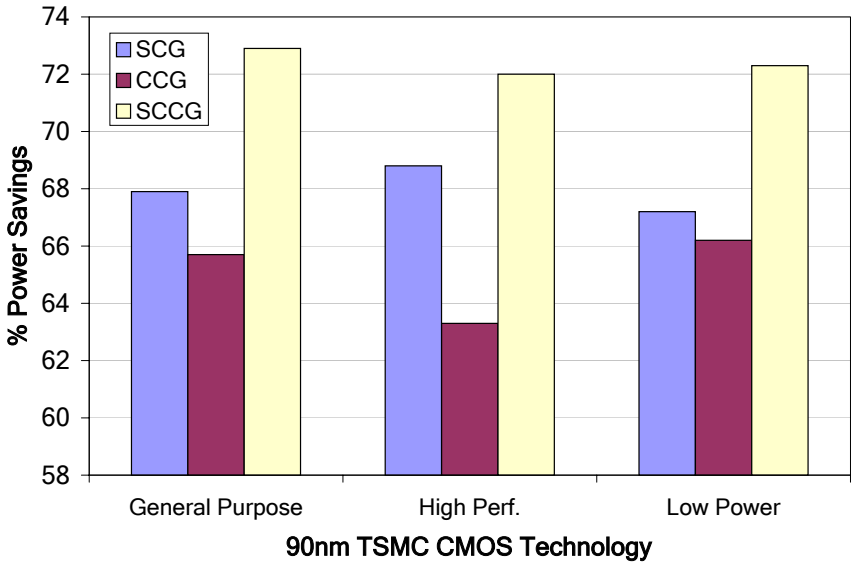


Figure 5.8: Average percentage dynamic power savings

Synopsys compiler chooses alternate transistor level components to satisfy the various constraints. These components have varying power, area and timing characteristics. In our results, the shorter path lengths for gated designs may mean that the compiler worked harder to satisfy tighter timing constraints when clock gating is added leading to shorter critical paths. Further investigation is required to clarify and confirm this hypothesis.

Figure 5.8 provides the average percentage dynamic power savings obtained using clock gating for the three 90nm CMOS technologies. The power savings are calculated as the percentage of power saved compared with the designs without any clock gating (NCG). As an example, the power savings from CoDeL clock gating (CCG) are calculated as

$$\frac{\text{Avg. NCG Power} - \text{Avg. CCG Power}}{\text{Avg. NCG Power}} \times 100\%,$$

where the averages are arithmetic means.

The results show that using the general purpose 90nm cell library, on average, CoDeL clock gating saves 65.7% power, Synopsys clock gating saves 67.9%, and combining CoDeL and Synopsys clock gating saves 72.9%. Further we find that the

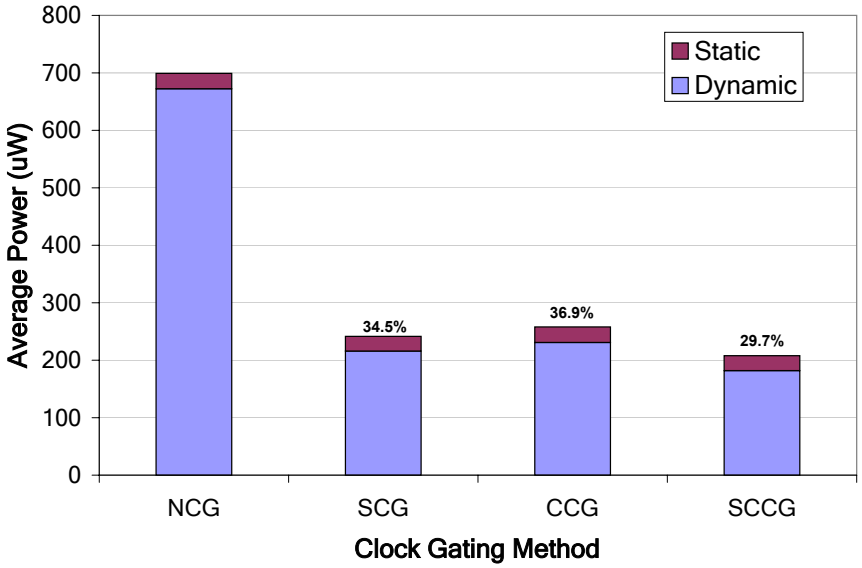


Figure 5.9: Average power - 400 MHz - General purpose 90nm

use of CoDeL and Synopsys clock gating gives 16% additional dynamic power savings than using just Synopsys clock gating.

The high performance and low power 90nm cell libraries exhibit similar results. Using the high performance library we find that CoDeL and Synopsys clock gating is 10% better than Synopsys clock gating, while using the low power library we get an additional 16% savings.

Figures 5.9, 5.10 and 5.11 show average dynamic and static power dissipation for the various clock gating methods and the various CMOS technologies. We see that the general purpose cell library gives the lowest overall power dissipation. However, the low power library gives very low static power dissipation. The high performance library dissipates about three times the power as the general purpose and low power libraries but it allows the fastest clock rate of 625 MHz. Examining the various clock gating methods across different technologies, we see that the relative power savings are quite similar. CoDeL and Synopsys clock gating (SCCG) provide the most power savings, while Synopsys clock gating (SCG) and CoDeL clock gating (CCG) provide lower savings in power dissipation.

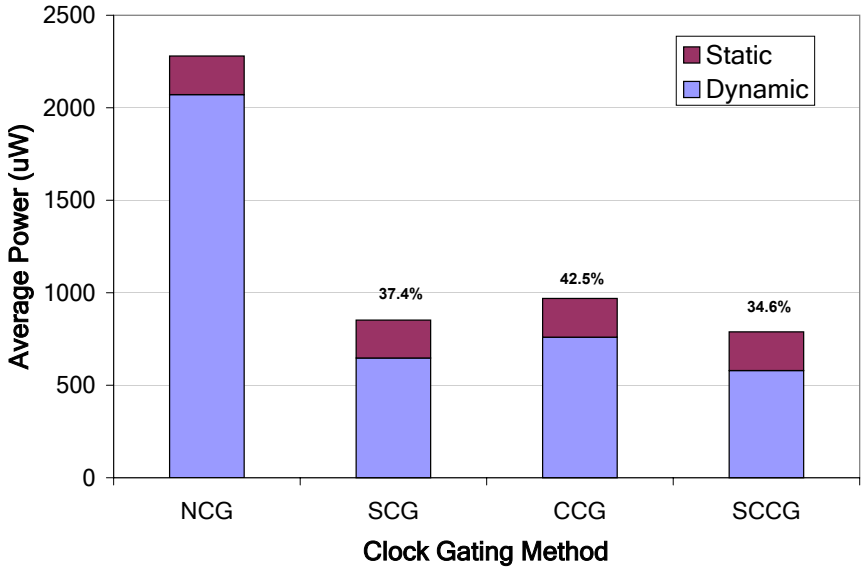


Figure 5.10: Average power - 625 MHz - High performance 90nm

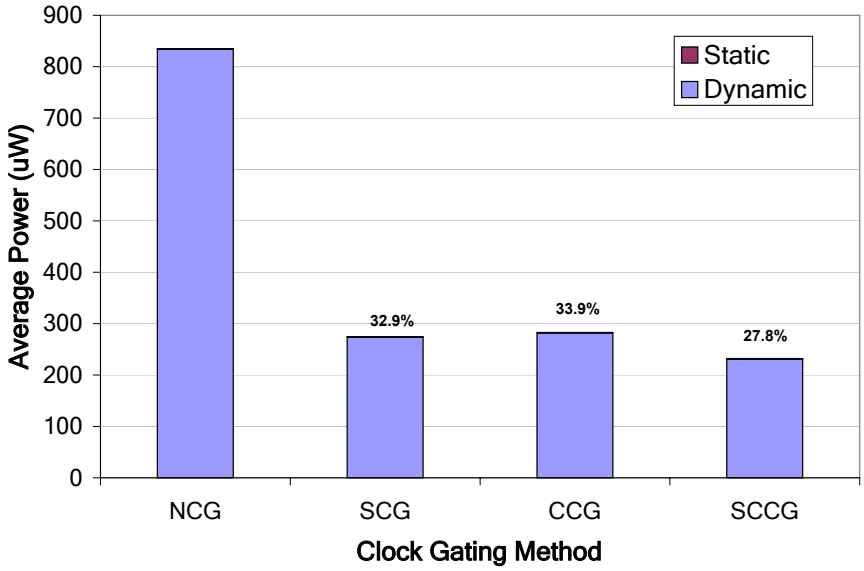


Figure 5.11: Average power - 333 MHz - Low power 90nm

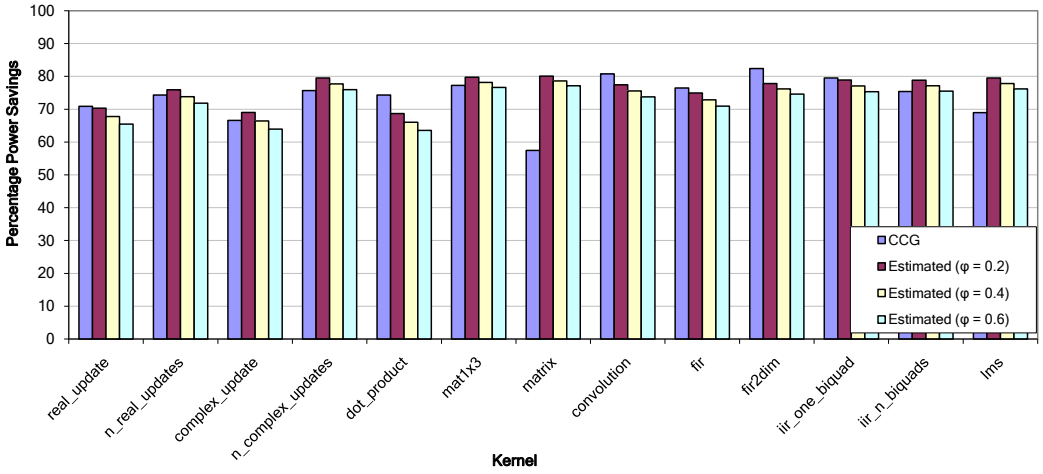


Figure 5.12: CoDeL clock gating (CCG) estimated power savings

5.4.2 Estimated Dynamic Power Savings

Figure 5.12 provides the estimated power savings obtained using statistical switching activity annotation in Synopsys Power Compiler and the estimated power savings provided by the CoDeL compiler based on the framework presented in section 5.2. All results are for the CoDeL clock gated designs (CCG) using the general purpose 90nm cell library. Since the estimation framework also implicitly uses statistical switching activity annotation, the statistical analysis performed using Synopsys provides a better comparison for our estimation framework than the simulation based analysis.

In figure 5.12, the estimated results are presented for three values of ϕ (0.2, 0.4 and 0.6), corresponding to three estimates for the average number of register bits that change value when a register changes value. We see that the overall power savings are not very sensitive to the ϕ parameter.

Comparing the estimated and obtained power savings, we see, from figure 5.12, that the savings estimated using the CoDeL based analysis framework (for $\phi = 0.2$ for example) compares quite well to the power savings using Synopsys statistical power analysis. In virtually all cases the CoDeL estimated power is within 5% of the Synopsys estimate.

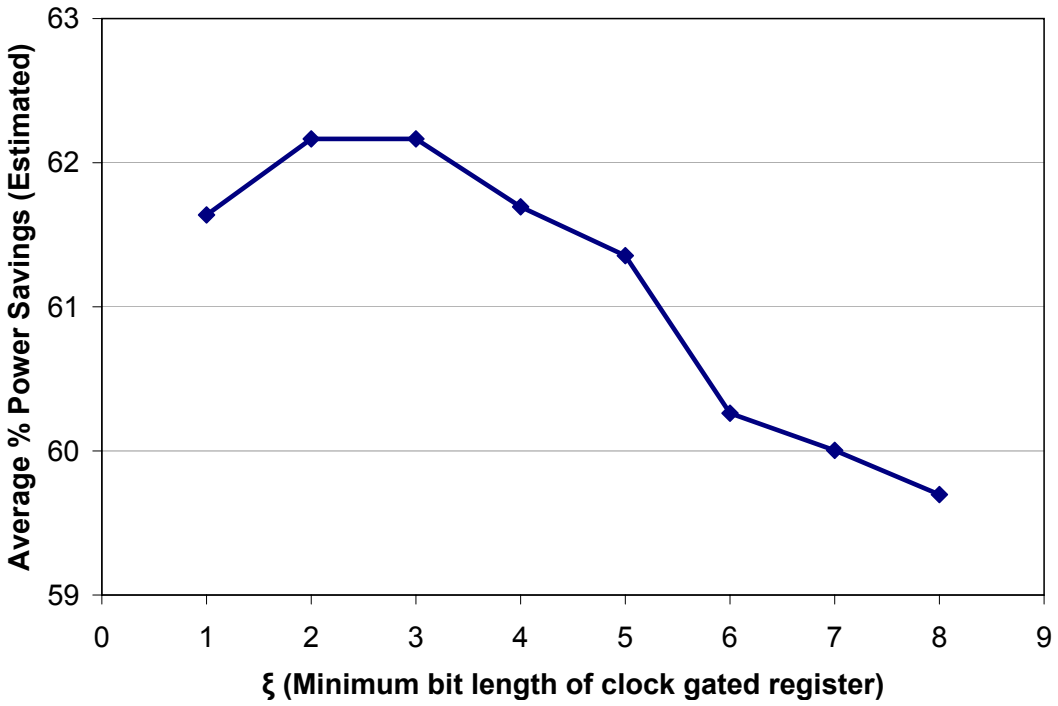


Figure 5.13: Geometric mean of the estimated power savings vs. minimum clock gated register word length (ξ)

5.4.3 Minimum Clock Gated Register Word Length (ξ)

Using the estimation framework we now try to find the optimal minimum word length for a register that should be clock gated, ξ. Figure 5.13 presents the estimated percentage power savings averaged (using a geometric mean) across the 13 kernels of the DSPstone benchmark for each value of ξ from one to eight. We can see that the power savings are maximized for $\xi = 2$ and $\xi = 3$. In fact, for every kernel, the power savings is maximized and is the same for $\xi = 2$ and $\xi = 3$. Therefore, we have used $\xi = 3$ in all our evaluations. In the design flow using CoDeL, for the particular circuit under development, it can be expected that the designer may try various values of ξ, and choose the value that maximizes the power savings estimated by the CoDeL compiler. This exploration of the optimum value for ξ may also be automated in the compiler. This would allow the development tool to automatically find the value for ξ that maximizes the power savings.

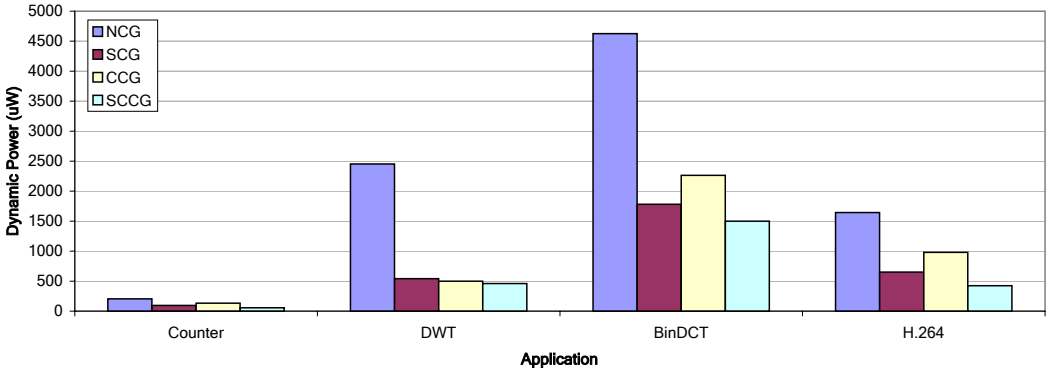


Figure 5.14: Application circuits - Power dissipation - 400 MHz - General purpose 90nm

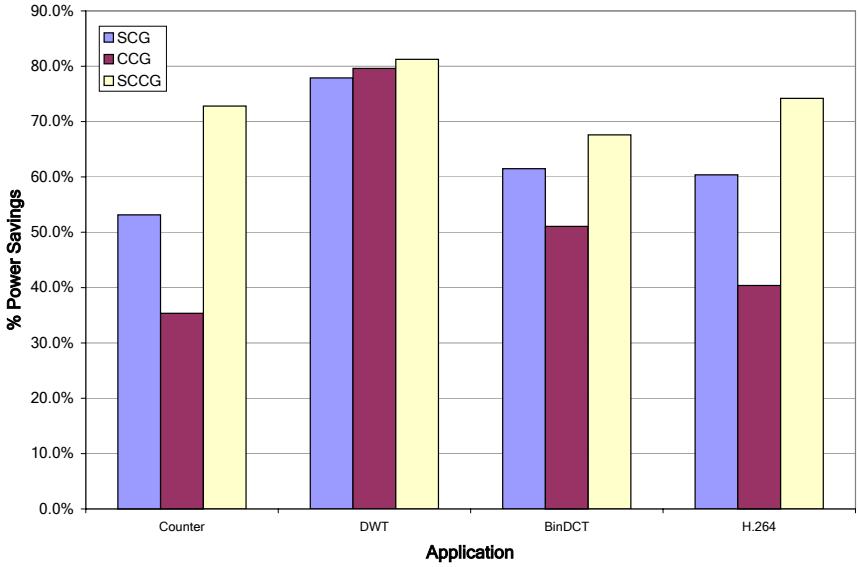


Figure 5.15: Application circuits - Power savings - 400 MHz - General purpose 90nm

5.4.4 Application Circuits Dynamic Power

Here we present dynamic power results for the four application circuits described in section 5.3: Counter, DWT, BinDCT, and H.264.

Figures 5.14 and 5.15 show the absolute power and the power savings achieved for the four circuits using the various clock gating mechanisms. As before, we find that in all cases the power savings are maximized using a combination of Synopsys and CoDeL clock gating (SCCG).

The area results are also consistent with those observed earlier for the DSPstone

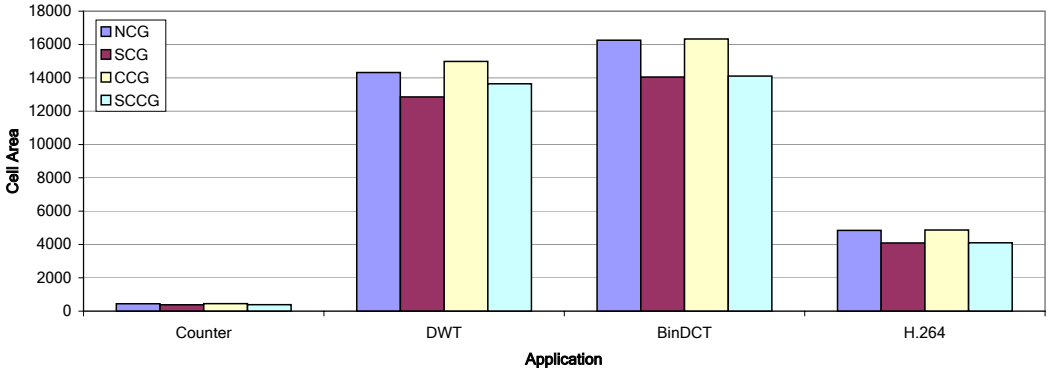


Figure 5.16: Application circuits - Cell area - 400 MHz - General purpose 90nm

kernels. The SCG circuits exhibit the lowest area while CCG exhibits the largest area usage. The SCCG circuits exhibit just slightly higher area usage than the SCG circuits.

5.5 Summary

In this chapter, we studied the effectiveness of CoDeL's automated clock gating mechanism by comparing it to the automated clock gating function provided by the Synopsys tools. We find that the use of CoDeL clock gating along with Synopsys clock gating outperforms Synopsys clock gating alone and provides 16% more power savings on average at an area overhead of 2.4%. Further, we find that the estimation framework presented provides a quick and accurate measure of the expected dynamic power savings using CoDeL based clock gating.

Chapter 6

Power Gating

Supply voltage scaling increases subthreshold leakage currents,
increases leakage power, and poses numerous challenges
in the design of special circuits.
- Shekhar Borkar [21]

Power gating of a circuit block is performed by using an appropriate header or footer transistor [41]. To begin power gating, a "sleep" signal is applied to the gate of this transistor to turn off the supply voltage to the circuit block. To revive the block for use, the "sleep" signal is de-asserted and power is restored.

In the case of memory elements, such as registers, multi-threshold CMOS (MTCMOS) [63] retention registers can be used (see figure 6.1). During normal operation, there is no loss in performance and during power-down mode the register state is saved to a "balloon" latch, which has a high voltage threshold resulting in minimal leakage. Using a MTCMOS register, all reads can be performed from the balloon latch. It is only when a write is necessary that we need to power up the high-performance low-threshold flip-flop.

In a power gating environment, the process of deactivating or activating circuit components is not instant, such as in clock gating. In clock gating the circuit state can be switched immediately by enabling the clock or enable signal [35]. In power

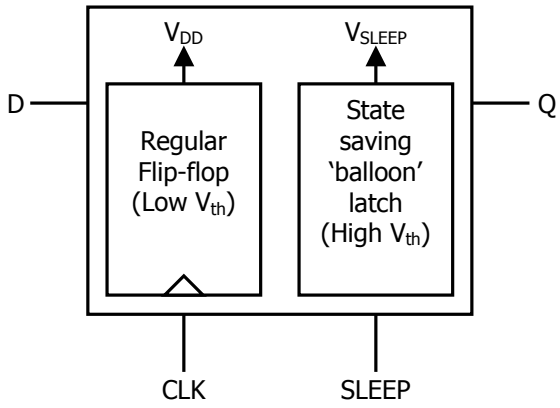


Figure 6.1: MTCMOS register

gating, however, deactivating or activating the circuit block involves the discharging or charging of capacitances, which can be time-consuming. In particular, the delay in activation poses a problem as it can lead to system stalls. To handle this delay, two methods are generally employed. First, while a sub-circuit is being restored, idle waiting cycles are inserted into the system until the sub-circuit is fully activated. Alternatively, to reduce delay overhead, the sub-circuit can be activated ahead of time prior to its usage. However, this causes additional power dissipation. To make the recovery process more efficient a branch prediction scheme can be used to reduce the cases where a sub-circuit is activated in anticipation but not used.

In figure 6.2 we present the supply voltage and the various phases of a circuit component as it is power gated. Our model here follows the description presented in [41]. From time T_0 to T_1 the circuit component is busy and thus can not be gated. This period is T_{busy}. At time T_1, the component becomes idle. It takes the control logic from T_1 to T_2 ($T_{idledetect}$) to make the decision to engage gating. From T_2 the supply voltage begins to drop. At T_3 the aggregate leakage power savings equals the overhead of switching the header transistor on and off. The period, $T_{breakeven}$, from T_2 to T_3, is the minimum power gating duration to achieve net leakage power savings. During the period T_{sleep}, from T_3 to T_4 the device is asleep and we accumulate net power savings. At T_4 the control logic needs to reactivate the component. From T_4

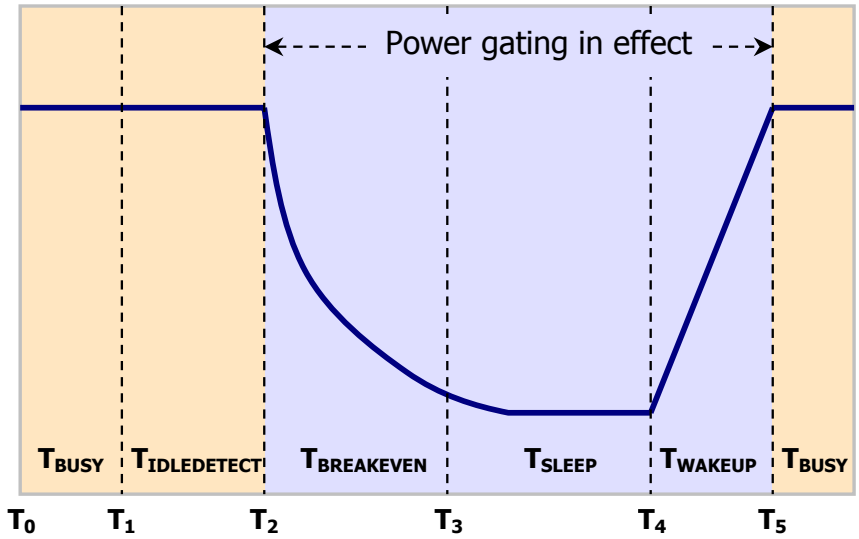


Figure 6.2: Voltage during power gating phases

to T_5 the voltage rises. During this period, T_{wakeup}, a performance penalty may be incurred if the pending operation needs to wait for the power to be restored. Finally, at T_5 the power is fully restored and the circuit can resume normal operation.

6.1 Gating Methods

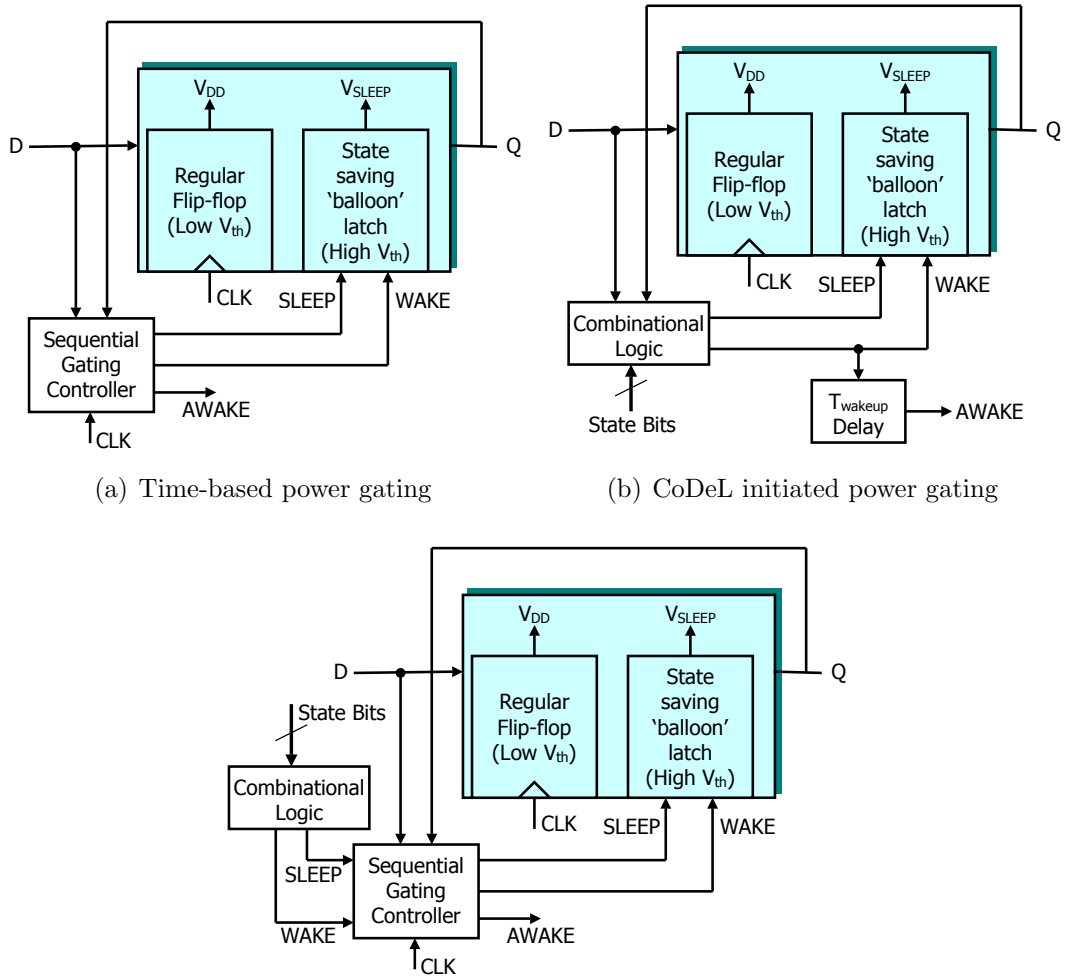
Here, we explore the gating potential of registers using three gating methods. We first study the power savings expected using a dynamic time-based technique, where gating is performed after observing a pre-determined number of idle cycles. Second, we use CoDeL to statically predict (at compile time) when the idle periods occur and appropriately power gate the registers. Third, the dynamic time-based technique is augmented by static gating predictions made by the CoDeL compiler.

6.1.1 Time-Based Power Gating

A simple technique to power gate circuit components is to dynamically observe their state and initiate power gating when a sufficient number of idle cycles are detected. Techniques such as this have been used for cache memories [30] and show significant leakage savings with minimal performance impact.

To implement this technique, each circuit component needs to have state machine logic similar to the one shown in figure 6.4. Normally the component is in the IDLE_DETECT or BUSY state. As long as the component is being used, the state remains BUSY. Once the component becomes idle we enter the IDLE_DETECT state. When the consecutive idle cycle count increases beyond $T_{idledetect}$, the component enters the POWER_DOWN state. Here it waits for period $T_{breakeven}$ to allow for the voltage supply to reduce. If at any time the component is needed, a signal is generated causing the component to enter the WAKEUP state. Otherwise, after $T_{breakeven}$ cycles, the SLEEP state is entered. When the circuit component is next needed, the WAKEUP state is entered where a waiting period of T_{wakeup} cycles is required to restore the supply voltage. Once the component is powered up, the BUSY state is entered. When the circuit prematurely goes from the POWER_DOWN state to the WAKEUP state, the component may not be fully powered down. Thus, for restoring the power it will not take the full T_{wakeup} cycles. However, we conservatively penalize the full T_{wakeup} cycles in this case. Further, we only consider the savings while in the SLEEP state. There may be some additional power savings in the WAKEUP state, which we conservatively do not include.

According to this framework, we see that our results are dependent on three parameters: $T_{idledetect}$, $T_{breakeven}$, and T_{wakeup}. $T_{breakeven}$ is the time it takes to overcome the energy overhead of gating a unit. T_{wakeup} is the overhead of restoring the power to a unit. The parameters $T_{breakeven}$ and T_{wakeup} are a function of the VLSI technology and thus can not be controlled by circuit design. The $T_{idledetect}$ parameter, however, can be controlled to effect the aggressiveness of the power gating mechanism. A lower $T_{idledetect}$ will result in the gating of shorter idle periods, which occur more often, resulting in potentially more gated cycles. However, gating of an increased number of idle periods will result in more cases where the circuit must be stalled for the power to be restored. These stalls will result in a greater performance loss. A large $T_{idledetect}$



(a) Time-based power gating

(b) CoDeL initiated power gating

(c) CoDeL assisted time-based power gating

Figure 6.3: Architectures for power-gating methods used for evaluation

will find fewer gating opportunities, and thus result in better performance.

To implement this scheme each register would require a controller to count the idle cycles, and logic to detect a new value being written to the register (see figure 6.3(a)). This logic is expensive in terms of area and power, and therefore motivates an alternative method of initiating power gating.

6.1.2 CoDeL Initiated Power Gating

The CoDeL platform [5] uses a sequential machine to determine the sequence of operations and data transfers in and out of registers. Because of this sequential

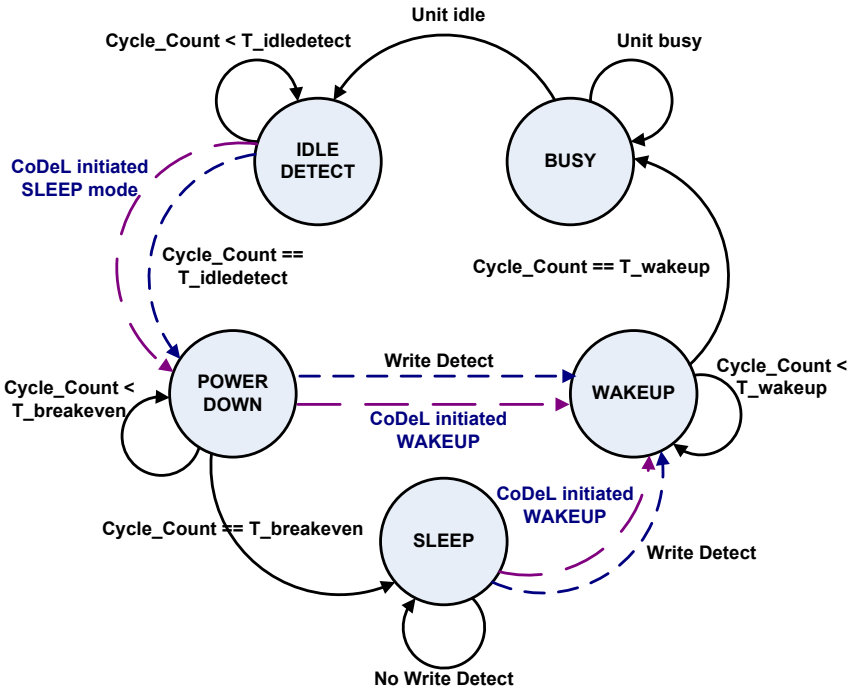


Figure 6.4: Gating logic. The short dashed line is used for time-based gating. The long dashed transition line is used for CoDeL initiated power gating. Both dashed lines are used for CoDeL assisted time-based gating.

machine, we know the exact time of the events, and we can anticipate them. For each register, at compile time, CoDeL iterates through each state of the state machine implementation of the circuit and looks ahead $T_{idledetect}$ states to determine if there are any potential writes to the register. If there is no write to the register in the next possible $T_{idledetect}$ states, a power off (SLEEP) suggestion is noted for the gating control logic. If during the next $T_{idledetect}$ possible states the register is written, a power off suggestion is not made. As with the time-based technique, the $T_{idledetect}$ parameter is chosen a priori, and is the same for all registers of the circuit under design.

To more efficiently wake up the registers, CoDeL performs a look ahead and prematurely powers up the register in anticipation of a write. This reduces the performance penalty normally incurred in waiting for a power up. For each register, at compile time, CoDeL examines each state of the state machine implementation

of the circuit and looks ahead T_{wakeup} states to determine if there are any potential writes to the register. If there is a write to the register in the next possible T_{wakeup} states, a power on (WAKE) suggestion is noted. Otherwise, a power on suggestion is not made. For example, referring to figure 6.5(a), for $T_{idledetect} = 3$ and $T_{wakeup} = 1$, a sleep suggestion will be generated at state S2, while a WAKE suggestion will be generated at state S10.

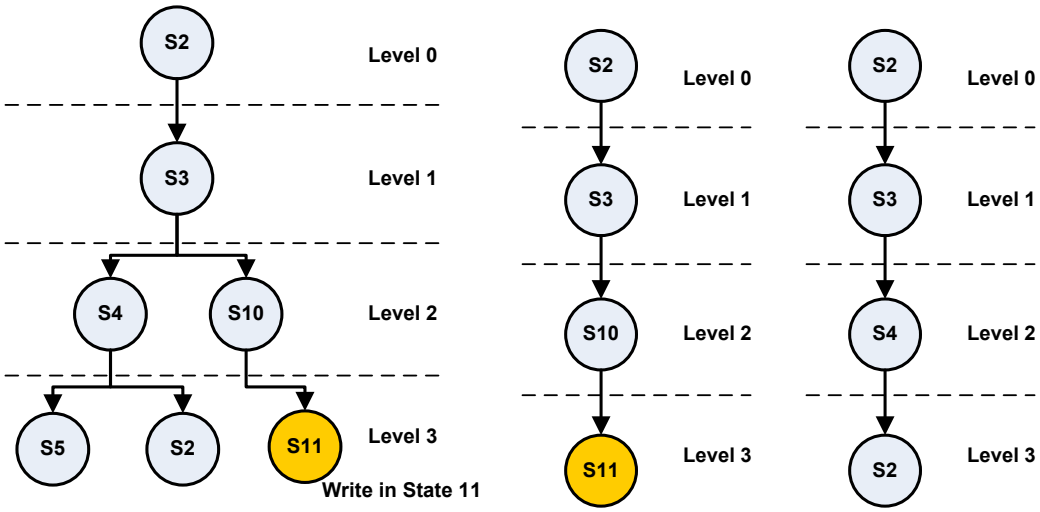
CoDeL initiated gating corresponds to a static environment where only suggestions made by CoDeL can initiate power gating (long dashed line in figure 6.4). The wakeup mechanism is triggered by a CoDeL suggestion or a detected write.

To implement this static gating scheme only combinational logic is needed. The current state is used to generate the desired SLEEP and WAKE signals to power down and power up the register. Some sequential logic may be needed to generate the `AWAKE` signal, which indicates that the register is powered up and ready for use. This allows CoDeL to stall the register write until the AWAKE signal is asserted. Figure 6.3(b) provides a possible implementation of this mechanism.

6.1.3 CoDeL Assisted Time-Based Power Gating

In CoDeL assisted time-based gating, the decision to initiate gating is still dependent on a streak of idle cycles as in the time-based technique (short dashed line in figure 6.4). In many cases, however, based on CoDeL's suggestion (long dashed line in figure 6.4), gating can be initiated prematurely without waiting for the full $T_{idledetect}$ cycles. The value of $T_{idledetect}$ used for the CoDeL and time-based parts is the same. Also, based on CoDeL's suggestion, wakeups are initiated in anticipation of a register write to reduce the performance penalty.

The implementation of this gating scheme is the most complex as it requires the circuit features of the static and dynamic gating methods. Figure 6.3(c) shows that this implementation would require features from both the time-based and the CoDeL initiated power gating mechanisms.



(a) No branch prediction (b) Forward (c) Backward

Figure 6.5: States look-ahead to determine possible writes. $T_{idledetect} = 3$

6.2 FSM Branch Prediction

CoDeL's gating and wakeup suggestions are dependent on a look-ahead search of the FSM description of the circuit to determine whether a register write is performed in the next $T_{idledetect}$ or T_{wakeup} possible states. In performing this search, branches in the state machine are handled in three different ways. The first method uses no branch prediction (figure 6.5(a)), and therefore searches all possible state paths. The second method uses static forward branch prediction and assumes that a branch to the furthest state forward is taken (figure 6.5(b)). The third method uses static backward branch prediction and assumes that a branch to the furthest state backward is taken (figure 6.5(c)).

We have only experimented with simple, purely static branch prediction schemes here, that can be automated at compile time, and do not require any area overhead in the circuit. More advanced, dynamic branch prediction mechanisms may be used and may provide somewhat better results, but we believe that the added complexity will not result in significant improvements. A further analysis of branch prediction,

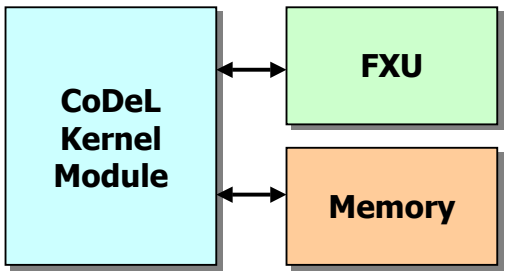


Figure 6.6: Benchmark Architecture

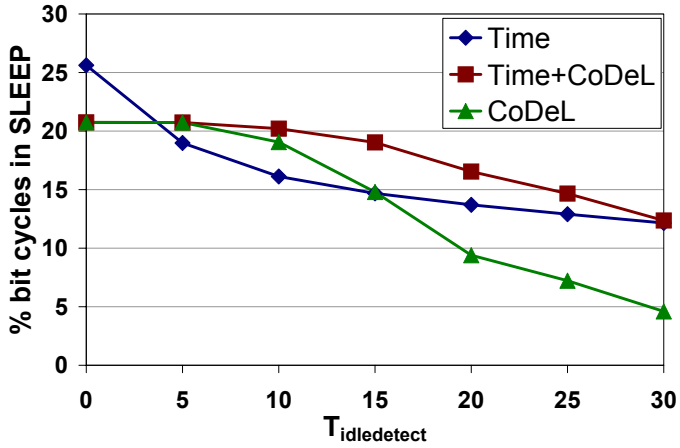
using a perfect branch predictor, would help in determining the maximum possible power savings and the minimum possible performance loss.

6.3 Evaluation Framework

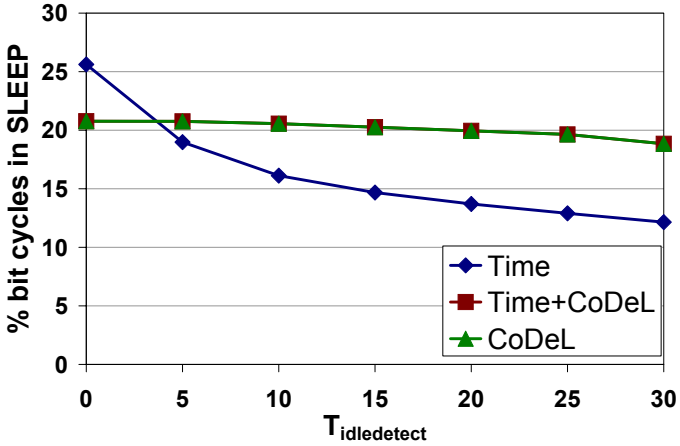
To evaluate the power gating methods we use the the DSP kernel benchmarks from the DSPstone suite [86]. These benchmarks consist of 13 code fragments which are commonly used in DSP algorithms. Table 4.1 provides a list of the code fragments in this benchmark kernel suite. All kernels from the suite are implemented using CoDeL and compiled to generate synthesizable VHDL. To perform the required arithmetic operations, we have used a single cycle 16 bit fixed point unit (FXU) written in VHDL using the fixed point package obtained from [22]. It is interfaced by the CoDeL implemented kernels to perform the required arithmetic operations. For data storage, a single port memory is implemented in VHDL for simulation. Any registers in the FXU or the memory are not gated. All clock cycle results presented are based on trace data obtained from VHDL simulation of the kernel circuits. The overall benchmark architecture is presented in figure 6.6.

6.4 Results

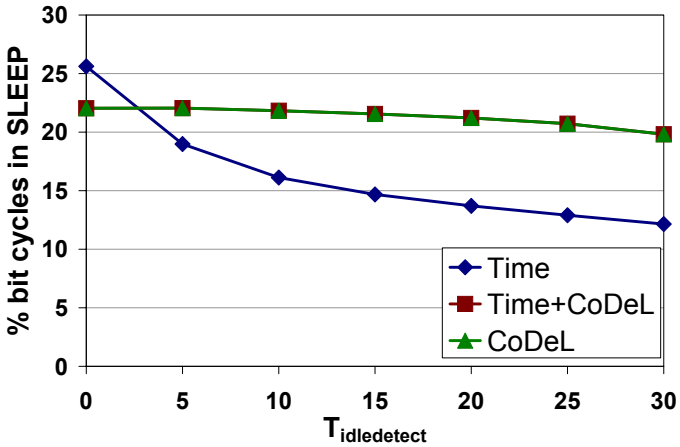
We examine the effects of $T_{idledetect}$, $T_{breakeven}$ and T_{wakeup} on the power gating ability and performance of the circuit. Specifically we look at the percentage of cycles that a register spends in the SLEEP state and weight the saved clock cycles by the width



(a) No branch prediction. Complete state path exploration.

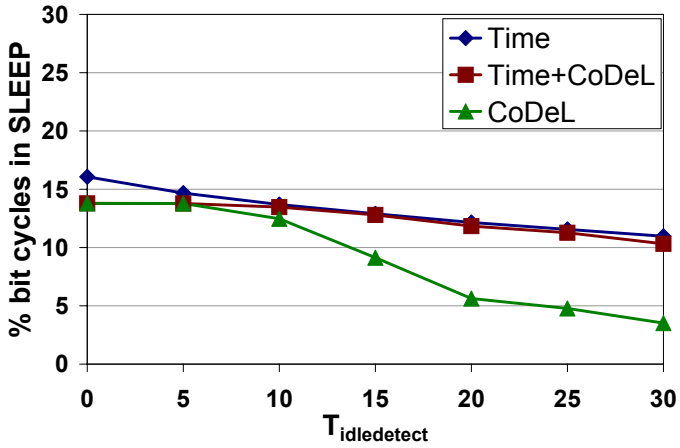


(b) Forward prediction.

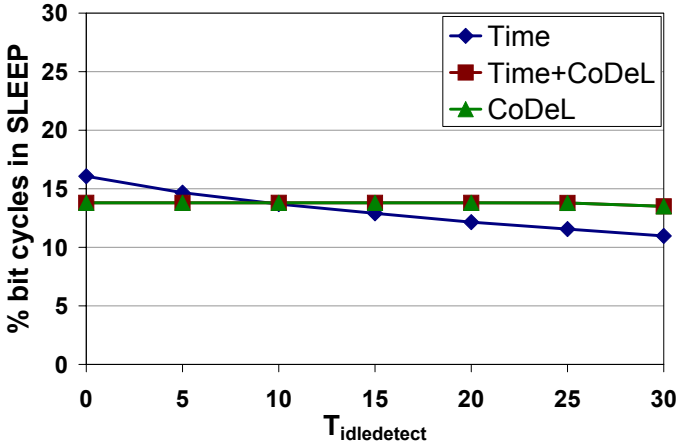


(c) Backward prediction.

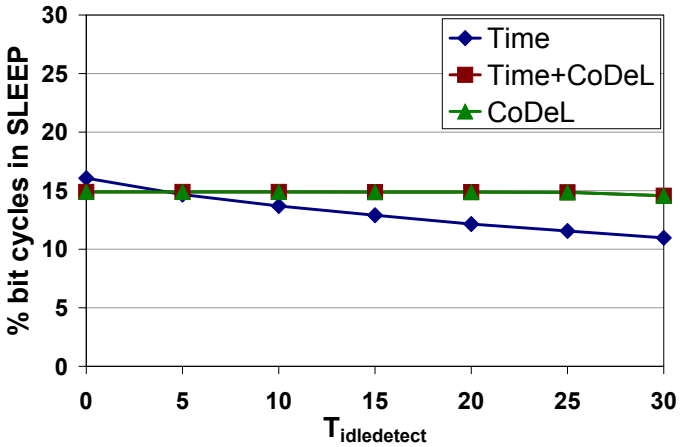
Figure 6.7: Gating effectiveness with $T_{wakeup} = 2$ and $T_{breakeven} = 10$



(a) No branch prediction. Complete state path exploration.



(b) Forward prediction.



(c) Backward prediction.

Figure 6.8: Gating effectiveness with $T_{wakeup} = 2$ and $T_{breakeven} = 20$

of the register. We call this the percentage of bit cycles in the SLEEP state. It is computed as

$$\frac{\sum_{i=0}^{N} |R_i| \cdot (\text{cycles in SLEEP state})_i}{\text{total cycles executed} \cdot \sum_{i=0}^{N} |R_i|} \cdot 100\%, \tag{6.1}$$

where N is the number of registers, R_i is the ith register and $|R_i|$ is the word length of the ith register. The performance impact of gating is computed as the number of additional clock cycles needed when power gating is introduced. This is computed as

$$\frac{\text{total cycles executed without gating}}{\text{total cycles executed with gating}} \cdot 100\%. \tag{6.2}$$

The results presented here are an arithmetic average of the results obtained for the 13 DSPstone kernels.

6.4.1 Gating Effectiveness

Figure 6.7 presents the gating effectiveness using the three methods presented and the different branch prediction schemes using $T_{breakeven} = 10$. From figure 6.7(a) we see that when no branch prediction is used, the CoDeL based gating schemes perform poorly for larger values of $T_{idledetect}$. This is because since all possible branches are searched to find a write, many more writes are predicted than actually occurring resulting in missed gating opportunities. This is exacerbated in the case of only CoDeL initiated gating, since there is no help from time-based gating to reclaim some of the lost gating opportunities.

Examining the branch prediction schemes (figures 6.7(b) and 6.7(c)) we see that the CoDeL based and the CoDeL assisted time-based schemes significantly outperform the time-based technique. For $T_{idledetect} = 30$, the CoDeL schemes provide 63% more gated bit cycles than the time-based technique. It is interesting to note that for $T_{idledetect} \geq 5$, both CoDeL based schemes exhibit the same savings. This means that the dynamic decision criteria in the CoDeL assisted time-based technique presents

no new gating opportunities in comparison to the purely static CoDeL scheme. This is an important result and suggests that the savings from using a dynamic method are small and do not justify the area and power overhead associated with using a dynamic scheme.

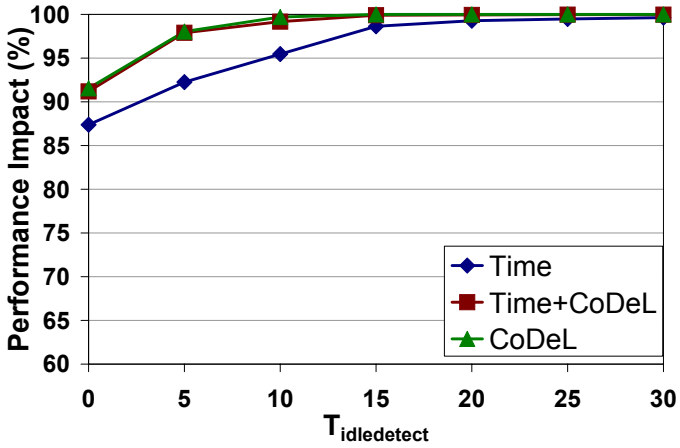
Comparing the forward and backward branch prediction schemes we find that the backward prediction results in more gating opportunities resulting in 15% more gated bit cycles than forward branch prediction. This means that more backward branches are taken in our designs than the forward branches.

Figure 6.8 presents the gating effectiveness using $T_{breakeven} = 20$. Comparing to figure 6.7 we find that a higher $T_{breakeven}$ reduces the overall gating effectiveness as expected. However, the general trends remain the same. Further, we find that with the higher $T_{breakeven}$ of 20 the CoDeL based schemes now perform better than the time-based technique for $T_{idledetect} \geq 10$. Therefore the suggested $T_{idledetect}$ value to be used is half the $T_{breakeven}$ parameter.

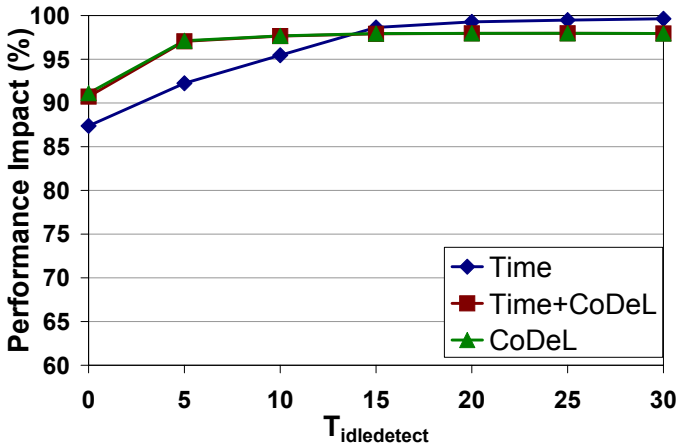
6.4.2 Performance Impact

In figure 6.9 we examine the effect of the various methods on the performance using $T_{wakeup} = 2$. It can be seen that using full state space exploration (no branch prediction) gives the best performance. This is because a full state exploration reduces the chances of misprediction reducing the number of stall cycles. As before, comparing the two CoDeL schemes we see they provide roughly the same performance. It should be noted that the value of $T_{breakeven}$ has no effect on the performance since it does not effect the number of penalty cycles encountered.

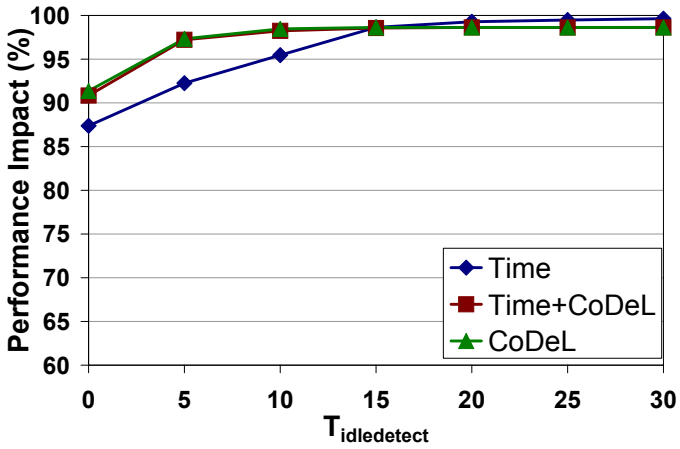
Figure 6.10 shows the performance impact for different values of T_{wakeup}. Backward branch prediction is used here for the CoDeL schemes since it provides the best gating potential as compared to forward and no branch prediction. Expectedly, as the value of T_{wakeup} increases, performance decreases as more cycles are spent in waiting for the power to be restored.



(a) No branch prediction. Complete state path exploration.

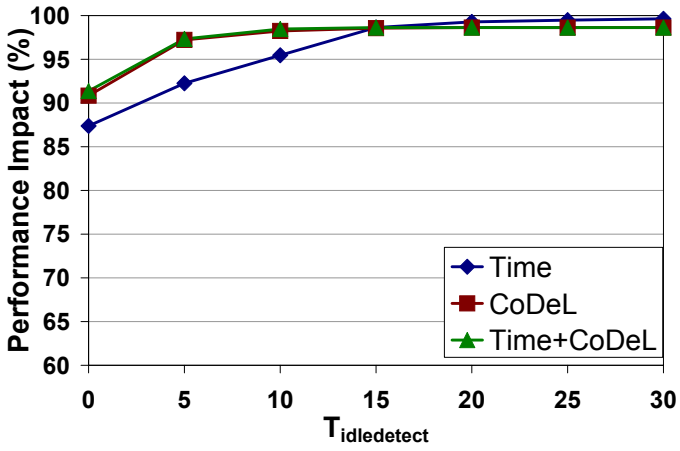


(b) Forward prediction.

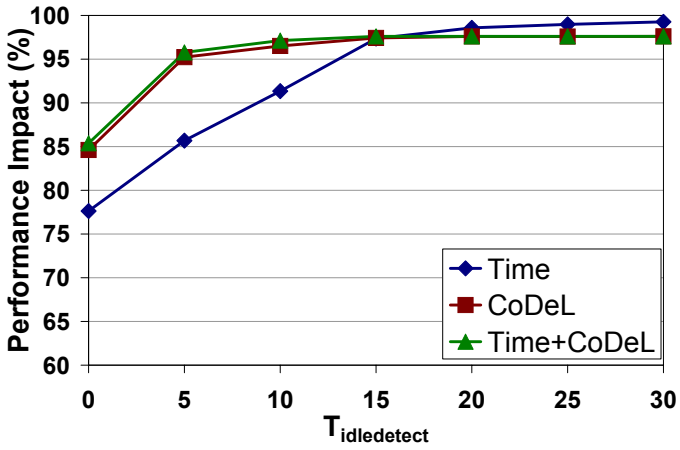


(c) Backward prediction.

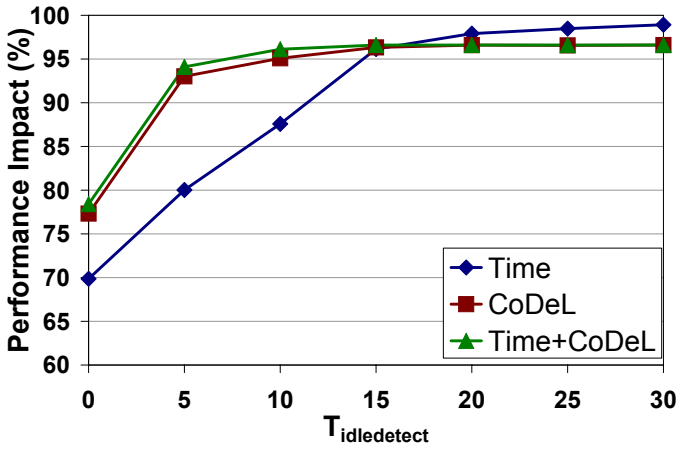
Figure 6.9: Performance impact with $T_{wakeup} = 2$ and $T_{breakeven} = 10$



(a) $T_{wakeup} = 2$



(b) $T_{wakeup} = 4$



(c) $T_{wakeup} = 6$

Figure 6.10: Performance impact with backward branch prediction

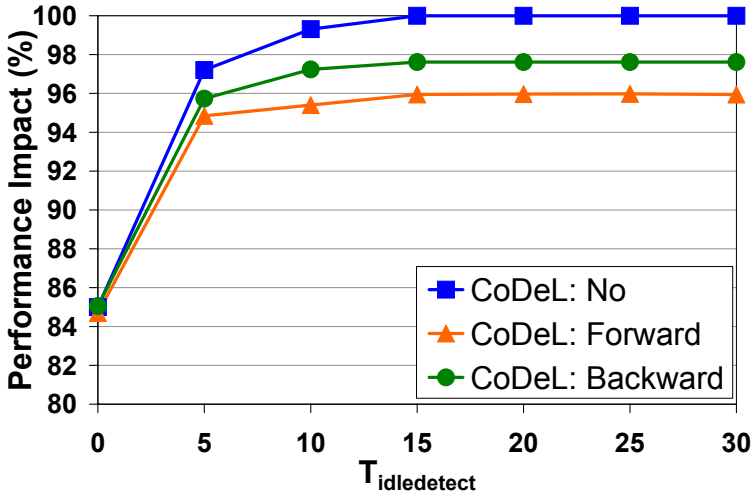
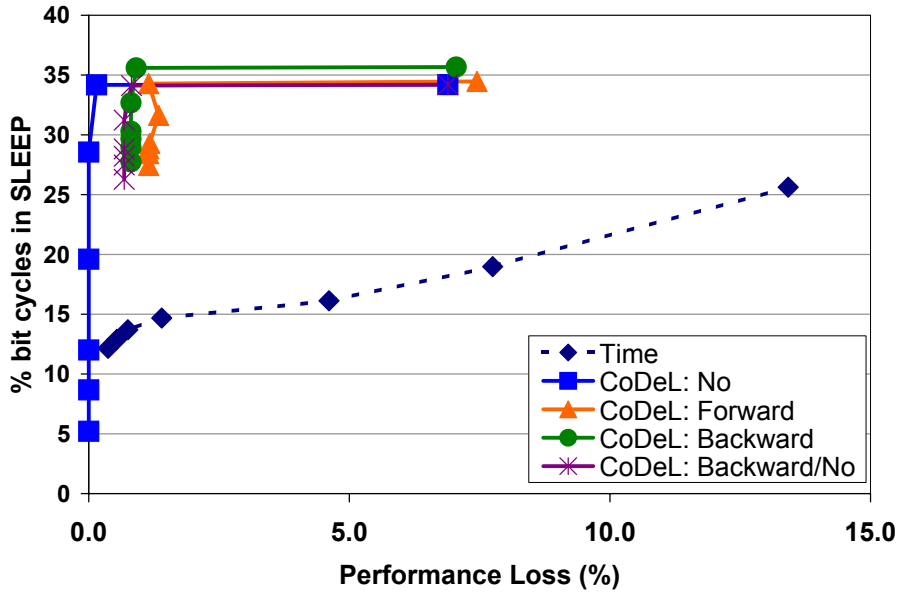


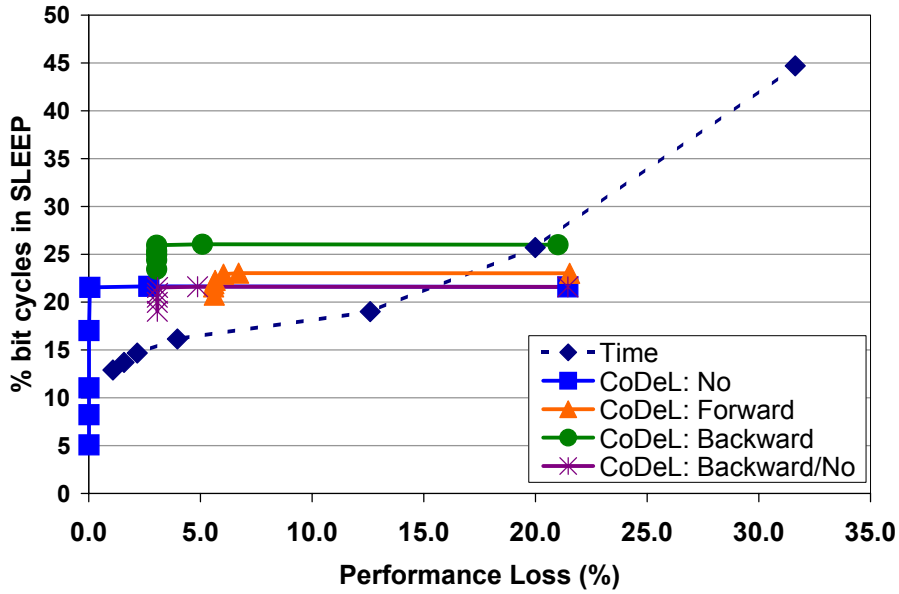
Figure 6.11: Branch prediction performance impact. Each curve represents values averaged across $T_{wakeup} = 2, 4, 6$.

In almost all cases of figures 6.9 and 6.10, we see that the CoDeL schemes outperform the time-based technique for lower values of $T_{idledetect}$ (less than 15), while for larger $T_{idledetect}$, the time-based technique dominates. This is because the time-based technique gates registers less frequently for larger $T_{idledetect}$, and thus results in fewer wakeup procedures resulting in lower performance loss. Even for these larger $T_{idledetect}$ values, however, the difference in performance for the time-based and CoDeL schemes is very small (less than 3%). But the number of sleep cycles gained with the CoDeL method far exceeds those of the time-based method by more than 60%.

In figure 6.11 we see the performance results for the various CoDeL based branch prediction methods averaged across several T_{wakeup} values. We see that the CoDeL scheme with no branch prediction performs more than 2% better than the other schemes, but it also results in the worst gating potential. Also, the backward branch prediction provides better performance than the forward branch prediction by 2% since it is more accurately able to predict wakeups.

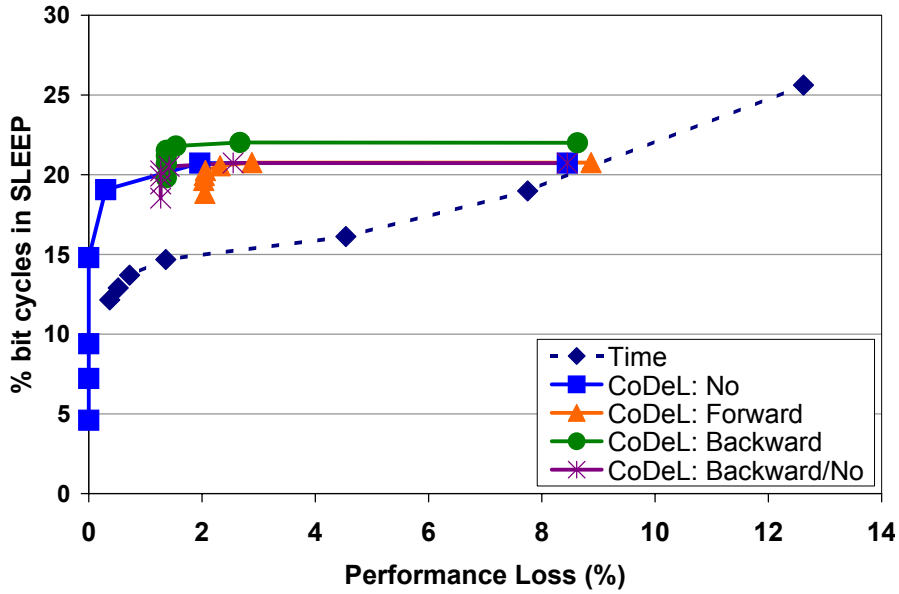


(a) $T_{breakeven} = 5$ and $T_{wakeup} = 2$

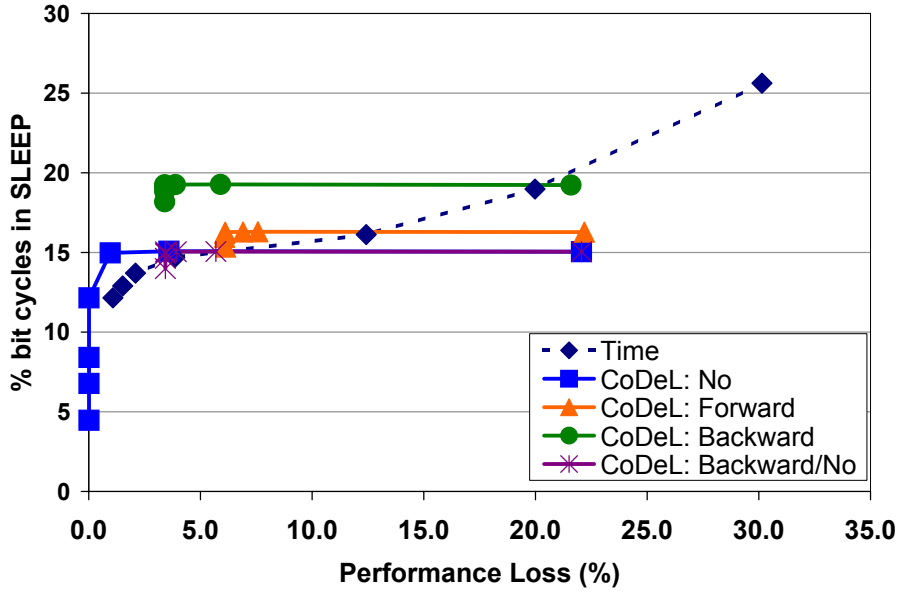


(b) $T_{breakeven} = 5$ and $T_{wakeup} = 6$

Figure 6.12: Gating effectiveness vs performance loss ($T_{breakeven} = 5$). $T_{idledetect}$ varies from 30 to 0 from left to right.

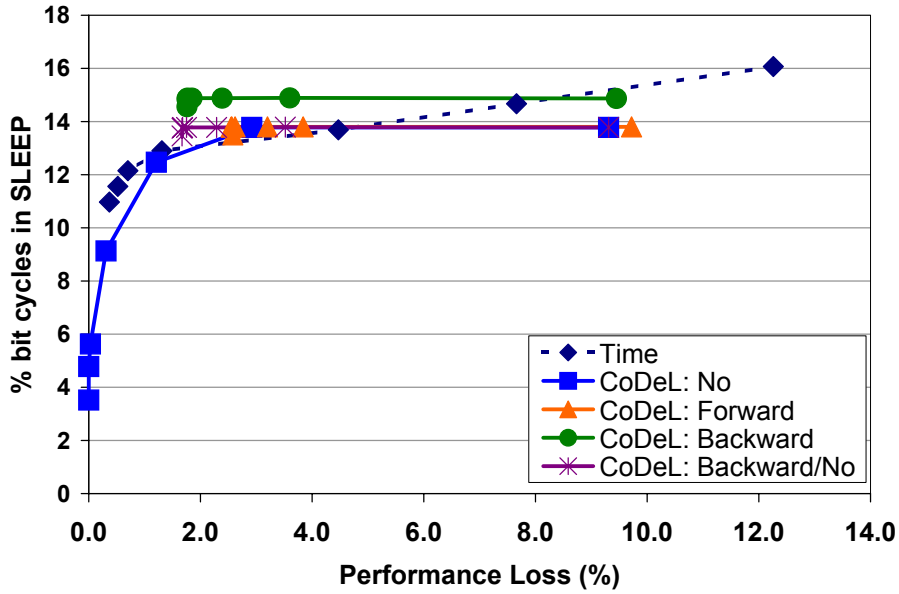


(a) $T_{breakeven} = 10$ and $T_{wakeup} = 2$

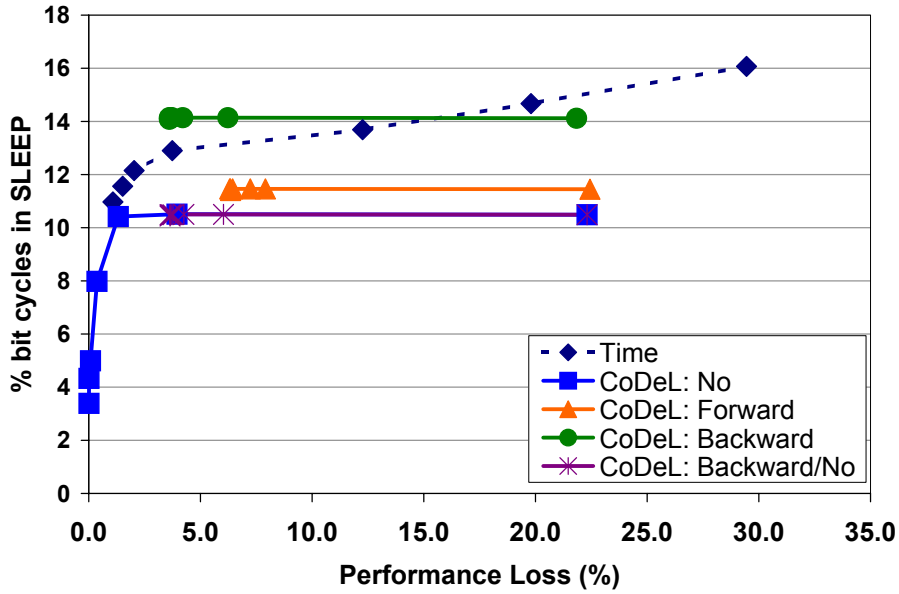


(b) $T_{breakeven} = 10$ and $T_{wakeup} = 6$

Figure 6.13: Gating effectiveness vs performance loss ($T_{breakeven} = 10$). $T_{idledetect}$ varies from 30 to 0 from left to right.



(a) $T_{breakeven} = 20$ and $T_{wakeup} = 2$



(b) $T_{breakeven} = 20$ and $T_{wakeup} = 6$

Figure 6.14: Gating effectiveness vs performance loss ($T_{breakeven} = 20$). $T_{idledetect}$ varies from 30 to 0 from left to right.

6.4.3 Results Summary

From the earlier results we find that the static CoDeL scheme with backward branch prediction provides the best gating effectiveness, while no branch prediction results in the fewest penalty cycles at wakeup resulting in the best performance. In figures 6.12, 6.13 and 6.14 we are able to more clearly see the entire design space consisting of the various techniques. This figure allows the designer to choose the right gating method and $T_{idledetect}$ parameter based on the technology parameters $T_{breakeven}$ and T_{wakeup}, and the desired gating effectiveness and the tolerable performance impact.

In figures 6.12, 6.13 and 6.14 we have also included a variation which uses backward branch prediction to predict SLEEP conditions while no branch prediction is used for WAKEUP conditions. This is labeled 'CoDeL: Backward/No'. The solid curves indicate results employing a static gating method where the area overhead is extremely low. A dashed curve is used for the time-based method which employs a dynamic scheme resulting in significant overhead.

The subfigures of figures 6.12, 6.13 and 6.14 provide the design space for various possible values of $T_{breakeven}$ and T_{wakeup}. Common in all figures we see that lower values of $T_{idledetect}$ cause significant performance loss. This means that although there are a large number of short idle periods which can benefit from gating, the performance degrades since this causes a large increase in the number of cases where the circuit needs to wait for a power up to occur. The design points where the performance loss exceeds 5% are impractical since in most situations a performance degradation that large may not be desired. Further, the incremental improvement in gating effectiveness beyond this point is not significant. Based on this observation, a $T_{idledetect}$ value of greater than or equal to 10 is recommended.

Overall, we find that CoDeL, with backward branch prediction, is able to provide the best compromise of high gating effectiveness and low performance loss. CoDeL with forward branch prediction provides a relatively poor combination of gating ef-

fectiveness and performance. This is due to the high rate of misprediction with this method. CoDeL with no branch prediction provides lower gating effectiveness but provides excellent performance for larger values of $T_{idledetect}$, and thus may be useful in cases where high performance is critical.

Examining the 'CoDeL: Backward/No' case where backward branch prediction is used for SLEEP prediction and no branch prediction is used for WAKEUP, we find that the overall effect is not beneficial. Using this method, we observe a slightly lower performance loss than that of the case employing backward branch prediction for both SLEEP and WAKEUP and see that the gating effectiveness is reduced. This means that the low performance loss we find using the case where no branch prediction is used for SLEEP and WAKEUP is due to the reduced number of SLEEP modes that are initiated resulting in lower WAKEUP penalties.

For lower values of $T_{breakeven}$ we see that the time-based technique provides poor overall gating effectiveness and performance. However, for the cases where $T_{breakeven}$ is large, the time-based scheme is competitive with the static CoDeL based schemes.

In figure 6.13(a) we examine the specific case where $T_{breakeven}$ is 10 and T_{wakeup} is 2. For $T_{idledetect} = 15$ we find that the CoDeL scheme with backward branch prediction provides 47% more bit cycles in SLEEP mode than the time-based technique for the same approximate performance loss of 1.4%.

6.5 Summary

Test circuits, implemented using the CoDeL design platform, were examined to determine the expected savings that can be achieved from power gating individual registers, and the associated performance impact. It was found that a CoDeL initiated power gating scheme with static backward branch prediction provides an overall superior combination of high gating effectiveness and low performance loss. For high performance applications, CoDeL with no branch prediction (full state space explo-

ration) is the best choice. In both these methods, since the gating decisions are made at compile time, there is very little circuit area overhead.

Here, we have introduced a methodology for implementing efficient power gating using the CoDeL platform. Using the ideas presented we hope to enhance the CoDeL design environment and fully automate the process of power gating in VLSI circuits.

Chapter 7

FSMD Partitioning

Nothing is particularly hard if you divide it into small jobs.

- Henry Ford

The proposed partitioning techniques work at the behavioral level, before synthesis. The FSMD described at the behavioral level is split into two or more separate FSMD units. At any given time, only one FSMD is active while the others are powered off or clock gated. This results in significant power savings (static and dynamic). A review of partitioning techniques for power reduction is presented earlier in section 3.5.

Ideally, data components must be isolated as much as possible so they can be turned off as long as possible. Further, when data components are shared between FSMDs, their updated values need to be communicated to the newly activated FSMD. This communication overhead results in power dissipation that should be minimized. A high level architecture of the proposed FSMD partitioning is presented in figure 7.1.

Another important criterion is to minimize the number of transitions between partitions. Each time a transition occurs we not only have the communication penalty, but also encounter a startup delay whereby the capacitances of the newly activated FSMD are charged up. To reduce this performance penalty a lookahead mechanism

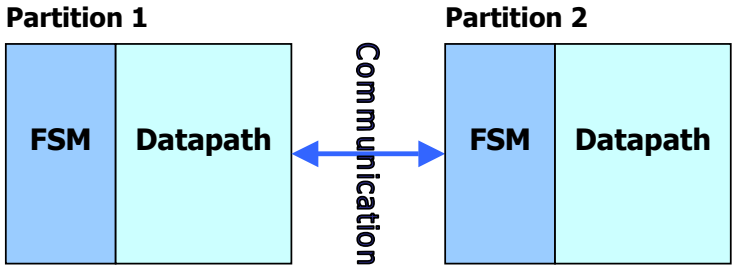


Figure 7.1: Partitioned FSMD

can be used, which is described briefly in section 7.3.

To minimize these adverse effects we need to efficiently partition the FSMD to reduce the amount of shared data components. To achieve such an efficient partition we formulate FSMD partitioning first as an Integer Linear Programming (ILP) problem, and second as a non-linear programming problem which we solve using the Simulated Annealing (SA) algorithm. In both cases, our objective is to minimize the number of shared components between partitions and also minimize the number of possible transitions between the partitions.

7.1 Problem Formulation

7.1.1 Preliminaries

Formally, a finite state machine with datapath (FSMD), P, is a 6-tuple defined as (see [42])

$$P = \langle S, s_1, I \cup STAT, O \cup A, \delta, \lambda \rangle \tag{7.1}$$

where:

- $S = \{s_1, ..., s_N\}$ is the set of finite states.
- s_1 is the reset state.
- I is the set of input variables.

- $STAT = \{Rel(a, b) : a, b \in EXP\}$ is the set of statements specifying relations between two expressions from the set EXP.
- $EXP = \{f(x, y, z, ...) : x, y, z, ... \in VAR\}$ is the set of expressions.
- VAR is the set of storage variables.
- $O = \{o_k\}$ is the set of primary output values.
- $A = \{x \Leftarrow e : x \in VAR, e \in EXP\}$ is the set of storage assignments.
- $\delta : S \times (I \cup STAT) \rightarrow S$ is the state transition function.
- λ is the output function where $\lambda : S \times (I \cup STAT) \rightarrow (O \cup A)$ for Mealy models, and $\lambda : S \rightarrow (O \cup A)$ for Moore models.

A FSMD can also be represented using a state transition graph (STG). The STG of P can be described as $G_P(V_P, E_P)$, where V_P is the set of nodes, representing the state set of P, S, and $E_P = \{\langle u, v \rangle, u, v \in V_P\}$ is the set of edges representing the state transition set of P.

The partitions of P, are a subset of S, along with the transitions related to the states in S. In addition, we require structures for coordination and communication between the partitions, while preserving functionality. It is our goal here to partition machine P into submachines P_k such that the interaction between these partitions is minimized. Let the number of partitions be M. The set of partitions can then be identified as $P_k \forall k \in [1, M]$.

The next two sections describe two approaches that we have investigated. The first method constructs and describes the ILP model. The second method presents a non-linear programming model which is solved using the SA algorithm.

It should be noted that the ILP model presented here was developed first [10]. It was found, however, that the runtime of the ILP approach is extremely long making

the approach highly impractical. Thus, a non-linear model was developed and solved using simulated annealing [11], which is significantly faster than the ILP method. The ILP approach is presented here for completeness.

7.1.2 The ILP Model

Given an FSMD, we first determine the set of variables that are shared between the various states. A variable, $v \in VAR$ is considered shared between states s_i and s_j if the variable is read or written in state s_i and read or written in state s_j. We refer to the bits of variable v as being *transition bits*. We, thus, represent the total number of transition bits between states s_i and s_j as T_{ij}.

We represent the set of edges of the STG as E_{ij}, which is a binary variable. It is 1 if and only if there exists an edge (transition) from state s_i to s_j.

Let s_{ij} be a binary variable which is 1 if and only if the states s_i and s_j are in the same partition.

The objective function can now be stated as

$$\min \left[\sum_{i,j=1}^{N} T_{ij} \left(1 - s_{ij}\right) + \lambda \sum_{i,j=1}^{N} E_{ij} \left(1 - s_{ij}\right) \right] \tag{7.2}$$

where N is the total number of states in a machine, P. The first summation term represents all transition bits between the various partitions, P_k, and the second summation term represents all edges between the partitions. The edges between the two partitions are weighted by

$$\lambda = \sum_{v \in VAR} |v|, \tag{7.3}$$

which is the sum of all register bits in the original partition P. This is because, in the worst case, all register bits may need to be communicated from one partition to the other.

The constraints on the ILP fall into two categories: quality constraints and correctness constraints. The quality constraints help to guide the solution toward a

useful solution, while the correctness constraints ensure the variables have consistent values.

The quality constraint for our design ensures that each partition contains at least a few of the total states from the original machine P. This creates partitions with a roughly balanced number of states, and avoids the creation of trivial partitions consisting of zero or one state. We now introduce the binary variables s_i^k for $i \in [1, N]$ and $k \in [1, M]$, which are 1 if and only if state s_i belongs to partition k. The quality constraint can now be formulated as

$$\sum_{i=1}^{N} s_i^k \geq \phi \frac{N}{M}, 0 < \phi < M, \tag{7.4}$$

where ϕ is a configurable parameter.

Through experimentation, it is found that $\phi = 0.6$ is able to effectively eliminate highly unbalanced partitions. This implies that for $M = 2$ partitions, each partition must contain at least 30% of the total number of states.

The first correctness constraint ensures that a state belongs to one and only one partition.

$$\sum_{k=1}^{M} s_i^k = 1, \ \forall i \in [1, N] \tag{7.5}$$

Second, we need to capture the fact that if states s_i and s_j are both in partition P_k, they belong to the same partition. This constraint is non-linear by nature and we need to take a few steps to linearize this. We first introduce variables, s_{ij}^k, which are 1 if and only if states s_i and s_j are both in partition k or neither is in partition k. The non-linear representation of this formulation can be captured by

$$s_{ij}^k = s_i^k \text{ XNOR } s_j^k \tag{7.6}$$

Minimize

$$\sum_{i,j=1}^{N} T_{ij}\left(1 - s_{ij}\right) + \lambda \sum_{i,j=1}^{N} E_{ij}\left(1 - s_{ij}\right)$$

Subject to

$$\sum_{i=1}^{N} s_i^k \geq \phi \, \frac{N}{M}, \ \forall k \in [1, M], 0 < \phi < M$$

$$\sum_{k=1}^{M} s_i^k = 1, \ \forall i \in [1, N]$$

$$s_i^k + s_j^k + 1 = 2c_{ij}^k + s_{ij}^k, \ \forall k \in [1, M], \forall \{i, j\} \in [1, N]$$

$$\sum_{k=1}^{M} s_{ij}^k + 2 = M + 2s_{ij}, \ \forall k \in [1, M], \forall \{i, j\} \in [1, N]$$

All variables $(s_{ij}, s_{ij}^k, s_i^k, c_{ij}^k)$ are binary.

Figure 7.2: ILP model

Equation 7.6 can be linearized as

$$s_i^k + s_j^k + 1 = 2c_{ij}^k + s_{ij}^k \tag{7.7}$$

where c_{ij}^k is also a binary variable.

We can now formulate a linearized equation for s_{ij} by observing that the sum of s_{ij}^k over k will be M if both states, s_i and s_j, are in the same partition. Otherwise, the sum will be $M - 2$. This can be represented as a linear equation as follows.

$$\sum_{k=1}^{M} s_{ij}^k + 2 = M + 2s_{ij}. \tag{7.8}$$

The ILP model presented is summarized in figure 7.2.

7.1.3 The Non-Linear Model

As before, we first determine the set of variables that are shared between the various states. A variable, $v \in VAR$, is considered shared between states s_i and s_j if the variable is read or written in state s_i and read or written in state s_j. We categorize

shared variables into two groups. The first group, which we call *duplicated* variables are those which are either read in both states s_i and s_j, or written in both states. The second group is called *transition* variables, which consists of registers that are read in state s_i and written is state s_j, or vice versa. We represent the total number of duplicated register bits between states s_i and s_j as D_{ij}, while the total number of transition bits is denoted T_{ij}.

Another important criteria in the model is to penalize the partitioning of a loop in the FSMD. To capture this, we introduce pseudo-edges between any two states s_i, s_j belonging to the same loop in the FSMD. Then the binary variable L_{ij} denotes the existence of such a pseudo-edge between states s_i and s_j ($L_{ij} = 1$). If s_i and s_j do not belong to the same loop, $L_{ij} = 0$.

We introduce the binary variables s_{ik} for all $i \in [1, N]$, which are 1 if and only if state s_i belongs to partition k. Here, $N = |S|$ is the number of states of the original machine P.

The total number of duplicated bits between partitions can be counted using the following

$$D_{total} = \sum_{i,j=1}^{N} D_{ij} \left[1 - \sum_{k=1}^{M} s_{ik} s_{jk} \right], \tag{7.9}$$

where M is the number of partitions. Also, we must adhere to the following constraint

$$\forall i \in [1, N] : \sum_{k=1}^{M} s_{ik} = 1, \tag{7.10}$$

which requires that each state s_i belong to one and only one partition k.

Equation 7.9 can be simplified to

$$D_{total} = \sum_{i,j=1}^{N} D_{ij} - \sum_{i,j=1}^{N} D_{ij} \left[\sum_{k=1}^{M} s_{ik} s_{jk} \right]. \tag{7.11}$$

The first term in equation 7.11 is constant since it represents all the shared variable bits in the original machine. Therefore it can be ignored in the optimization. We now get

$$D_{total} \approx -\sum_{i,j=1}^{N} D_{ij} \left[\sum_{k=1}^{M} s_{ik} s_{jk} \right]. \tag{7.12}$$

Let $\mathbf{D}$ be a square, NxN, symmetric matrix where each element D_{ij} is the number of duplicated register bits between states s_i and s_j. Further, let $\mathbf{s}$ be an NxM matrix where s_{ik} is 1 if and only if state s_i belongs to partition k. We can now formulate a matrix multiplication as follows.

$$\mathbf{\Theta} = \mathbf{s^T D s}, \tag{7.13}$$

which can be expanded as

$$\mathbf{\Theta} = \begin{bmatrix} s_{1,1} & \cdots & s_{N,1} \\ \vdots & \ddots & \vdots \\ s_{1,M} & \cdots & s_{N,M} \end{bmatrix} \begin{bmatrix} D_{1,1} & \cdots & D_{1,N} \\ \vdots & \ddots & \vdots \\ D_{N,1} & \cdots & D_{N,N} \end{bmatrix} \begin{bmatrix} s_{1,1} & \cdots & s_{1,M} \\ \vdots & \ddots & \vdots \\ s_{N,1} & \cdots & s_{N,M} \end{bmatrix}.$$

The first two matrices are multiplied to give the following.

$$\mathbf{\Theta} = \begin{bmatrix} \sum_{i=1}^{N} s_{i,1} D_{i,1} & \cdots & \sum_{i=1}^{N} s_{i,1} D_{i,N} \\ \vdots & \ddots & \vdots \\ \sum_{i=1}^{N} s_{i,M} D_{i,1} & \cdots & \sum_{i=1}^{N} s_{i,M} D_{i,N} \end{bmatrix} \begin{bmatrix} s_{1,1} & \cdots & s_{1,M} \\ \vdots & \ddots & \vdots \\ s_{N,1} & \cdots & s_{N,M} \end{bmatrix}.$$

Multiplying the remaining two matrices gives

$$\mathbf{\Theta} = \begin{bmatrix} \sum_{i,j=1}^{N} s_{i,1} s_{j,1} D_{i,j} & \cdots & \cdots \\ \vdots & \ddots & \vdots \\ \cdots & \cdots & \sum_{i,j=1}^{N} s_{i,M} s_{j,M} D_{i,j} \end{bmatrix}.$$

The trace of matrix $\mathbf{\Theta}$ is given by

$$\text{trace}\left(\mathbf{\Theta}\right) = \sum_{k=1}^{M}\sum_{i,j=1}^{N} s_{i,k}s_{j,k}D_{i,j}. \tag{7.14}$$

Switching the summations in equation 7.12 gives us an equation of the form specified above in equation 7.14. Using this, equation 7.12 can be written more concisely as

$$D_{total} \approx -\text{trace}\left(\mathbf{\Theta}\right),$$

or equivalently,

$$D_{total} \approx -\text{trace}\left(\mathbf{s^T D s}\right). \tag{7.15}$$

As with the total number of duplicated bits between partitions, the number of transition bits can be represented as

$$T_{total} \approx -\text{trace}\left(\mathbf{s^T T s}\right). \tag{7.16}$$

The total number of edges between all partitions can be counted as

$$E_{total} \approx -\text{trace}\left(\mathbf{s^T E s}\right). \tag{7.17}$$

The total number of pseudo-edges between partitions, due to loops, can be counted as

$$L_{total} \approx -\text{trace}\left(\mathbf{s^T L s}\right). \tag{7.18}$$

The objective function to be minimized can now be formulated as a combination of the parameters introduced earlier. It can be stated as

$$\min\left[\alpha_D D_{total} + \alpha_T T_{total} + \lambda\left(\alpha_E E_{total} + \alpha_L L_{total}\right)\right], \tag{7.19}$$

where the edges are weighted by

$$\lambda = \sum_{v \in VAR} |v|, \tag{7.20}$$

which is the sum of all register bits in the original partition P. This is because, in the worst case, all register bits may need to be communicated from one partition to the other. We have also introduced the factors, α, which allow us to adjust relative weights for the four parameters based on their relative importance in the final result. Through experimentation we have determined that $\alpha_D = 0.5$, $\alpha_T = 1$, $\alpha_E = 1$ and $\alpha_L = 2$ provide effective results. This means that we penalize heavily for breaking loops, while the minimization of duplicated variables is less important. Although we chose the values of the parameters α_D, α_T, α_E and α_L experimentally, we have not performed a complete design-space exploration nor studied the relative impact of each of these parameters on the final partitioning of the circuit. We anticipate that we will employ a Plackett-Burman methodology [67] to ascertain the impact of each of these parameters as we further explore their optimum values.

Equation 7.19 can be simplified further and can be stated as

$$\min \left[-\text{trace} \left(\mathbf{s^T} \left[\alpha_D \mathbf{D} + \alpha_T \mathbf{T} + \lambda \left(\alpha_E \mathbf{E} + \alpha_L \mathbf{L} \right) \right] \mathbf{s} \right) \right]. \tag{7.21}$$

In each iteration of the simulated annealing algorithm, the partition of a randomly chosen state is modified. We have added a quality constraint to this update procedure such that each partition contains at least a few of the total states from the original machine P. This constraint can be specified as

$$\forall k \in [1, M] : \sum_{i=1}^{N} s_{ik} \geq \phi \frac{N}{M}, 0 < \phi < M, \tag{7.22}$$

where ϕ is a configurable parameter.

```
module counter (
    in    inc,
    out countOut[8]
)
{
    register count[8];

    if (inc == 1)
    {
        count = count + 1;
        countOut = count;
    }
}
```

Figure 7.3: Counter CoDeL code

```
CASE state_value IS
    WHEN S0 =>
                IF inc THEN
                  state_value <= S1;
                ELSE
                  state_value <= S3;
                END IF;
    WHEN S1 =>
                  count <= count + 1;
                  state_value <= S2;
    WHEN S2 =>
                  countOut <= count;
                  state_value <= S3;
    WHEN S3 =>
                  state_value <= S0;
END CASE;
```

Figure 7.4: Counter FSMD pseudocode

As in the case of the ILP model, experimentation shows that $\phi = 0.6$ is able to effectively eliminate highly unbalanced partitions. This implies that for $M = 2$ partitions, each partition must contain at least 30% of the total number of states.

For the simulated annealing algorithm, the cooling schedule used is $T_{i+1} = 0.9\ T_i$. This is experimentally found to give a good convergence profile.

7.2 Example

We present here a complete example using a simple 8 bit counter, whose FSMD is partitioned into two submachines. Using both of the proposed methods the resulting

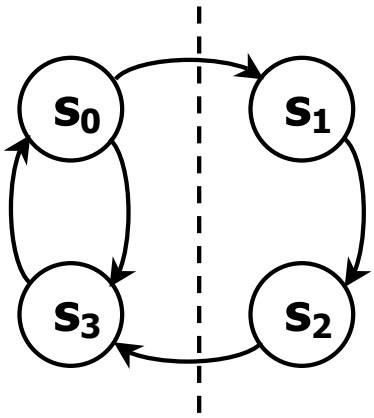


Figure 7.5: Counter STG with partition

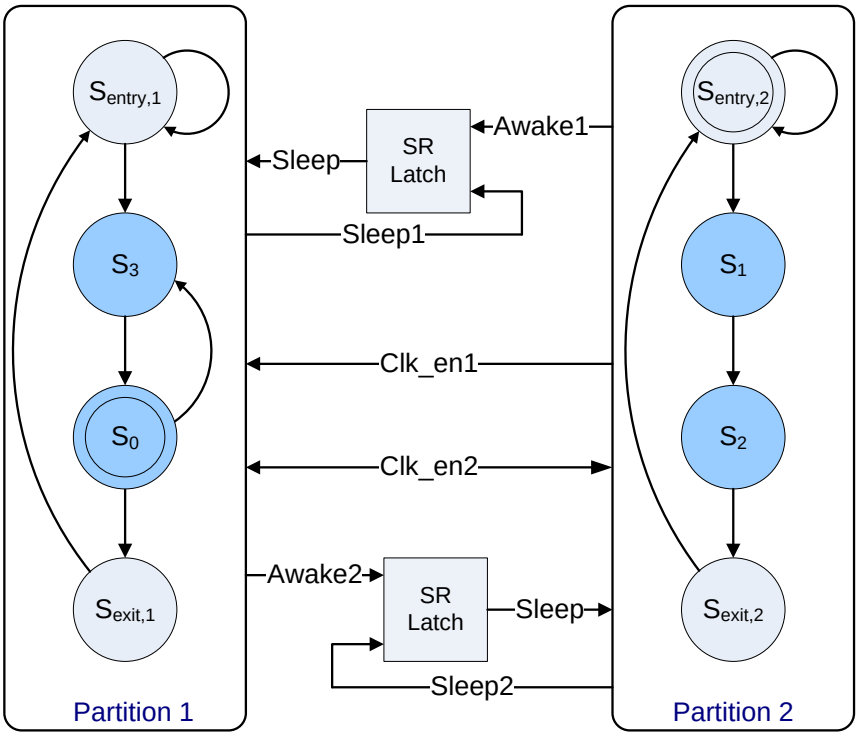


Figure 7.6: Counter STGs after partitioning

partition is the same. In figure 7.3 the CoDeL representation of the counter is presented. The CoDeL compiler produces a 4 state FSMD in VHDL. The pseudocode for the produced FSMD is presented in figure 7.4. Figure 7.5 presents the STG representation of the counter. After partitioning, we find that states 0 and 3 have been partitioned into submachine, P_1, while states 1 and 2 are in submachine, P_2 (see figure 7.6). This is ideal since the variables `count`, `countOut` and the adder are only used in states 1 and 2. This allows the register `count`, the output latch `countOut`, and the adder to be completely isolated into partition P_2. Therefore, they need not exist in P_1. As a consequence, whenever partition P_1 is active and P_2 is inactive, we save power that would normally be dissipated in a non-partitioned FSMD. Further, due to the complete isolation of the `count` and `countOut` variables, no data transfer is needed when the active partition is changed. This saves communication overhead. It should be noted that the variable `state_value` is a special register that encodes state and must exist in all partitions.

In figure 7.6 we present a detailed view of the two partitions of the partitioned counter FSMD. We see the addition of the entry and exit states, and the four signals, `Awake1`, `Clk_en1` `Awake2` and `Clk_en2`, which are required for transitioning from one partition to another, as explained in section 7.3. A detailed timing diagram is presented in figure 7.7, which shows how the various signals provide support for clock and/or power gating.

7.3 Implementation

Although we are working on an automated method to implement the partitioning ideas presented, here we provide some guidelines and issues that need to be considered for efficient implementation based on our experiences with the manual partitioning of a counter circuit (see section 7.2).

Given a feasible and optimal state partitioning, the FSMD needs to be partitioned

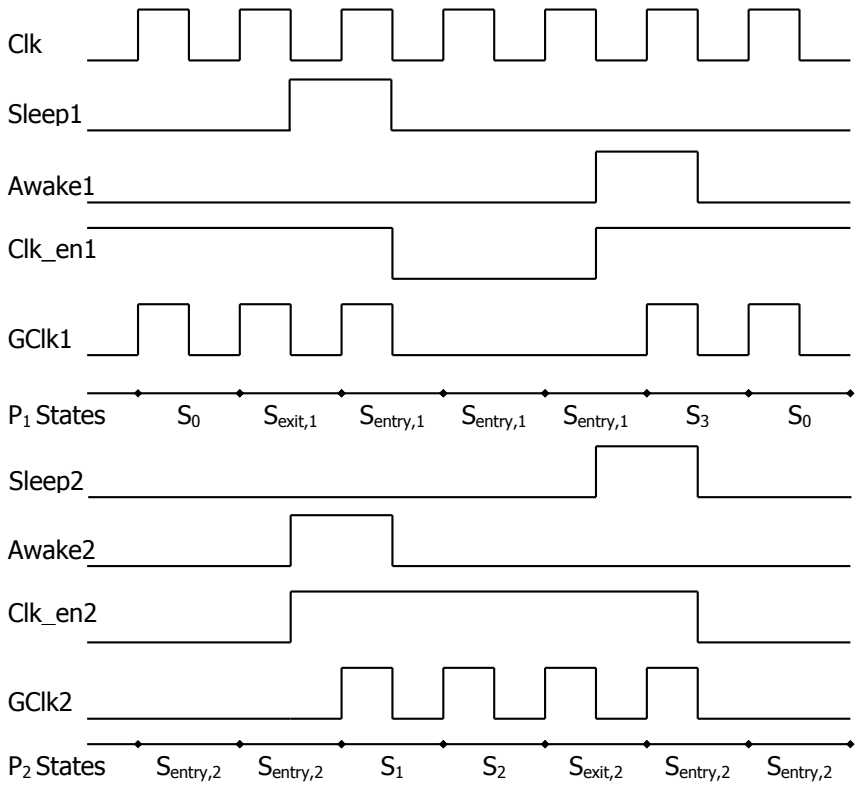


Figure 7.7: Counter timing after partitioning

in order to maintain the functionality provided by the original machine. Each FSMD partition, P_k, can be represented as

$$P_k = \langle S_k, s_{1,k}, s_{\text{entry},k}, s_{\text{exit},k}, I_k \cup STAT_k \cup IP_k, O_k \cup A_k \cup OP_k, \delta_k, \lambda_k \rangle \qquad (7.23)$$

where $s_{\text{entry},k}$ and $s_{\text{exit},k}$ are the entry and exit states of the submachine and $IP_k \subseteq VAR$ and $OP_k \subseteq VAR$ are the additional inputs and outputs, respectively, passed between submachines at the point of transition of the active submachine.

Once a partition, P_k, is activated it needs to update any changes to its data elements that may have occurred in the other machines while it was asleep. Therefore, upon activation, the sub-circuit needs to receive all shared data element values from the previously active partition. To provide this functionality, each partition needs the addition of an entry and exit state, s_{entry} and s_{exit}, respectively. When a partition is

activated, it is in its entry state, while the partition being deactivated is in its exit state. It is here that the transfer of data takes place from the previous to the new partition. This requires one additional clock cycle. During this time both partitions need to be active. It is important to note here that due to this additional clock cycle a performance penalty is incurred, which is equal to one cycle for every change in the active partition.

7.3.1 Clock Gating

To facilitate clock gating, we have introduced a `Clk_en` signal that is sent from the partition being deactivated to the new partition being activated. As with the `Awake` signal, this signal is activated by the old partition on the negative clock edge while it is in its exit state. However, this signal remains active as long as the new partition remains active. Therefore, this signal is lowered when the old partition is reactivated. The gated clock, `GClk`, used for the partitioned FSMD, P_k, can be represented as

$$\text{GClk}_k = \text{Clk} \quad \text{AND} \quad \text{Clk\_en}_k.$$

Figure 7.7 shows a sample timing diagram for the partitioned counter circuit.

7.3.2 Power Gating

Our anticipated implementation for power gating is that the "sleep" signal may be generated using an SR latch which is not part of the circuit components being powered off. The latch is set by the circuit partition which is ready to sleep, and is reset through an `Awake` signal sent from the partition being deactivated to the partition being activated. To allow set-up and hold times to be satisfied, both signals are raised on the negative clock edge. Further, these signals are active for only one clock cycle. The time required to power up a circuit is currently unknown and remains a topic for further exploration. This time is estimated as two clock cycles, resulting in a total of three cycles needed for each partition change.

7.4 Evaluation Framework

To test the effectiveness of our FSMD partitioning approach we have implemented a counter circuit and the set of circuits from the DSP kernel suite of the DSPstone benchmark [86]. Table 4.1 provides a list of the code fragments in this benchmark kernel suite.

The designs are implemented using CoDeL [75], which produces synthesizable FSMD descriptions in VHDL. The CoDeL compiler has been augmented to provide all the required parameters for our model.

The ILP is modeled using the AIMMS [19] modeling environment and solved using the CPLEX 10.0 solver. The solution to our model is obtained using a simulated annealing algorithm [82] implemented in Matlab [69].

For effective power estimation, trace data is used from circuit simulation using Synopsys. The trace data provides information on the state transition sequence during computation.

7.5 Power Estimation

Here we present a framework for estimating the potential power savings from partitioning a FSMD. This framework provides a coarse level approximate to the expected power savings based on examining the proportion of time each partition is active and the complexity of each partition.

The savings in power dissipation can be broken down into savings in static power and savings in dynamic power.

7.5.1 Static Power

Using experimentation we have found that, at least in the circuit implementations we have used, the static power of the circuit is roughly proportional to the amount of

sequential logic in the circuit[1]. Thus, by examining the number of sequential elements in the partitions, and the proportion of time they are put to sleep, we can estimate the static power savings.

Let Q equal the total number of register bits in the original partition P. The value Q can be calculated as

$$Q = \sum_{v_i \in VAR} |v_i|,$$

where $|v_i|$ is the word length of register v_i Similarly the total number of register bits in a submachine P_k is

$$Q_k = \sum_{v_i \in VAR_k} |v_i|,$$

where $VAR_k \subseteq VAR$ is the set of registers in partition k. The static power (SP) savings can now be expressed as

$$\text{SP Savings} = 1 - \sum_{k=1}^{M} \frac{P(P_k) \cdot Q_k}{Q} - \frac{\eta}{Q} \tag{7.24}$$

where the parameter $P(P_k)$ is the proportion of total time spent in partition k, obtained through trace analysis using behavioral simulation. The parameter η is used to capture the addition of any extra register bits required upon partitioning. This is required when extra state encoding bits are needed to incorporate the addition of the entry and exit states in the partitioned state machine. This only becomes significant when the number of states in a submachine is small (less than 4).

7.5.2 Dynamic Power

The dynamic power dissipation in a circuit is due to switching activity. After partitioning, the largest component of power savings is from the reduction of clocking of register components. All other activity in the circuit is necessarily the same as the unpartitioned FSMD to achieve the desired functionality. This includes register

[1] A power-area relationship has also been exploited in [56, 35].

value updates and arithmetic computations. Thus, the dynamic power savings can be estimated by examining the reduction in the number of register bits that need to be clocked after FSMD partitioning. However, we need to take into account the overhead added due to data communication whenever a change in active partition occurs. This switching overhead is given by

$$\text{Overhead} = 0.5 \sum_{k=1}^{M} \sum_{l=k+1}^{M} \left[NPC_{kl} \cdot \sum_{v_i \in TVAR_{kl}} |v_i| \right] \tag{7.25}$$

where NPC_{kl} is the number of partition changes between partitions k and l over a time period, T, $TVAR_{kl} = \{VAR_k \cap VAR_l\}$ is the number of shared variables between partitions k and l, and $|v_i|$ is the bit length of variable v_i. The factor 0.5 is used to capture that on average roughly half the bit values will be modified. This factor of 0.5 may be a bit conservative but the overhead is not particularly sensitive to this parameter. Further, we find that the switching overhead is so small (less than 0.5%) that the effect of this parameter on the overall dynamic power savings is negligible.

The dynamic power (DP) savings can be estimated as

$$\text{DP Savings} = \alpha_C \cdot \left[1 - \sum_{k=1}^{M} \frac{P(P_k) \cdot Q_k}{Q} - \frac{\text{Overhead}}{f \cdot T \cdot Q} \right] \tag{7.26}$$

where f is the circuit frequency, T is the run time, and α_C is the proportion of dynamic power due to clocking. In using the CoDeL environment to generate FSMD circuits we have found that the switching clock accounts for about 60% of the total dynamic power dissipation [4]. Therefore, we must adjust the total dynamic power savings by this factor. Thus, we use $\alpha_C = 0.6$.

Table 7.1: ILP estimated power savings

	Power Savings		Performance
	SP (%)	DP (%)	Overhead (%)
Counter	62.7	37.6	100.0
dot_product	32.8	19.7	35.3
real_update	37.3	22.4	35.3
complex_multiply	39.1	23.4	30.0
complex_update	37.0	22.2	36.0
Geo Mean	40.7	24.4	42.2
mat1x3	29.8	17.9	5.0
convolution	34.7	20.8	3.3
fir	34.8	20.9	3.0
n_real_updates	29.5	17.7	3.2
matrix	29.0	17.4	0.2
Geo Mean	31.4	18.9	1.9

7.5.3 Performance Overhead

The performance overhead from partitioning is determined by the number of extra cycles spent in changing partitions. Our power gating implementation (see section 7.3) estimates a penalty of three clock cycles for each partition change. The performance overhead can then be measured as

$$\text{Perf. Overhead} = \frac{3\sum_{k=1}^{M}\sum_{l=k+1}^{M} NPC_{kl}}{f \cdot T} \cdot 100\%. \tag{7.27}$$

In the case where power gating is not employed and only clock gating is used, there is no delay in charging up capacitances. In this case there is a penalty of a single cycle only resulting in one-third the performance overhead of that estimated in equation 7.27.

7.6 Estimation Results

7.6.1 Integer Linear Programming

Table 7.1 presents the estimated power savings using the framework from section 7.5. In all cases, the original machines have been partitioned into two submachines using the ILP formulation presented in section 7.1. The runtime of the ILP algorithm for partitioning is extremely slow. It can take hours for circuits with more than about 15 states. For circuits with more than 30 states, the runtime can be several days. For this reason, only a subset of the DSPstone circuits are partitioned and presented. We find that in most cases the static power savings are between 30% and 40%, while the dynamic power savings are between 15% and 25%.

Examining the performance overhead, we see that it varies considerably. As expected however, for the longer executing, complex kernels, the overhead is quite small (less than 5%), while for the short, simple kernels (shown in grey) the overhead is large making partitioning impractical.

7.6.2 Simulated Annealing

For effective power estimation, trace data is used from circuit simulation using Synopsys. This data provides information on the state transition sequence during computation. In tables 7.2, 7.3, and 7.4 we present the estimated power savings using the framework from section 7.5. The original machines have been partitioned into two, three and four submachines using the model presented in section 7.1. For the case of two partitions, we find that in most cases the static power savings are between 30% and 40%, while the dynamic power savings are between 15% and 25%.

Comparing the two partition results of the Simulated Annealing algorithm (table 7.2) with the ILP algorithm (table 7.1) we find that the results are almost the same. However, the ILP algorithm runs orders of magnitude slower than simulated annealing. This makes ILP impractical for circuits beyond about 20 states. Due to the long

Table 7.2: Simulated annealing estimated power savings (2 Partitions)

	Execution	Power Savings		Performance
Circuit	Cycles	SP (%)	DP (%)	Overhead (%)
counter	6	62.7	37.6	100.0
dot_product	17	37.6	22.6	35.3
real_update	17	33.6	20.1	35.3
complex_multiply	20	39.1	23.4	30.0
complex_update	25	37.0	22.2	36.0
iir_one_biquad	38	42.1	25.2	15.8
Geo Mean		41.1	24.6	35.9
mat1x3	120	29.8	17.9	5.0
convolution	182	34.7	20.8	3.3
iir_n_biquads	198	45.4	27.2	4.5
fir	203	34.8	20.9	3.0
n_real_updates	284	30.5	18.3	2.1
lms	356	22.0	13.2	2.5
n_complex_updates	554	32.2	19.3	1.1
fir2dim	919	30.5	18.3	0.7
matrix	5360	29.0	17.4	0.2
Geo Mean		31.6	19.0	1.7

Table 7.3: Simulated annealing estimated power savings (3 Partitions)

	Execution	Power Savings		Performance
Circuit	Cycles	SP (%)	DP (%)	Overhead (%)
counter	6	41.2	38.8	200.0
dot_product	17	59.9	36.0	52.9
real_update	17	52.8	31.7	70.6
complex_multiply	20	49.9	29.9	60.0
complex_update	25	51.1	30.7	48.0
iir_one_biquad	38	47.6	28.6	39.5
Geo Mean		50.1	32.4	66.3
mat1x3	120	41.4	24.8	7.5
convolution	182	37.0	22.2	6.6
iir_n_biquads	198	49.7	29.8	6.1
fir	203	41.0	24.6	5.9
n_real_updates	284	32.7	19.6	5.3
lms	356	26.8	16.1	4.2
n_complex_updates	554	38.2	22.9	19.0
fir2dim	919	37.0	22.2	1.3
matrix	5360	29.4	17.6	0.3
Geo Mean		36.5	21.9	4.0

Table 7.4: Simulated annealing estimated power savings (4 Partitions)

	Execution	Power Savings		Performance
Circuit	Cycles	SP (%)	DP (%)	Overhead (%)
counter	6	38.2	44.1	300.0
dot_product	17	68.1	40.8	70.6
real_update	17	61.9	37.2	88.2
complex_multiply	20	64.1	38.5	75.0
complex_update	25	58.6	35.2	60.0
iir_one_biquad	38	59.4	35.6	39.5
Geo Mean		57.4	38.4	83.2
mat1x3	120	44.4	26.6	15.0
convolution	182	37.4	22.5	8.2
iir_n_biquads	198	54.5	32.7	16.7
fir	203	42.1	25.3	7.4
n_real_updates	284	41.8	25.1	38.0
lms	356	30.5	18.3	5.1
n_complex_updates	554	44.1	26.5	27.1
fir2dim	919	21.8	13.1	38.5
matrix	5360	29.8	17.9	1.3
Geo Mean		37.3	22.4	11.7

run time of ILP, results for several circuits could not be computed in a reasonable time frame. Therefore, these circuits are missing from the ILP results in table 7.1.

For the case of three and four partitions, we see that the average power savings continues to increase. However, the rate of increase in power savings is decreasing, as the amount of logic isolation decreases and we end up with more transition variables. Further, there is a rise in the number of partition changes as the number of partitions increases affecting the dynamic power savings. This rise in the number of partition changes also results in significant performance loss. This phenomenon is exaggerated in the case of the *fir2dim* kernel as the power savings go down significantly with 4 partitions.

The circuits in tables 7.1, 7.2, 7.3, and 7.4 are arranged in the order of increasing algorithm complexity based on the number of execution cycles. Examining the performance overhead, we see that the impact is quite large for simpler kernels, while the more complex kernels show little loss of performance. In fact, the presence of loops facilitates the partitioning by providing minimum performance loss and large power savings by ensuring that the computation trace remains within its partition for long times and avoids expensive partition switches. This is exemplified by the *matrix* and *fir2dim* benchmarks. Both include loops. For the *matrix* benchmark, the innermost loop, which is executed a large number of times, is confined in its entirety within a partition in all cases (two, three and four partitions). Meanwhile, for the *fir2dim* benchmark, when four partitions are used the innermost loop is split between partitions and this results in both performance and power losses.

For the simple kernels (shown with a shaded background), the high performance impact suggests partitioning is not advisable. For more complex algorithms, where the performance overhead is less than 5% a real power savings opportunity is present with little impact on performance.

Table 7.5: Power and area for the Counter FSMD

Power

	Unpartitioned	Partitioned	Savings (%)
Static (nW)	775.7	450.3	41.9
Dynamic (uW)	64.1	37.6	41.3

Area

	Unpartitioned	Partitioned	Overhead (%)
Total cell area	445.2	580.7	30.4

7.6.3 Counter Implementation Results

We present here the post implementation results of partitioning the counter into two submachines. The VHDL description produced by CoDeL is manually partitioned into two submachines as described in section 7.2. The partitions (described in VHDL), along with the original FSMD are synthesized using the Synopsys Design Compiler using the TSMC 90nm general purpose CMOS technology.

Table 7.5 presents the power and area results for the original, unpartitioned FSMD and the partitioned FSMD. We see that after partitioning the static and dynamic power savings are about 41%. The area overhead of partitioning is about 30%. This is large since the original counter circuit area is small resulting in a larger relative overhead. For larger circuits the relative area overhead should be much less.

Comparing the post-implementation results for the counter from table 7.5 with the estimation results from table 7.2 we find that the estimation provides a reasonable approximation. The estimation framework developed in section 7.5 relates power savings to the percentage of time a partition remains idle. While accounting for the idle periods has been measured accurately through the traces we obtained from the synthesized circuits, the power estimates are a first order approximation. We have made assumptions relating power to the amount of sequential logic present, which introduce uncertainty. However, we are encouraged by the close (within 20%)

approximation we obtained for the counter circuit.

7.7 Summary

We have presented FSMD partitioning techniques, which efficiently decompose the controller and the datapath into multiple partitions. We first presented an ILP based approach which yields good results but has an extremely slow runtime making it impractical. The simulated annealing approach, presented next, has much shorter runtimes and also provides excellent results making it a useful and practical partitioning framework.

Implementing and analyzing a sample counter circuit in detail shows that up to 41% power savings are possible. An estimation framework is developed to evaluate the potential power savings from the partitioning methods developed. Using this framework on a broad set of circuits we find that, in most cases, significant power savings can be expected.

Chapter 8

Conclusions

Reasoning draws a conclusion, but does not make the
conclusion certain, unless the mind discovers
it by the path of experience.
- Roger Bacon

In this work, we have laid a foundation for various techniques, that can be fully automated at the system-level, to help reduce power dissipation in CMOS circuits. The major accomplishments presented in this thesis can be summarized as follows.

- We have created a system-level design platform, called CoDeL, which allows architectural descriptions at the algorithmic level. Support has been added for fixed point data types and operations making this platform particularly suitable for DSP applications. The power of this design environment lies in its ability to allow the designer to create a highly power efficient computational architecture extremely quickly. The speed of design and development using this platform is of similar order to that of a microprocessor, however the resulting energy requirements for the application specific architecture developed using CoDeL can be orders of magnitude less than the microprocessor. Further, with capabilities for automated clock gating and possibilities of automated power gating and partitioning, in the future, CoDeL is the only known, power aware,

system-level design platform.

- The automated clock gating extension to CoDeL is an effective tool for lowering dynamic power consumption. The CoDeL compiler uses a built-in framework to provide a quick and accurate estimate of the expected power savings from clock gating. This provides the designers with an early indication of power savings and allows them to use this information to create a power efficient compute architecture.

- To lower static power, a power gating mechanism that gates individual registers was studied which can be automatically implemented at compile time. In this study, it was discovered that a static gating scheme provides the same or better gating effectiveness and performance than dynamic schemes. Further, the static nature of the proposed power gating mechanism means that the area overhead of gating is significantly reduced as compared with comparable dynamic methods.

- Finally, we have proposed a partitioning approach that can be fully automated to allow macro-level isolation of circuit components for clock or power gating. The proposed partitioning framework uses a simulated annealing algorithm which is fast and provides excellent potential for power savings.

8.1 Future Research

There are some outstanding issues and areas still to explore. Here we provide a summary where we categorize and list the major topics.
The following is a list of open items in the area of clock gating.

- The effect and performance impact of clock gating on the critical path length needs to be carefully studied.

- The power analysis performed through Synopsys tools needs to be verified through actual implementation on Silicon.

- Since FPGAs do not support clock gating, we need to look at general architectural guidelines to allow clock gating support in FPGAs.

The following is a list of open power gating issues.

- We need to study the power up and power down effects in VLSI circuits to discover specific timing requirements.
- The power gating mechanisms needs to be implemented to get accurate power savings.
- Further analysis and exploration of branch prediction mechanisms is needed to qualify the optimality of the presented techniques. An oracle, or perfect, branch prediction method can be used for this evaluation.

The following is a list of open issues in partitioning.

- The partitioning framework needs to be automated and various circuits need to be partitioned to get accurate results for the power savings.
- Other partitioning models need to be explored, including graph partitioning approaches.
- The partitioning approach needs to be implemented and tested on an FPGA environment, where the FPGA provides separated clock and/or power domains, where individual partitions can be mapped.

Finally, we look forward to building partnerships with industry members that develop and provide robust system-level design tools in the hope of integrating our power reduction techniques into their product offerings.

Appendix A

CoDeL Language Reference

The basic notation for describing the behavior of a controller in CoDeL is shown in figure A.1. It comprises the following parts:

- Structure declarations
- Macros
- Module declaration
- Port declarations
- Register declarations
- CoDeL Statements

```
[structure declarations]
[macros]
module  module_name ( port declarations )
{
   [ register declarations ]
   CoDeL Statements
}
```

Figure A.1: Basic structure of a CoDeL program

```
bitstruct  field3
{
   (bits)  b[3];
}

bitstruct field2
{
   (bits)  b[2];
}

bitstruct address
{
   (field2)    d1;
   (field2)    d2;
   (field3)    d3;
   (field3)    d4;
   (field3)    d5;
}

bitstruct  header
{
   (address)  source;
   (address)  destination;
   (bits)     type[2];
}
```

Figure A.2: Bitstruct example

A.1 Structure Declarations

Analogous to the *struct* definition in C, *bitstruct* in CoDeL defines complex data structures. A *bitstruct* may be composed of collections of bits or *bitstruct*s. An example is presented in figure A.2.

A.2 Macros

Macros are bodies of code allowing code reuse and better readability. Macros can contain any number of input parameters and can be nested. They are defined through the use of the keyword *macro*. An example is shown in figure A.3.

```
macro  macro_name ( parameters )
{
   CoDeL Statements
}
```

Figure A.3: Example macro definition

A.3 Module Declarations

A CoDeL program defines the name of a module beginning with the keyword *module* which uniquely identifies the top level circuit that is being synthesized. A *module* is declared as follows:

`module` *module_name* `([`*port_declarations*`]);`

A.4 Ports and Protocols

A.4.1 Port declarations

The communication abstraction that CoDeL assumes is that of interacting agents that exchange data of a specific and known structure and accomplish this exchange using a predetermined protocol.

Ports encapsulate both data and protocols in an abstract data type and implement the I/O interaction between the module and external components. Ports are of type *input* or *output* [1].

A novel feature in the definition of ports is the introduction of protocols as objects that can be associated with any arbitrary port. The inclusion of a protocol in the definition of a port allows the *hiding* of the details of the I/O interaction.

If no protocol (null protocol) is included in the declaration of a port, the designer must explicitly specify the necessary control sequences that will effect the data transfer through the port. This technique can be used to develop new protocols.

The syntax for port declarations in CoDeL is given below.

[1] Bidirectional ports have not yet been introduced in the current implementation of CoDeL.

```
# Define a 16-bit address
# in 4 dimensions
bitstruct mixed_radix_4
{
   (bits) field1[4];
   (bits) field2[4];
   (bits) field3[4];
   (bits) field4[4];
}

# Define a 36-bit
# message header using
# the above
bitstruct data_frame
{
   (mixed_radix_4) source_address;
   (mixed_radix_4) destn_address;
   (bits) header[4];
}

in  (data_frame) p1 with input_handshake;
out (data_frame) p3 with output_handshake;
```

Figure A.4: Examples of port and protocol declarations

in (*bitstruct_name*) *port_name* [with *protocol_name*];

out(*bitstruct_name*) *port_name* [with *protocol_name*];

Examples of port definitions and protocol usage are given in figure A.4.

A.4.2 Exporting/Importing data

input (*port_name*);

output (*port_name*);

isready (*port_name*);

The input/output primitives *import/export* data through the declared ports engaging the associated protocols, if any. The *isready* primitive is used to check the availability of data in an input port or the readiness of an output port to accept data.

A.5 Register Declarations

Registers are declared as having a particular structure through a declaration of the form

`register [(`*bitstruct_name*`)]` *register_name*`;`

The default structure is bits and the components of a register can be addressed at the bitstruct level or at the bit level. We use a dot notation to identify the components under the convention that a sequence of dot separated fields represents a node in the tree representing the structure of the object and it consists of all the bits at the leaf nodes of the sub-tree rooted at the node. The individual bit positions can also be selected using brackets while bit ranges are specified by using a colon. For example, `r2(5:0)` addresses bits starting from position 5 (most significant) to position 0 (least significant)

A.6 CoDeL Statements

A.6.1 Assignment statements

A number of operations may need to be performed on the bitfields composing a frame. The assignment statement places the results of a computation to an output port or a register or portions thereof as discussed earlier.

register_name | *output_port_name* = *computation_expression*

Assignments manifest as a Register Transfer Level (RTL) implementation where data is stored in registers, operations are effected by a combinational circuit or a sequence of combinational circuits and the results are stored back in a register.

A.6.2 CoDeL Operators

A *computation_expression* can have either input ports or registers as its operands. It is formed using a number of standard C operators, summarized in Table A.1, and following the usual C associativity and precedence rules. Parentheses affect the

Table A.1: List of CoDeL operators

Symbol	Operation	Symbol	Operation
(,... ,...,)	Concatenation	+, - , *	Arithmetic
>, >=, <, <=	Relational	++ , –	Increment/Decrement
==	Logical Equality	!=	Logical Inequality
!	Logical Negation		
||	Logical OR	&&	Logical AND
∼	Bit-wise Negation	&	Bit-wise AND
|	Bit-wise OR	^	Bit-wise XOR
∼ |	Bit-wise NOR	∼&	Bit-wise NAND
<<	Left Shift	>>	Right Shift
<@	Left Rotate	>@	Right Rotate
? :	Conditional	(:)	Bit Selector

structure of the constructed circuit.

A.6.3 Control Statements

CoDeL avoids the explicit description of the control path which is automatically synthesized from the algorithm itself. In a sequential environment, the control path is inherently described by the order of the operations in a program. In CoDeL, the control path of the design is extracted based on the sequentiality of the algorithm and it includes states which explicitly clock the registers included in the design. The basic control structures in CoDeL include loop, conditional and wait primitives. These are listed below.

Loop primitives:

`while` (*condition*)

{

CoDeL statements

}

`for` (*initializer*; *test_condition*; *incr/decr_expression*)

{

CoDeL statements

}

Note that the CoDeL compiler inserts the appropriate branches to exit (re-enter) the "while" loop when the test condition is false (true) but performs a simple loop unfolding of the statements in the "for" loop substituting the index as the case may be for the given range.

Branch primitive:

`if` (*condition*)

{

CoDeL statements

}

`else`

{

CoDeL statements

}

where the else statement is optional.

Wait statement:

`wait (`*condition*`);`

The wait primitive maintains the current state of the computation until a certain condition is satisfied. It is used mainly for synchronization with external signals.

Call statement:

`call` *m*acro_name`(`*parameters*`);`

The call statement causes the program flow to jump to the macro code of the specified macro with the parameters provided.

Appendix B

DSPstone Benchmark - CoDeL Source Code

B.1 Shared Macros

```
macro fpu_add(opA, opB, result)
{
    opa_i = opA;
    opb_i = opB;
    fpu_op_i = 0;
    delay;
    result = output_o;
}

macro fpu_sub(opA, opB, result)
{
    opa_i = opA;
    opb_i = opB;
    fpu_op_i = 1;
    delay;
    result = output_o;
}

macro fpu_mult(opA, opB, result)
{
    opa_i = opA;
    opb_i = opB;
    fpu_op_i = 2;
    delay;
    result = output_o;
}

macro fpu_div(opA, opB, result)
{
    opa_i = opA;
    opb_i = opB;
    fpu_op_i = 3;
    delay;
    result = output_o;
}

macro fpu_sqrt(opA, opB, result)
{
    opa_i = opA;
```

```
    opb_i = opB;
    fpu_op_i = 4;
    delay;
    result = output_o;
}

macro mem_in1(addr, data)
{
    mem_addr1 = addr;
    mem_wr1 = 0;
    delay;
    data = mem_data_in1;
}

macro mem_out1(addr, data)
{
    delay;
    mem_addr1 = addr;
    mem_wr1 = 1;
    mem_data_out1 = data;
}

macro mem_in2(addr, data)
{
    mem_addr2 = addr;
    mem_wr2 = 0;
    delay;
    data = mem_data_in2;
}

macro mem_out2(addr, data)
{
    delay;
    mem_addr2 = addr;
    mem_wr2 = 1;
    mem_data_out2 = data;
}

macro mem_in_par(addr1, data1, addr2, data2)
{
    mem_addr1 = addr1;
    mem_wr1 = 0;
    mem_addr2 = addr2;
    mem_wr2 = 0;
    delay;
    data1 = mem_data_in1;
    data2 = mem_data_in2;
}

macro mem_out_par(addr1, data1, addr2, data2)
{
    delay;
    mem_addr1 = addr1;
    mem_wr1 = 1;
    mem_data_out1 = data1;
    mem_addr2 = addr2;
    mem_wr2 = 1;
    mem_data_out2 = data2;
}
```

B.2 real_update

```
macro init()
{
    call mem_out1(100, 0xD6D3);    # A = 10
    call mem_out2(200, 0x09B5);    # B = 2
    call mem_out1(300, 0x9500);    # C = 1
```

```
    call mem_out2(400, 0);        # D = 0
}

macro real_update()
{
    call mem_in_par(100, A, 200, B);
    call mem_in1(300, C);
    D = 0x0000;    # 0
    call fpu_mult(A, B, D);
    call fpu_add(C, D, D);
    call mem_out1(400, D);
}


module real_update (
    in    start,
    out    ready,
    out   opa_i[16],        # fpu operand A
    out   opb_i[16],        # fpu operand B
    out   fpu_op_i[3],    # fpu operation
    in    output_o[16],    # fpu result
    out   mem_addr1[32],    # Memory address
    out    mem_data_out1[16],# Memory data out
    in    mem_data_in1[16],    # Memory data in
    out    mem_wr1,        # Memory write enable
    out   mem_addr2[32],    # Memory address
    out    mem_data_out2[16],# Memory data out
    in    mem_data_in2[16],    # Memory data in
    out    mem_wr2,        # Memory write enable
    out    profile        # profile enable
)
{
    register A[16];
    register B[16];
    register C[16];
    register D[16];

    ready = 1;
    wait(start);
    ready = 0;

    call init();

    delay;
    profile = 1;
    call real_update();
    delay;
    profile = 0;
}
```

B.3 n_real_updates

```
macro init(offset)
{
    address = 100 + offset;
    call mem_out1(address, 0x76A1);    # A = 10
    address = 200 + offset;
    call mem_out2(address, 0xBB23);    # B = 2
    address = 300 + offset;
    call mem_out1(address, 0x2947);    # C = 10
    address = 400 + offset;
    call mem_out2(address, 0);    # D = 0
}

macro real_update(offset)
{
```

```
    mem_addr1 = 200 + offset;
    mem_wr1 = 0;
    mem_addr2 = 100 + offset;
    mem_wr2 = 0;
    call fpu_mult(A, B, D);
    B = mem_data_in1;
    A = mem_data_in2;

    mem_addr1 = 300 + offset;
    mem_wr1 = 0;
    call fpu_add(C, D, D);
    C = mem_data_in1;

    mem_addr2 = 400 + offset;
    mem_wr2 = 1;
    mem_data_out2 = output_o;
}

module n_real_updates (
    in    start,
    out    ready,
    out   opa_i[16],        # fpu operand A
    out   opb_i[16],        # fpu operand B
    out   fpu_op_i[3],    # fpu operation
    in    output_o[16],    # fpu result
    out   mem_addr1[32],    # Memory address
    out    mem_data_out1[16],# Memory data out
    in    mem_data_in1[16],    # Memory data in
    out    mem_wr1,        # Memory write enable
    out   mem_addr2[32],    # Memory address
    out    mem_data_out2[16],# Memory data out
    in    mem_data_in2[16],    # Memory data in
    out    mem_wr2,        # Memory write enable
    out    profile        # profile enable
)
{
    register A[16];
    register B[16];
    register C[16];
    register D[16];
    register i[5];
    register address[32];

    ready = 1;
    wait(start);
    ready = 0;

    i = 0;
    while (i < 16)
    {
        call init(i);
        i = i + 1;
    }

    delay;
    profile = 1;
    i = 0;
    mem_addr1 = 100 + i;
    mem_wr1 = 0;
    mem_addr2 = 200 + i;
    mem_wr2 = 0;
    delay;
    A = mem_data_in1;
    B = mem_data_in2;

    while (i < 16)
    {
```

```
        call real_update(i);
        i = i + 1;
    }
    delay;
    profile = 0;
}
```

B.4 complex_update

```
macro init()
{
    call mem_out_par(100, 0x3856, 200, 0x7A04);    # Ar = 2, Br = 2
    call mem_out_par(300, 0x28B6, 400, 0);        # Cr = 3, Dr = 0

    call mem_out_par(150, 0x061A, 250, 0x0475);    # Ai = 1, Bi = 5
    call mem_out_par(350, 0x0C37, 450, 0);        # Ci = 4, Di = 0
}

macro complex_update()
{
    call mem_in_par(100, Ar, 150, Ai);
    call mem_in_par(200, Br, 250, Bi);
    call mem_in_par(300, Cr, 350, Ci);

    Dr = 0;    # 0
    Di = 0;    # 0

    call fpu_mult(Ar, Br, temp1);
    call fpu_mult(Ai, Bi, temp2);
    call fpu_add(Cr, temp1, Dr);
    call fpu_sub(Dr, temp2, Dr);

    call fpu_mult(Ar, Bi, temp1);
    call fpu_mult(Ai, Br, temp2);
    call fpu_add(Ci, temp1, Di);
    call fpu_add(Di, temp2, Di);

    call mem_out_par(400, Dr, 450, Di);
}

module complex_update (
    in    start,
    out    ready,
    out   opa_i[16],        # fpu operand A
    out   opb_i[16],        # fpu operand B
    out   fpu_op_i[3],    # fpu operation
    in    output_o[16],    # fpu result
    out   mem_addr1[32],    # Memory address
    out    mem_data_out1[16],# Memory data out
    in    mem_data_in1[16],    # Memory data in
    out    mem_wr1,        # Memory write enable
    out   mem_addr2[32],    # Memory address
    out    mem_data_out2[16],# Memory data out
    in    mem_data_in2[16],    # Memory data in
    out    mem_wr2,        # Memory write enable
    out    profile        # profile enable
)
{
    register Ar[16];
    register Br[16];
    register Cr[16];
    register Dr[16];
    register Ai[16];
    register Bi[16];
    register Ci[16];
    register Di[16];
    register temp1[16];
```

```
    register temp2[16];

    ready = 1;
    wait(start);
    ready = 0;

    call init();

    delay;
    profile = 1;
    call complex_update();
    delay;
    profile = 0;
}
```

B.5 n_complex_updates

```
macro init(offset)
{
    address = 100 + offset;
    call mem_out1(address, 0x2847);    # Ar = 2
    address = 200 + offset;
    call mem_out2(address, 0x7455);    # Br = 2
    address = 300 + offset;
    call mem_out1(address, 0x0044);    # Cr = 3
    address = 400 + offset;
    call mem_out2(address, 0);        # Dr = 0

    address = 150 + offset;
    call mem_out1(address, 0x7544);    # Ai = 1
    address = 250 + offset;
    call mem_out2(address, 0x1112);    # Bi = 5
    address = 350 + offset;
    call mem_out1(address, 0x0046);    # Ci = 4
    address = 450 + offset;
    call mem_out2(address, 0);        # Di = 0

}

macro complex_update(offset)
{
    mem_addr1 = 100 + offset;
    mem_wr1 = 0;
    mem_addr2 = 200 + offset;
    mem_wr2 = 0;
    delay;
    Ar = mem_data_in1;
    Br = mem_data_in2;

    call fpu_mult(mem_data_in1, mem_data_in2, temp1);

    mem_addr1 = 300 + offset;
    mem_wr1 = 0;
    delay;
    Cr = mem_data_in1;

    call fpu_add(mem_data_in1, temp1, Dr);

    mem_addr1 = 150 + offset;
    mem_wr1 = 0;
    mem_addr2 = 250 + offset;
    mem_wr2 = 0;
    delay;
    Ai = mem_data_in1;
    Bi = mem_data_in2;

    call fpu_mult(mem_data_in1, mem_data_in2, temp2);
```

```
    call fpu_mult(Ar, Bi, temp1);

    mem_addr1 = 350 + offset;
    mem_wr1 = 0;
    delay;
    Ci = mem_data_in1;

    call fpu_sub(Dr, temp2, Dr);

    call fpu_mult(Ai, Br, temp2);
    call fpu_add(Ci, temp1, Di);
    call fpu_add(output_o, temp2, Di);

    mem_addr1 = 400 + offset;
    mem_wr1 = 1;
    mem_data_out1 = Dr;
    mem_addr2 = 450 + offset;
    mem_wr2 = 1;
    mem_data_out2 = Di;
}

module n_complex_updates (
    in    start,
    out    ready,
    out   opa_i[16],         # fpu operand A
    out   opb_i[16],         # fpu operand B
    out   fpu_op_i[3],     # fpu operation
    in    output_o[16],     # fpu result
    out   mem_addr1[32],     # Memory address
    out    mem_data_out1[16],# Memory data out
    in    mem_data_in1[16],     # Memory data in
    out    mem_wr1,         # Memory write enable
    out   mem_addr2[32],     # Memory address
    out    mem_data_out2[16],# Memory data out
    in    mem_data_in2[16],     # Memory data in
    out    mem_wr2,         # Memory write enable
    out    profile         # profile enable
)
{
    register Ar[16];
    register Br[16];
    register Cr[16];
    register Dr[16];
    register Ai[16];
    register Bi[16];
    register Ci[16];
    register Di[16];
    register temp1[16];
    register temp2[16];
    register i[5];
    register address[32];

    ready = 1;
    wait(start);
    ready = 0;

    i = 0;
    while (i < 16)
    {
        call init(i);
        i = i + 1;
    }

    delay;
    profile = 1;
    i = 0;
```

```
    while (i < 16)
    {
        call complex_update(i);
        i = i + 1;
    }
    delay;
    profile = 0;
}
```

B.6 dot_product

```
macro init()
{
    call mem_out_par(100, 0x4857, 200, 0x0A54);    # A1 = 2, B1 = 1
    call mem_out_par(150, 0x751B, 250, 0x2854);    # A2 = 1, B2 = 5
}

macro dot_prod()
{
    call mem_in_par(100, A1, 150, A2);
    call mem_in_par(200, B1, 250, B2);

    call fpu_mult(A1, B1, temp1);
    call fpu_mult(A2, B2, C);
    call fpu_add(C, temp1, C);

    call mem_out1(300, C);
}

module dot_product (
    in    start,
    out    ready,
    out   opa_i[16],        # fpu operand A
    out   opb_i[16],        # fpu operand B
    out   fpu_op_i[3],    # fpu operation
    in    output_o[16],    # fpu result
    out   mem_addr1[32],    # Memory address
    out    mem_data_out1[16],# Memory data out
    in    mem_data_in1[16],    # Memory data in
    out    mem_wr1,        # Memory write enable
    out   mem_addr2[32],    # Memory address
    out    mem_data_out2[16],# Memory data out
    in    mem_data_in2[16],    # Memory data in
    out    mem_wr2,        # Memory write enable
    out    profile        # profile enable
)
{
    register A1[16];
    register B1[16];
    register A2[16];
    register B2[16];
    register C[16];
    register temp1[16];

    ready = 1;
    wait(start);
    ready = 0;

    call init();

    delay;
    profile = 1;
    call dot_prod();
    delay;
    profile = 0;
}
```

B.7 mat1x3

```
macro init_h(offset)
{
    address = 100 + offset;
    call mem_out1(address, 0x0938);    # h_i = 1
}

macro init_xy(offset)
{
    address = 200 + offset;
    call mem_out1(address, 0x0036);    # x_i = 1
    address = 300 + offset;
    call mem_out2(address, 0);        # y_i = 0
}

macro accumulate(hi, xi)
{
    call fpu_mult(h, x, temp1);

    mem_addr1 = 100 + hi;
    mem_wr1 = 0;
    mem_addr2 = 200 + xi;
    mem_wr2 = 0;
    call fpu_add(output_o, y, y);
    h = mem_data_in1;
    x = mem_data_in2;
    hi = hi + 1;
    xi = xi + 1;
}

module mat1x3 (
    in    start,
    out    ready,
    out   opa_i[16],         # fpu operand A
    out   opb_i[16],         # fpu operand B
    out   fpu_op_i[3],    # fpu operation
    in    output_o[16],    # fpu result
    out   mem_addr1[32],    # Memory address
    out    mem_data_out1[16],# Memory data out
    in    mem_data_in1[16],    # Memory data in
    out    mem_wr1,        # Memory write enable
    out   mem_addr2[32],    # Memory address
    out    mem_data_out2[16],# Memory data out
    in    mem_data_in2[16],    # Memory data in
    out    mem_wr2,        # Memory write enable
    out    profile        # profile enable
)
{
    register x[16];
    register y[16];
    register h[16];

    register temp1[16];
    register i[4];
    register hi[4];
    register row[3];
    register address[32];

    ready = 1;
    wait(start);
    ready = 0;

    hi = 0;
    while (hi < 9)
    {
        call init_h(hi);
        hi = hi + 1;
```

```
    }

    row = 0;
    while (row < 3)
    {
        call init_xy(row);
        row = row + 1;
    }

    delay;
    profile = 1;

    row = 0;
    hi = 0;
    while (row < 3)
    {
        i = 0;
        y = 0;
        mem_addr1 = 100 + hi;
        mem_wr1 = 0;
        mem_addr2 = 200 + i;
        mem_wr2 = 0;
        delay;
        h = mem_data_in1;
        hi = hi + 1;
        x = mem_data_in2;
        i = i + 1;

        while (i < 3)
        {
            call accumulate(hi, i);
        }
        delay;
        mem_addr1 = 300 + row;
        mem_wr1 = 1;
        mem_data_out1 = y;
        row = row + 1;
    }

    delay;
    profile = 0;
}
```

B.8 matrix

```
macro init(offset)
{
    address = 100 + offset;
    call mem_out1(address, 0x0030);        # a_i = 1
    address = 200 + offset;
    call mem_out1(address, 0x0010);        # b_i = 1
}

macro accumulate(ai, bi)
{
    C = output_o;
    opa_i = A;
    opb_i = B;
    fpu_op_i = 2;
    mem_addr1 = 100 + ai;
    mem_wr1 = 0;
    mem_addr2 = 200 + bi;
    mem_wr2 = 0;
    delay;
    A = mem_data_in1;
    ai = ai + 1;
    B = mem_data_in2;
```

```
    bi = bi + 10;
    opa_i = output_o;
    opb_i = C;
    fpu_op_i = 0;
}

module matrix (
    in    start,
    out    ready,
    out   opa_i[16],         # fpu operand A
    out   opb_i[16],         # fpu operand B
    out   fpu_op_i[3],    # fpu operation
    in    output_o[16],     # fpu result
    out   mem_addr1[32],    # Memory address
    out    mem_data_out1[16],# Memory data out
    in    mem_data_in1[16],    # Memory data in
    out    mem_wr1,         # Memory write enable
    out   mem_addr2[32],    # Memory address
    out    mem_data_out2[16],# Memory data out
    in    mem_data_in2[16],    # Memory data in
    out    mem_wr2,         # Memory write enable
    out    profile         # profile enable
)
{
    register A[16];
    register B[16];
    register C[16];

    register temp1[16];
    register ai[7];
    register bi[7];
    register ci[7];
    register X[4];
    register Y[4];
    register Z[4];
    register address[32];

    ready = 1;
    wait(start);
    ready = 0;

    ai = 0;
    while (ai < 100)
    {
        call init(ai);
        ai = ai + 1;
    }

    delay;
    profile = 1;
    ci = 0;
    Z = 0;
    while (Z < 10)
    {
        ai = 0;
        X = 0;
        while (X < 10)
        {
            bi = Z*10;
            C = 0;
            Y = 0;
            ci = ci + 1;
            mem_addr1 = 100 + ai;
            mem_wr1 = 0;
            mem_addr2 = 200 + bi;
            mem_wr2 = 0;
            delay;
```

```
            A = mem_data_in1;
            ai = ai + 1;
            B = mem_data_in2;
            bi = bi + 10;
            opa_i = mem_data_in1;
            opb_i = mem_data_in2;
            fpu_op_i = 2;

            while (Y < 10)
            {
                call accumulate(ai, bi);
                Y = Y + 1;
            }
            mem_addr2 = 299 + ci;            # update c_i
            mem_wr2 = 1;
            mem_data_out2 = output_o;
            X = X + 1;
        }
        Z = Z + 1;
    }
    delay;
    profile = 0;
}
```

B.9 convolution

```
macro init(offset)
{
    address = 100 + offset;
    call mem_out1(address, 0x9465);        # x_i = 1
    address = 200 + offset;
    call mem_out2(address, 0x04B8);        # h_i = 1
}

macro accumulate(xi, hi)
{
    mem_addr1 = 100 + xi;
    mem_wr1 = 0;
    mem_addr2 = 200 + hi;
    mem_wr2 = 0;
    call fpu_mult(X, H, temp1);
    X = mem_data_in1;
    H = mem_data_in2;

    xi = xi + 1;
    hi = hi - 1;

    call fpu_add(output_o, Y, Y);
}

module convolution (
    in    start,
    out    ready,
    out   opa_i[16],        # fpu operand A
    out   opb_i[16],        # fpu operand B
    out   fpu_op_i[3],    # fpu operation
    in    output_o[16],    # fpu result
    out   mem_addr1[32],    # Memory address
    out    mem_data_out1[16],# Memory data out
    in    mem_data_in1[16],    # Memory data in
    out    mem_wr1,        # Memory write enable
    out   mem_addr2[32],    # Memory address
    out    mem_data_out2[16],# Memory data out
    in    mem_data_in2[16],    # Memory data in
    out    mem_wr2,        # Memory write enable
    out    profile        # profile enable
)
```

```
{
    register X[16];
    register H[16];
    register Y[16];

    register temp1[16];
    register xi[5];
    register hi[5];
    register address[32];

    ready = 1;
    wait(start);
    ready = 0;

    xi = 0;
    while (xi < 16)
    {
        call init(xi);
        xi = xi + 1;
    }

    delay;
    profile = 1;

    xi = 0;
    hi = 15;
    Y = 0;

    mem_addr1 = 100 + xi;
    mem_wr1 = 0;
    mem_addr2 = 200 + hi;
    mem_wr2 = 0;
    delay;
    X = mem_data_in1;
    H = mem_data_in2;

    xi = xi + 1;
    hi = hi - 1;

    while (xi < 16)
    {
        call accumulate(xi, hi);
    }

    delay;
    profile = 0;
}
```

B.10 fir

```
macro init(offset)
{
    address = 100 + offset;
    call mem_out1(address, 0x07A1);        # x_i = 1
    address = 200 + offset;
    call mem_out2(address, 0x195D);        # h_i = 1
}

macro accumulate(xi, hi, xi2)
{
    mem_addr1 = 100 + xi2;
    mem_wr1 = 0;
    call fpu_mult(X, H, temp1);
    X = mem_data_in1;
    xi2 = xi2 - 1;

    call fpu_add(output_o, Y, Y);
```

```
    mem_addr1 = 100 + xi;
    mem_wr1 = 1;
    mem_data_out1 = X;

    mem_addr2 = 200 + hi;
    mem_wr2 = 0;
    delay;
    H = mem_data_in2;

    xi = xi - 1;
    hi = hi - 1;
}

module fir (
    in    start,
    out    ready,
    out   opa_i[16],        # fpu operand A
    out   opb_i[16],        # fpu operand B
    out   fpu_op_i[3],    # fpu operation
    in    output_o[16],    # fpu result
    out   mem_addr1[32],    # Memory address
    out    mem_data_out1[16],# Memory data out
    in    mem_data_in1[16],    # Memory data in
    out    mem_wr1,        # Memory write enable
    out   mem_addr2[32],    # Memory address
    out    mem_data_out2[16],# Memory data out
    in    mem_data_in2[16],    # Memory data in
    out    mem_wr2,        # Memory write enable
    out    profile        # profile enable
)
{
    register X[16];
    register H[16];
    register Y[16];

    register temp1[16];
    register xi[5];
    register xi2[5];
    register hi[5];
    register i[5];
    register address[32];

    ready = 1;
    wait(start);
    ready = 0;

    xi = 0;
    while (xi < 16)
    {
        call init(xi);
        xi = xi + 1;
    }

    delay;
    profile = 1;

    xi = 15;
    xi2 = 14;
    hi = 15;
    i = 0;
    Y = 0;

    mem_addr1 = 100 + xi;
    mem_wr1 = 0;
    mem_addr2 = 200 + hi;
    mem_wr2 = 0;
```

```
    delay;
    X = mem_data_in1;
    H = mem_data_in2;

    while (i < 15)
    {
        call accumulate(xi, hi, xi2);
        i = i + 1;
    }
    call accumulate(xi, hi, xi2);
    address = 101 + xi;
    call mem_out1(address, 0f1.0);

    delay;
    profile = 0;
}
```

B.11 fir2dim

```
macro init_array(start_addr, num_vals, init_val)
{
    i = 0;
    while (i < num_vals)
    {
        address = start_addr + i;
        call mem_out1(address, init_val);
        i = i + 1;
    }
}

macro init()
{
    call init_array(100, 16, 0x0030);    # image[i,j] = 1
    call init_array(200, 9, 0x0050);    # coeff[i,j] = 1
    call init_array(400, 16, 0x0010);    # output[i,j] = 1

    i = 0;
    parray = 0;
    while (i < 6)
    {
        address = 300 + parray;
        call mem_out1(address, 0);
        i = i + 1;
        parray = parray + 1;
    }

    pimage = 0;
    j = 0;
    while (j < 4)
    {
        address = 300 + parray;
        call mem_out1(address, 0);
        parray = parray + 1;

        i = 0;
        while (i < 4)
        {
            address = 100 + pimage;
            call mem_in1(address, image);
            pimage = pimage + 1;

            address = 300 + parray;
            call mem_out2(address, image);
            parray = parray + 1;

            i = i + 1;
        }
```

```
        address = 300 + parray;
        call mem_out1(address, 0);
        parray = parray + 1;

        j = j + 1;
    }

    i = 0;
    while (i < 6)
    {
        address = 300 + parray;
        call mem_out1(address, 0);
        i = i + 1;
        parray = parray + 1;
    }


}

macro accumulate(coeffi, arri)
{
    mem_addr1 = 200 + coeffi;
    mem_wr1 = 0;
    mem_addr2 = 300 + arri;
    mem_wr2 = 0;
    delay;
    coeff = mem_data_in1;
    darray = mem_data_in2;

    coeffi = coeffi + 1;
    arri = arri + 1;

    call fpu_mult(mem_data_in1, mem_data_in2, temp1);

    call fpu_add(output_o, outval, outval);
}


module fir2dim (
    in    start,
    out    ready,
    out   opa_i[16],        # fpu operand A
    out   opb_i[16],        # fpu operand B
    out   fpu_op_i[3],    # fpu operation
    in    output_o[16],    # fpu result
    out   mem_addr1[32],    # Memory address
    out    mem_data_out1[16],# Memory data out
    in    mem_data_in1[16],    # Memory data in
    out    mem_wr1,        # Memory write enable
    out   mem_addr2[32],    # Memory address
    out    mem_data_out2[16],# Memory data out
    in    mem_data_in2[16],    # Memory data in
    out    mem_wr2,        # Memory write enable
    out    profile        # profile enable
)
{
    register image[16];
    register darray[16];
    register coeff[16];
    register outval[16];

    register temp1[16];
    register pimage[5];
    register parray[6];
    register parray2[6];
    register parray3[6];
```

```
    register pcoeff[4];
    register poutput[5];
    register i[5];
    register j[3];
    register k[3];
    register address[32];

    ready = 1;
    wait(start);
    ready = 0;

    call init();

    delay;
    profile = 1;

    pimage = 0;
    parray = 0;
    pcoeff = 0;
    poutput = 0;

    i = 0;
    while (i < 4)
    {
        j = 0;
        while (j < 4)
        {
            pcoeff = 0;
            parray = i*6+j;
            parray2 = parray + 6;
            parray3 = parray + 12;

            outval = 0;

            call accumulate(pcoeff, parray);
            call accumulate(pcoeff, parray);
            call accumulate(pcoeff, parray);
            delay;
            call accumulate(pcoeff, parray2);
            call accumulate(pcoeff, parray2);
            call accumulate(pcoeff, parray2);
            delay;
            call accumulate(pcoeff, parray3);
            call accumulate(pcoeff, parray3);
            call accumulate(pcoeff, parray3);

            #address = 400 + poutput;
            #call mem_out(address, outval);
            #delay;
            mem_addr1 = 400 + poutput;
            mem_wr1 = 1;
            mem_data_out1 = outval;
            poutput = poutput + 1;

            j = j + 1;
        }

        i = i + 1;
    }

    delay;
    profile = 0;
}
```

B.12 iir_one_biquad

```
macro init()
```

```
{
    call mem_out1(100, 0x2746);    # x = 7

    call mem_out1(200, 0x2845);    # w1 = 7
    call mem_out2(201, 0x6946);    # w2 = 7

    call mem_out1(300, 0x1389);    # b0 = 7
    call mem_out2(301, 0x0586);    # b1 = 7
    call mem_out1(302, 0x9547);    # b2 = 7

    call mem_out1(400, 0x0215);    # a1 = 7
    call mem_out2(401, 0x6453);    # a2 = 7
}

macro get_data()
{
    call mem_in1(100, x);

    call mem_in1(200, w1);
    call mem_in2(201, w2);

    call mem_in1(300, b0);
    call mem_in2(301, b1);
    call mem_in1(302, b2);

    call mem_in1(400, a1);
    call mem_in2(401, a2);
}

module iir_one_biquad (
    in    start,
    out    ready,
    out   opa_i[16],        # fpu operand A
    out   opb_i[16],        # fpu operand B
    out   fpu_op_i[3],    # fpu operation
    in    output_o[16],    # fpu result
    out   mem_addr1[32],    # Memory address
    out    mem_data_out1[16],# Memory data out
    in    mem_data_in1[16],    # Memory data in
    out    mem_wr1,        # Memory write enable
    out   mem_addr2[32],    # Memory address
    out    mem_data_out2[16],# Memory data out
    in    mem_data_in2[16],    # Memory data in
    out    mem_wr2,        # Memory write enable
    out    profile        # profile enable
)
{
    register x[16];
    register w1[16];
    register w2[16];
    register b0[16];
    register b1[16];
    register b2[16];
    register a1[16];
    register a2[16];
    register y[16];
    register w[16];
    register temp1[16];

    ready = 1;
    wait(start);
    ready = 0;

    call init();
    call get_data();

    delay;
```

```
    profile = 1;

    call fpu_mult(a1, w1, temp1);
    call fpu_sub(x, temp1, w);

    call fpu_mult(a2, w2, temp1);
    call fpu_sub(w, temp1, w);

    call fpu_mult(b0, w, y);

    call fpu_mult(b1, w1, temp1);
    call fpu_add(y, temp1, y);

    call fpu_mult(b2, w2, temp1);
    call fpu_add(y, temp1, y);

    w2 = w1;
    w1 = w;

    delay;
    profile = 0;
}
```

B.13 iir_n_biquads

```
macro init_array(start_addr, num_vals, init_val)
{
    wi1 = 0;
    while (wi1 < num_vals)
    {
        address = start_addr + wi1;
        call mem_out1(address, init_val);
        wi1 = wi1 + 1;
    }
}

macro init()
{
    call init_array(100, 20, 0x05AD);    # coeff[i] = 7
    call init_array(200, 8, 0);        # wi[i] = 0

    call mem_out1(300, 0x0315);            # x = 1
}

module iir_n_biquads (
    in    start,
    out    ready,
    out   opa_i[16],        # fpu operand A
    out   opb_i[16],        # fpu operand B
    out   fpu_op_i[3],    # fpu operation
    in    output_o[16],    # fpu result
    out   mem_addr1[32],    # Memory address
    out    mem_data_out1[16],# Memory data out
    in    mem_data_in1[16],    # Memory data in
    out    mem_wr1,        # Memory write enable
    out   mem_addr2[32],    # Memory address
    out    mem_data_out2[16],# Memory data out
    in    mem_data_in2[16],    # Memory data in
    out    mem_wr2,        # Memory write enable
    out    profile        # profile enable
)
{
    register i[3];
    register ci[4];
    register wi1[5];
    register wi2[5];
```

```
register coeff[16];
register x[16];
register w[16];
register wi1_val[16];
register wi2_val[16];
register y[16];
register temp1[16];
register address[32];

ready = 1;
wait(start);
ready = 0;

call init();
call mem_in1(300, x);

delay;
profile = 1;

y = x;
ci = 0;
wi1 = 0;
wi2 = 1;
i = 0;
while ( i < 4)
{
    mem_addr1 = 100 + ci;
    mem_wr1 = 0;
    mem_addr2 = 200 + wi1;
    mem_wr2 = 0;
    delay;
    coeff = mem_data_in1;
    wi1_val= mem_data_in2;

    ci = ci + 1;

    call fpu_mult(mem_data_in1, mem_data_in2, temp1);

    mem_addr1 = 100 + ci;
    mem_wr1 = 0;
    mem_addr2 = 200 + wi2;
    mem_wr2 = 0;
    delay;
    coeff = mem_data_in1;
    wi2_val= mem_data_in2;
    ci = ci + 1;

    call fpu_sub(y, temp1, w);

    call fpu_mult(coeff, wi2_val, temp1);
    call fpu_sub(w, output_o, w);

    mem_addr1 = 100 + ci;
    mem_wr1 = 0;
    delay;
    coeff = mem_data_in1;
    ci = ci + 1;

    call fpu_mult(mem_data_in1, w, y);

    mem_addr1 = 100 + ci;
    mem_wr1 = 0;
    delay;
    coeff = mem_data_in1;
    ci = ci + 1;

    call fpu_mult(mem_data_in1, wi1_val, temp1);
```

```
        call fpu_add(y, output_o, y);

        mem_addr1 = 100 + ci;
        mem_wr1 = 0;
        mem_addr2 = 200 + wi2;
        mem_wr2 = 1;
        mem_data_out2 = wi1_val;
        delay;
        coeff = mem_data_in1;
        ci = ci + 1;

        call fpu_mult(mem_data_in1, wi2_val, temp1);
        call fpu_add(y, output_o, y);

        wi2 = wi2 + 1;
        mem_addr1 = 200 + wi1;
        mem_wr1 = 1;
        mem_data_out1 = w;
        wi1 = wi1 + 1;
        i = i + 1;
    }

    delay;
    profile = 0;
}
```

B.14 lms

```
macro init()
{
    call mem_out1(400, 0x9475);        # d = 7
    call mem_out2(500, 0xE7A3);        # x0 = 8
    call mem_out1(600, 0x00E9);        # delta = 1
}

macro get_data()
{
    call mem_in1(400, d);
    call mem_in2(500, x0);
    call mem_in1(600, delta);
}

macro initxh(offset)
{
    address = 100 + offset;
    call mem_out1(address, 0x0010);        # x_i = 1
    address = 200 + offset;
    call mem_out2(address, 0x0010);        # h_i = 1
}

macro accumulate(xi, hi, xi2)
{
    mem_addr1 = 100 + xi2;
    mem_wr1 = 0;
    call fpu_mult(X, H, temp1);
    X = mem_data_in1;
    xi2 = xi2 - 1;

    call fpu_add(output_o, Y, Y);

    mem_addr1 = 100 + xi;
    mem_wr1 = 1;
    mem_data_out1 = X;
    xi = xi - 1;

    mem_addr1 = 100 + xi;
    mem_wr1 = 0;
```

```
    mem_addr2 = 200 + hi;
    mem_wr2 = 0;
    delay;
    X = mem_data_in1;
    H = mem_data_in2;
    hi = hi - 1;
}

macro update(xi, hi)
{
    mem_addr1 = 100 + xi;
    mem_wr1 = 0;
    mem_addr2 = 200 + hi;
    mem_wr2 = 0;
    delay;
    X = mem_data_in1;
    H = mem_data_in2;
    xi = xi + 1;
    hi = hi + 1;

    call fpu_mult(X, error, temp1);

    call fpu_add(output_o, H, H);

    mem_wr2 = 1;
    mem_data_out2 = output_o;
}

module lms (
    in    start,
    out    ready,
    out   opa_i[16],        # fpu operand A
    out   opb_i[16],        # fpu operand B
    out   fpu_op_i[3],    # fpu operation
    in    output_o[16],    # fpu result
    out   mem_addr1[32],    # Memory address
    out    mem_data_out1[16],# Memory data out
    in    mem_data_in1[16],    # Memory data in
    out    mem_wr1,        # Memory write enable
    out   mem_addr2[32],    # Memory address
    out    mem_data_out2[16],# Memory data out
    in    mem_data_in2[16],    # Memory data in
    out    mem_wr2,        # Memory write enable
    out    profile        # profile enable
)
{
    register d[16];
    register x0[16];
    register delta[16];
    register error[16];

    register X[16];
    register H[16];
    register Y[16];

    register xi[5];
    register xi2[5];
    register hi[5];
    register i[5];
    register address[32];
    register temp1[16];

    ready = 1;
    wait(start);
    ready = 0;

    call init();
```

```
    i  = 0;
    while (i  < 16)
    {
        call initxh(i);
        i = i + 1;
    }

    call get_data();

    delay;
    profile = 1;

    xi = 15;
    xi2 = 14;
    hi = 15;
    i = 0;
    Y = 0;
    mem_addr1 = 100 + xi;
    mem_wr1 = 0;
    mem_addr2 = 200 + hi;
    mem_wr2 = 0;
    delay;
    X = mem_data_in1;
    H = mem_data_in2;
    while (i < 15)
    {
        call accumulate(xi, hi, xi2);
        i = i + 1;
    }

    call accumulate(xi, hi, xi2);
    address = 100 + xi;
    call mem_out1(address, x0);

    call fpu_sub(d, Y, temp1);
    call fpu_mult(output_o, delta, error);

    i  = 0;
    while (i  < 16)
    {
        call update(xi, hi);
        i = i + 1;
    }

    delay;
    profile = 0;
}
```

Bibliography

[1] N. Agarwal and N. Dimopoulos. Using CoDeL to rapidly prototype network processsor extensions. In *SAMOS IV: International Symposium on Systems, Architectures, Modeling and Simulation*, pages 333–342, November 2004.

[2] N. Agarwal and N. Dimopoulos. Power-efficient rapid system prototyping using CoDeL: The 2D DWT using lifting. In *PacRim 2005: IEEE Pacific Rim Conference on Communications, Computers and signal Processing*, pages 550–553, August 2005.

[3] N. Agarwal and N. Dimopoulos. Rapidly prototyping DSP extensions using CoDeL: the DWT using lifting. In *CCECE 2005: Canadian Conference on Electrical and Computer Engineering*, pages 802–805, May 2005.

[4] N. Agarwal and N. Dimopoulos. Efficient automated clock gating using CoDeL. In *SAMOS VI: International Symposium on Systems, Architectures, Modeling and Simulation*, pages 79–88, July 2006.

[5] N. Agarwal and N. Dimopoulos. Power efficient rapid hardware development using codel and automated clock gating. In *ISCAS 2006: IEEE International Symposium on Circuits and Systems*, pages 5310–5313, May 2006.

[6] N. Agarwal and N. Dimopoulos. Automated power gating of registers using CoDeL and FSM branch prediction. In *SAMOS VII: International Symposium on Systems, Architectures, Modeling and Simulation*, pages 294–303, July 2007.

[7] N. Agarwal and N. Dimopoulos. A DSPstone benchmark of CoDeL's automated clock gating platform. In *ISVLSI 2007: IEEE Computer Society Annual Symposium on VLSI*, pages 508–509, March 2007.

[8] N. Agarwal and N. Dimopoulos. High level FSMD design and automated clock gating using CoDeL. In *PacRim 2007: IEEE Pacific Rim Conference on Communications, Computers and signal Processing*, August 2007.

[9] N. Agarwal and N. Dimopoulos. Towards automated power gating of registers using CoDeL. In *ISCAS 2007: IEEE International Symposium on Circuits and Systems*, pages 1629–1632, May 2007.

[10] N. Agarwal and N. Dimopoulos. FSMD partitioning for low power using ILP. In *ISVLSI 2008: IEEE Computer Society Annual Symposium on VLSI*, pages 63–68, April 2008.

[11] N. Agarwal and N. Dimopoulos. FSMD partitioning for low power using simulated annealing. In *ISCAS 2008: IEEE International Symposium on Circuits and Systems*, pages 1244–1247, May 2008.

[12] N. Agarwal and N. Dimopoulos. High-level FSMD design and automated clock gating with CoDeL. *Canadian Journal of Electrical and Computer Engineering*, 33(1):31–38, Winter 2008.

[13] N. Agarwal and N. Dimopoulos. Towards automated FSMD partitioning for low power using simulated annealing. In *SAMOS IX: International Symposium on Systems, Architectures, Modeling and Simulation*, July 2009.

[14] A. V. Aho and J. D. Ullman. *Principles of compiler design*. Addison-Wesley, Reading, Massachusetts, 1977.

[15] P. Alexander and D. Barton. A tutorial introduction to rosetta. In *HDLCon 2001: Hardware Description Languages Conference*, March 2001.

[16] P. Babighian, L. Benini, and E. Macii. A scalable algorithm for RTL insertion of gated clocks based on ODCs computation. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, 24(1):29–42, January 2005.

[17] P. Bellows and B. Hutchings. JHDL - an HDL for reconfigurable systems. In *IEEE Symposium on FPGAs for Custom Computing Machines*, pages 175–184, March 1998.

[18] L. Benini, P. Siegel, and G. D. Micheli. Saving power by synthesizing gated clocks for sequential circuits. *IEEE Design and Test of Computers*, 11(4):32–40, December 1994.

[19] J. Bisschop and M. Roelofs. *AIMMS - User's Guide*. Lulu.com, 2006.

[20] D. C. Black and J. Donovan. *SystemC: From the Ground Up*. Springer, 2004.

[21] S. Borkar. Design challenges of technology scaling. *IEEE Micro*, 19(4):23–29, 1999.

[22] J. Bromley. Synthesizable VHDL fixed point arithmetic package. http://www.doulos.com/knowhow/vhdl_designers_guide/models/fp_arith/, 2006.

[23] T. D. Burd and R. W. Brodersen. Processor design for portable systems. *Journal of VLSI Signal Processing Systems*, 13(2-3):203–221, 1996.

[24] O. Cadenas and G. Megson. Power performance with gated clocks of a pipelined cordic core. In *5th International Conference on ASIC*, volume 2, pages 1226–1230, October 2003.

[25] N. Chabini and W. Wolf. Reducing dynamic power consumption in synchronous sequential digital designs using retiming and supply voltage scaling. *IEEE Transactions on Very Large Scale Integration (VLSI) Systems*, 12(6):573–589, June 2004.

[26] H. Chang and S. S. Sapatnekar. Full-chip analysis of leakage power under process variations, including spatial correlations. In *DAC 2005: Design Automation Conference*, pages 523–528, June 2005.

[27] J. Chang and M. Pedram. Energy minimization using multiple supply voltages. In *ISLPED 1996: International Symposium on Low Power Electronics and Design*, pages 157–162, August 1996.

[28] X. Chang, M. Zhang, G. Zhang, Z. Zhang, and J. Wang. Adaptive clock gating technique for low power IP core in SoC design. In *ISCAS 2007: IEEE International Symposium on Circuits and Systems*, pages 2120–2123, May 2007.

[29] D. Duarte, Y. Tsai, N. Vijaykrishnan, and M. J. Irwin. Evaluating run-time techniques for leakage power reduction. In *ASP-DAC 2002: Asia and South Pacific Design Automation Conference*, pages 31–38, January 2002.

[30] K. Flautner, N. S. Kim, S. Martin, D. Blaauw, and T. Mudge. Drowsy caches: simple techniques for reducing leakage power. In *ISCA 2002: International Symposium on Computer Architecture*, pages 148–157, May 2002.

[31] D. W. Franke and M. K. Purvis. Hardware/software codesign: a perspective. In *Software Engineering, 1991. Proceedings., 13th International Conference on*, pages 344–352, May 1991.

[32] J. Friedrich, B. McCredie, N. James, B. Huott, B. Curran, E. Fluhr, G. Mittal, E. Chan, Y. Chan, D. Plass, Sam Chu, Hung Le, L. Clark, J. Ripley, S. Taylor, J. Dilullo, and M. Lanzerotti. Design of the Power6 microprocessor. In

ISSCC 2007: IEEE International Solid-State Circuits Conference, pages 96–97, February 2007.

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

[34] D. Le Gall and A. Tabatabai. Subband coding of digital images using symmetric kernel filters and arithmetic coding techniques. In *ICASSP-88: International Conference on Acoustics, Speech, and Signal Processing*, pages 761–764, April 1988.

[35] F. Gao and J. P. Hayes. ILP-based optimization of sequential circuits for low power. In *ISLPED 2003: International Symposium on Low Power Electronics and Design*, pages 140–145, August 2003.

[36] D. Garrett, M. Stan, and A. Dean. Challenges in clockgating for a low power ASIC methodology. In *ISLPED 1999: International Symposium on Low Power Electronics and Design*, pages 176–181, August 1999.

[37] M. Gowan, L. Biro, and D. Jackson. Power considerations in the design of the Alpha 21264 microprocessor. In *DAC 1998: Design Automation Conference*, pages 726– 731, June 1998.

[38] J. R. Haigh, M. W. Wilkerson, J. B. Miller, T. S. Beatty, S. J Strazdus, and L. T. Clark. A low power 2.5-GHz 90-nm level 1 cache and memory management unit. *IEEE Journal of Solid State Circuits*, 40(5):1190–1199, May 2005.

[39] F. Hamzaoglu and M. R. Stan. Circuit-level techniques to control gate leakage for sub-100nm CMOS. In *ISLPED 2002: International Symposium on Low Power Electronics and Design*, pages 60–63, August 2002.

[40] P. Brinch Hansen. *On PASCAL compilers*. Prentice-Hall, Englewood Cliffs, New Jersey, 1985.

[41] Z. Hu, A. Buyuktosunoglu, V. Srinivasan, V. Zyuban, H. Jacobson, and P. Bose. Microarchitectural techniques for power gating of execution units. In *ISLPED 2004: International Symposium on Low Power Electronics and Design*, pages 32–37, August 2004.

[42] E. Hwang, F. Vahid, and Y. Hsu. FSMD functional partitioning for low power. In *DATE 1999: Design, Automation and Test in Europe Conference and Exhibition*, pages 22–28, March 1999.

[43] IEEE Standards. 1076-1993 IEEE standard VHDL language reference manual, 1993.

[44] IEEE Standards. 1364-1995 IEEE standard description language based on the Verilog hardware description language, 1995.

[45] Impulse Accelerated Technologies. Impulse c. http://www.impulsec.com/, 2009.

[46] M. C. Johnson, D. Somasekhar, Lih-Yih Chiou, and K. Roy. Leakage control with efficient use of transistor stacks in single threshold CMOS. *IEEE Transactions on Very Large Scale Integration (VLSI) Systems*, 10(1):1–5, February 2002.

[47] R. K. Kamat, S. A. Shinde, and V. G Shelake. *Unleash the System On Chip using FPGAs and Handel C*. Springer, 2009.

[48] J. Kao and A. Chandrakasan. Dual-threshold voltage techniques for low-power digital circuits. *IEEE Journal of Solid State Circuits*, 35(7):1009–1018, July 2000.

[49] T. Karnik, Y. Ye, J. Tschanz, L. Wei, S. Burns, V. Govindarajulu, V. De, and S. Borkar. Total power optimization by simultaneous dual-Vt allocation

and device sizing in high performance microprocessors. In *DAC 2002: Design Automation Conference*, pages 486–491, June 2002.

[50] H. K. Kim and R.V. Cox. A bitstream-based front-end for wireless speech recognition on IS-136 communications system. *IEEE Transactions on Speech and Audio Processing*, 9(5):558–568, July 2001.

[51] N. S. Kim, T. Austin, D. Blaauw, T. Mudge, K. Flautner, J. S. Hu, M. J. Irwin, M. Kandemir, and V. Narayanan. Leakage current: Moore's law meets static power. *IEEE Computer*, 36(12):68–75, 2003.

[52] S. Kumar, J. H. Aylor, B. W. Johnson, and W. A. Wulf. A framework for hardware/software codesign. *IEEE Computer*, 26(12):39–45, December 1993.

[53] V. Kursun and E. G. Friedman. Sleep switch dual threshold voltage domino logic with reduced standby leakage current. *IEEE Transactions on Very Large Scale Integration (VLSI) Systems*, 12(5):485–496, May 2004.

[54] T. Lang, E. Musoll, and J. Cortadella. Individual flip-flops with gated clocks for low power datapaths. *IEEE Transactions on Circuits and Systems Part II*, 44(6):507–516, June 1997.

[55] J. Liang and T. D. Tran. Fast multiplierless approximations of the dct with the lifting scheme. *IEEE Transactions on Signal Processing*, 49(12):3032–3044, December 2001.

[56] B. Liu, Y. Cai, Q. Zhou, J. Bian, and X. Hong. FSM decomposition for power gating design automation in sequential circuits. In *ASICON 2005: International Conference on ASIC*, pages 944–947, October 2005.

[57] Yung-Hsiang Lu, L. Benini, and G. De Micheli. Dynamic frequency scaling with buffer insertion for mixed workloads. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, 21(11):1284–1305, November 2002.

[58] H. S. Malvar, A. Hallapuro, M. Karczewicz, and L. Kerofsky. Low-complexity transform and quantization in H.264/AVC. *IEEE Transactions on Circuits and Systems for Video Technology*, 13(7):598–603, 2003.

[59] S. Manne, A. Klauser, and D. Grunwald. Pipeline gating: Speculation control for energy reduction. In *ISCA 1998: International Symposium on Computer Architecture*, pages 132–141, June 1998.

[60] Mentor Graphics. Catapult synthesis. http://www.mentor.com/products/esl/high_level_synthes 2009.

[61] J. L. Mitchell, W. B. Pennebaker, C. E. Fogg, and D. J. Legall, editors. *MPEG Video Compression Standard*. Chapman & Hall, Ltd., London, UK, 1996.

[62] J. Montanaro, R.T. Witek, K. Anne, A.J. Black, E.M. Cooper, D.W. Dobberpuhl, P.M. Donahue, J. Eno, W. Hoeppner, D. Kruckemyer, T.H. Lee, P.C.M. Lin, L. Madden, D. Murray, M.H. Pearce, S. Santhanam, K.J. Snyder, R. Stehpany, and S.C. Thierauf. A 160-MHz, 32-b, 0.5-W CMOS RISC microprocessor. *IEEE Journal of Solid-State Circuits*, 31(11):1703–1714, Nov 1996.

[63] S. Mutoh, T. Douseki, Y. Matsuya, T. Aoki, S. Shigematsu, and J. Yamada. 1-V power supply high-speed digital circuit technology with multithreshold-voltage CMOS. *IEEE Journal of Solid-State Circuits*, 30(8):847–854, August 1995.

[64] S. Narayan, F. Vahid, and D. D. Gajski. System specification with the SpecCharts language. *IEEE Design and Test of Computers*, 9(4):6–13, December 1992.

[65] G. Palumbo, F. Pappalardo, and S. Sannella. Evaluation on power reduction applying gated clock approaches. In *ISCAS 2002: IEEE International Symposium on Circuits and Systems*, volume 4, pages 85–88, May 2002.

[66] W. B. Pennebaker and J. L. Mitchell. *JPEG Still Image Data Compression Standard*. Kluwer Academic Publishers, Norwell, MA, USA, 1992.

[67] R. L. Plackett and J. P. Burman. The design of optimum multifactorial experiments. *Biometrika*, 33(4):305–325, June 1946.

[68] M. Powell, S. Yang, B. Falsafi, K. Roy, and T. N. Vijaykumar. Gated-Vdd: a circuit technique to reduce leakage in deep-submicron cache memories. In *ISLPED 2000: International Symposium on Low Power Electronics and Design*, pages 90–95, July 2000.

[69] R. Pratap. *Getting Started with MATLAB 7: A Quick Introduction for Scientists and Engineers*. Oxford University Press, USA, 2005.

[70] M. Rabbani and R. Joshi. An overview of the JPEG2000 still image compression standard. *Signal Processing: Image Communication Journal*, 17(1):3–48, 2002.

[71] N. Raghavan, V. Akella, and S. Bakshi. Automatic insertion of gated clocks at register transfer level. In *Twelfth International Conference On VLSI Design*, pages 48–54, January 1999.

[72] S. Rele, S. Pande, S. Onder, and R. Gupta. Optimizing static power dissipation by functional units in superscalar processors. In *CC 2002: International Conference on Compiler Construction*, pages 261–275, April 2002.

[73] T. Salonidis and V. Digalakis. Robust speech recognition for multiple topological scenarios of the gsm mobile phone system. In *ICASSP-98: International Con-*

ference on Acoustics, Speech, and Signal Processing, volume 1, pages 101–104, May 1998.

[74] D. Shin, A. Gerstlauer, R. Domer, and D. D. Gajski. An interactive design environment for C-based high-level synthesis of RTL processors. *IEEE Transactions on Very Large Scale Integration (VLSI) Systems*, 16(4):466–475, April 2008.

[75] R. Sivakumar, V. Dimakopoulos, and N. Dimopoulos. CoDeL: A rapid prototyping environment for the specification and automatic synthesis of controllers for multiprocessor interconnection networks. In *SAMOS III: International Symposium on Systems, Architectures, Modeling and Simulation*, pages 58–63, July 2003.

[76] R. Sivakumar, V.V. Dimakopoulos, and N.J. Dimopoulos. A rapid prototyping environment for the specification and automatic synthesis of controllers for interconnection routers. In *Asilomar Conference on Signals, Systems and Computers*, pages 193–198, October 1995.

[77] S. Sutherland, S. Davidmann, P. Flake, and P. Moorby. *SystemVerilog for Design Second Edition: A Guide to Using SystemVerilog for Hardware Design and Modeling*. Springer, July 2006.

[78] W. Sweldens. The lifting scheme: A new philosophy in biorthogonal wavelet constructions. In *SPIE 2569*, pages 68–79, 1995.

[79] Texas Instruments. DSP selection guide, 2009. http://www.ti.com.

[80] K. Usami and M.Horowitz. Clustered voltage scaling technique for low-power design. In *ISLPED 1995: International Symposium on Low Power Electronics and Design*, pages 3–8, April 1995.

[81] F. Vahid, S. Narayan, and D. D. Gajski. SpecCharts: a VHDL front-end for embedded systems. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, 14(6):694–706, June 1995.

[82] J. Vandekerckhove. General simulated annealing algorithm. http://www.mathworks.com/matlabcentral/fileexchange/10548, 2009.

[83] Q. Wang and S. Roy. Power minimization by clock root gating. In *ASP-DAC 2003: Asia and South Pacific Design Automation Conference*, pages 249–254, January 2003.

[84] N. H. E. Weste and K. Eshraghian. *Principles of CMOS VLSI design, a systems perspective*. Addison-Wesley, Reading, Massachusetts, second edition, 1993.

[85] R. Xu, D. Mossé, and R. Melhem. Minimizing expected energy consumption in real-time systems through dynamic voltage scaling. *ACM Transactions on Computer Systems (TOCS)*, 25(4), 2007.

[86] V. Zivojnovic, J. Martinez, C. Schlger, and H. Meyr. DSPstone: A DSP-oriented benchmarking methodology. In *ICSPAT 1994: International Conference on Signal Processing Applications and Technology*, October 1994.