Power Management
for
Deep Submicron Microprocessors

by

Ahmed Youssef

A thesis
presented to the University of Waterloo
in fulfillment of the
thesis requirement for the degree of
Doctor of Philosophy
in
Electrical and Computer Engineering

Waterloo, Ontario, Canada, 2008

I hereby declare that I am the sole author of this thesis. This is a true copy of the thesis, including any required final revisions, as accepted by my examiners.

I understand that my thesis may be made electronically available to the public.

Abstract

As VLSI technology scales, the enhanced performance of smaller transistors comes at the expense of increased power consumption. In addition to the dynamic power consumed by the circuits there is a tremendous increase in the leakage power consumption which is further exacerbated by the increasing operating temperatures. The total power consumption of modern processors is distributed between the processor core, memory and interconnects. In this research two novel power management techniques are presented targeting the functional units and the global interconnects.

First, since most leakage control schemes for processor functional units are based on circuit level techniques, such schemes inherently lack information about the operational profile of higher-level components of the system. This is a barrier to the pivotal task of predicting standby time. Without this prediction, it is extremely difficult to assess the value of any leakage control scheme. Consequently, a methodology that can predict the standby time is highly beneficial in bridging the gap between the information available at the application level and the circuit implementations.

In this work, a novel Dynamic Sleep Signal Generator (DSSG) is presented. It utilizes the usage traces extracted from cycle accurate simulations of benchmark programs to predict the long standby periods associated with the various functional units. The DSSG bases its decisions on the current and previous standby state of the functional units to accurately predict the length of the next standby period. The DSSG presents an alternative to Static Sleep Signal Generation (SSSG) based on static counters that trigger the generation of the sleep signal when the functional units idle for a prespecified number of cycles.

The test results of the DSSG are obtained by the use of a modified RISC superscalar processor, implemented by SimpleScalar, the most widely accepted open source vehicle for architectural analysis. In addition, the results are further verified by a Simultaneous Multithreading simulator implemented by SMTSIM. Leakage saving results shows an increase of up to 146% in leakage savings using the DSSG versus the SSSG, with an accuracy of 60-80% for predicting long standby periods.

Second, chip designers in their effort to achieve timing closure, have focused on achieving the lowest possible interconnect delay through buffer insertion and routing techniques. This approach, though, taxes the power budget of modern ICs, especially those intended for wireless applications. Also, in order to achieve more functionality, die sizes are constantly increasing. This trend is leading to an increase in the average global interconnect length which, in turn, requires more buffers to achieve timing closure.

Unconstrained buffering is bound to adversely affect the overall chip performance, if the power consumption is added as a major performance metric. In fact, the number of global interconnect buffers is expected to reach hundreds of thousands to achieve an appropriate timing closure.

To mitigate the impact of the power consumed by the interconnect buffers, a power-efficient multi-pin routing technique is proposed in this research. The problem is based on a graph representation of the routing possibilities, including buffer insertion and identifying the least power path between the interconnect source and set of sinks.

The novel multi-pin routing technique is tested by applying it to the ISPD and IBM benchmarks to verify the accuracy, complexity, and solution quality. Results obtained indicate that an average power savings as high as 32% for the 130-nm technology is achieved with no impact on the maximum chip frequency.

Acknowledgements

All praise is due to Allah for guiding me throughout my life and giving me the ability to complete this work. I am at a loss of words to express my gratitude to my family for their continuous love and support.

This thesis would not be possible without the support of many individuals, to whom I would like to express my gratitude. I will always be indebted to my supervisors Prof. Mohamed Elmasry and Prof. Mohab Anis, for their key role in my development as a person and as a researcher. They offered me support, encouragement, guidance, and most importantly trust. Their input and guidance was invaluable to the quality and contribution of the work presented in this thesis, as well as in other publications. Their trust and support was instrumental in giving me confidence to achieve many accomplishments.

I would like also to thank many faculty members of the University of Waterloo, most notably my committee members, Prof. John Yeow, Prof. Mark Aagaard, and Prof. Manoj Sachdev, for their valuable input and suggestions. I would like also to thank Prof. Shawki Areibi and Prof. Mohamed Zahran for valuable discussions and ideas. I wish to thank many of my colleagues at the VLSI Lab, especially Mohamed Elsaid, Muhammad Nummer, Mohamed Elgebaly, Hassan Hassan, Ayman Ismail, and Mohamed Hassan, for valuable discussions and feedback. I am grateful to Wendy Boles for her help on administrative issues and to Phil Regier for his great computing resources support.

I would like to thank NSERC for their financial support. I would like to thank Hazem Shehata, Shady Shehata, Mohamed El-Abd, Ismael El-Samahy, Mohamed El-Dery, and Hatem Zeineldin for being such great friends.

My wife, Wessam, shared with me every day throughout the course of this work. Her love, support, and understanding played a major role in helping me finish this thesis. I am also grateful to my sister, Amira, for her love, support, and encouragement.

My deepest gratitude to my mother and father for their ever continuous support, encouragement, and prayers. No words of appreciation could ever reward them for all they have done for me. I am, and will ever be, indebted to them for all achievements in my life.

Dedication

To my wife Wessam, my sister Amira,
my Father Mohamed and Mother Eman.
with love and appreciation.

Contents

List of Tables

List of Figures

Glossary of Terms

BBV Basic Block Vector

BTB Branch Target Buffer

CISC Complex Instruction Set Computer

CPI Cycles per Instruction

CMOS Complementary Metal Oxide Semiconductor

DSM Deep Submicron

DSSG Dynamic Sleep Signal Generator

DVS Dynamic Voltage Scaling

DIBL Drain Induced Barrier Lowering

EDA Electronic Design Automation

FO4 Fanout of 4

FP Floating-point Unit

FU Functional Unit

ID Instruction Decode

IE Instruction Execute

IF Instruction Fetch

ILP Instruction Level Parallelism

IPC Instructions per Cycle

ITRS International Technology Roadmap for Semiconductors

ISA Instruction Set Architecture

MTCMOS MultiThreshold CMOS (Power Gating)

NOP No Operation

NP-hard Non deterministic polynomial hard problems are a subset of combinatorial problems which are not solvable in polynomial time

O-o-O Out-of-Order

PDP Power Delay Product

RAS Return Address Stack

RAW Read after Write (Data Hazard)

RF Register File

RISC Reduced Instruction Set Computer

ROB Reorder Buffer

RUU Register Update Unit

SMT Simultaneous MultiThreading

SOC System on Chip

SSSG Static Sleep Signal Generator

TK/NT Taken versus Not Taken branch

TLB Translation Look-aside Buffer

VTCMOS Variable Threshold CMOS

WAR Write after Read (Data Hazard)

WAW Write after Write (Data Hazard)

Chapter 1

Introduction

Microprocessors determine the sophistication of many consumer electronics applications [1]. Particularly, portable equipment not only requires more processing power, but also a very stringent power envelope. Accordingly, modern microprocessors must handle increased throughput, complex algorithms, and real time requirements. At the same time, though, the VLSI complexity must remain within the bounds of low area and power consumption.

1.1 Motivation

As the demand for faster and more complex applications increases, the semiconductor industry has been aggressively scaling the silicon technology in order to match the continuous increase in chip requirements. Dynamic and leakage power has consistently increased with every technology generation. The processor core, memory and global interconnects power consumption constitute the majority of the power budget of modern processors.

In this research, two power management techniques are proposed to handle leakage power consumption in the processor core as well as the dynamic power consumption of the global interconnects. Section 1.1.1 details the motivation behind the implementation of a standby predictor for managing leakage power for processor cores, while Section 1.1.2 presents the motivation behind a low power multi-pin global routing methodology that handles the dynamic power consumption of the processor interconnects.

1.1.1 Leakage Power in Processor Functional Units

The continuous reduction of the transistor dimensions, entailed by technology scaling, has created many challenges to chip designers. One of the major challenges is the reduction of the supply voltage due to smaller gate oxides that cannot withstand the traditional 3.3V and 5V supplies [2]. Reducing the supply voltage necessitates the reduction of the transistor threshold to maintain an adequate overdrive voltage. In turn, the reduction of the threshold voltage increases the transistor's subthreshold conduction, which translates into an increase in leakage power consumption.

Due to the significant increase in leakage power, research has focused on the reduction of the core leakage [3–5]. Research in [3] shows that managing leakage power in the microprocessor core can save up to 18% of the total power when power gating and adaptive body biasing is employed.

To deal with the increased leakage power and to maintain the complexity of modern processors, low power techniques are necessary throughout the design process. If the process is divided into levels, as depicted in Fig. 1.1, it is imperative to consider the combined effect of all these levels in order to exploit the full potential of leakage management [6]. Many circuit and system-level techniques have been proposed to deal with leakage power. However, very few of them focus on crossing the level boundaries, limiting their leakage saving capabilities.

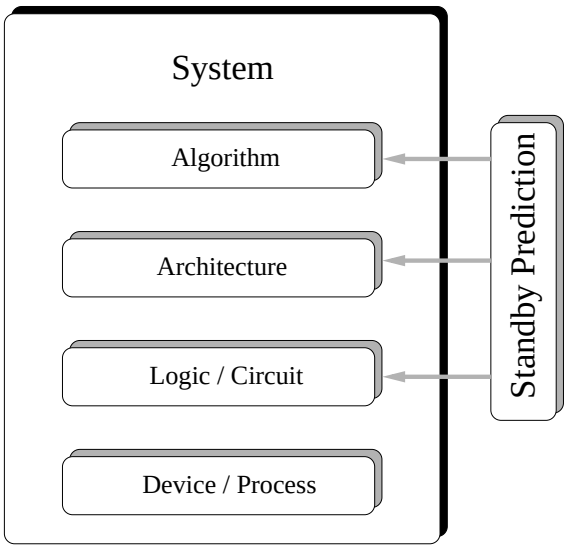


Figure 1.1: System design space [6].

Circuit level leakage control techniques should reduce the leakage power, and ensure that the intended logic operation is performed correctly. However, these circuit tech-

niques are characterized by their inability to predict the circuit usage patterns. These usage patterns can be obtained by answering questions such as how many times was the circuit accessed during a certain number of cycles?, how many times were the neighboring circuit blocks used?, and was there any regularity in the usage of the circuit?. Knowing the answers to these questions can significantly enhance the design process.

To deduce the usage patterns, the architecture needs to keep track of the standby periods of the various circuit blocks. These periods, defined as the length of time that the circuit stays idle, can be used as indicators of the circuits' usage profiles. Fortunately, for microprocessors, and programmable DSPs in particular, the information about the standby periods can be obtained from the executed code. Accordingly, a standby prediction methodology is expected to enhance the performance of the various circuit level techniques by supplying the circuits with standby information extracted from the algorithm.

The objective of this research is to introduce the concept of standby prediction by developing a Dynamic Sleep Signal Generator (DSSG). The new DSSG uses the high-level information found in the running application code to predict the standby profile of the processor functional units.

Contributions

The following outlines the main contributions of the proposed DSSG

1. Contrary to compiler-based low leakage techniques that are bound to a specific Instruction Set Architecture (ISA), the DSSG depends on only the information about current and previous standby periods.

2. Phase extraction techniques that are concerned with the coarse granularity prediction of program phases are not adequate for the task of low leakage management. These techniques are generally slow and perform complex tasks, prohibiting the techniques' application in small-scale leakage management. In contrast, the DSSG focuses on a fine granularity analysis (a few hundred cycles) of the profile information.

3. Current circuit techniques for leakage power reduction depend on detecting when the circuit is actually in standby, which leaves the techniques prone to erroneous decisions regarding very short standby periods, since the techniques lack the ability to predict the length of the period ahead of time [3–5]. Accordingly, the proposed

DSSG should eliminate these short periods from the decision tree of these techniques

4. The proposed DSSG and the associated finite state machine are capable of tracking the executed program behavior across different time segments and predict the length of the standby periods accordingly. This leads to a high accuracy in asserting the sleep signal when it is most likely to achieve a net increase in the total power savings. This is accomplished with minimal power overhead.

5. The DSSG is a simple hardware based approach. This allows the incorporation of the DSSG in existing microprocessors (General purpose and embedded) with minimal design overheads.

6. The DSSG finite state machine does not rely on any memory like structure reducing the layout footprint and the overall power consumed to achieve the higher prediction accuracy targeted by the DSSG.

7. The DSSG exhibits low power consumption, in the order of 300 μW to achieve accuracies up to 80% in predicting the length of the standby period.

In summary, the principal objective of this part of the research is to devise an accurate prediction methodology that allows circuit designers to explore novel low leakage power management schemes.

1.1.2 Dynamic Power in Global Interconnects

Global interconnects are gradually dominating the performance of deep sub-micron chips. In fact, the number of interconnects and buffers for achieving timing closure are one of the primary challenges facing designers for sub-$90nm$ ICs. This is attributed to the continual increase in the number of logic blocks, due to the continuous shrinking of device dimensions. To mitigate the impact of the interconnects, designers have shifted their focus to interconnect centric designs, where the wire is the center of the chip design flow [7].

The interconnect problem is a multi-objective optimization problem, where delay, power, and routing are core objectives. Traditionally, delay and routing have been the focus of most optimization efforts [7]. However, the power consumption of the interconnects is becoming a crucial factor in determining the overall chip performance [8]. Research in [9] shows that global signaling nets can consume up to 21% of the total dynamic power [10].

To address the interconnect bottleneck, researchers have developed several subproblems that deal with the various aspects of the interconnects. These problems begin with the simple problem of *buffer insertion*, determining the number and the positions of buffers to minimize delay [11], and increase in complexity up to *Optimal Power Maze Routing*, where a combined routing and power optimization under relaxed delay constraints is introduced [12, 13].

Research in [12–19] focused on the low power interconnect problem through optimum buffer insertion. The main limitation facing many of the interconnect power minimization efforts is the lack of simultaneous optimization of the various interconnect performance metrics. As an example, the work in [14, 15] focuses on only single net optimization which inherently implies a net ordering effect. The net ordering effect means that for successful chip timing closure the nets has to be ordered according to their importance. This ordering limits the ability of the technique to find globally optimum solution. In the meantime, the research in [16–19] focuses on the optimum buffer insertion assuming prerouted nets. This in turn limits the ability of the technique to prevent congestion in general and especially buffer related congestion where the availability of buffer insertion locations is contested by several nets. Finally, to address some of the limitations, the work in [12, 13][1] simultaneously routes the interconnects while inserting the buffers to optimize for the power consumption. However, the effort in [12] focuses only on two-pin nets while the work in [13] is characterized with excessively long runtime.

On the other hand, analytical solutions to the power optimization problem [20–22] are also limited in their ability to simultaneously optimize the various metrics since they depend on mathematically differentiating some cost function with respect to one parameter. In addition, these techniques are incapable of accommodating the buffer blockage arising from the preplaced blocks.

In order to address the limitation of the previous techniques, a formulation for a power-efficient interconnect optimization technique is proposed in this work. The goal is to find a solution through the formulation of a Power-Efficient multi-pin Integer linear programming based global Routing Technique (*PIRT*). The new formulation simultaneously solves for a minimum power tree for each global net in the chip without affecting the chips maximum frequency requirements.

Contributions

The technical contributions of the newly formulated PIRT can be summarized as follows

[1] This work is the authors pervious contributions in the power optimal interconnect optimization.

1. Unlike previous approaches, the newly developed approach is capable of timing optimization, buffer insertion and power reduction simultaneously with routability consideration.

2. The optimization of power consumption and simultaneously accounting for the buffer blockage, which has not been considered in previous analytical formulations of the power optimization problem, is formulated.

3. The optimization of the power consumption without affecting the chip's maximum frequency.

4. The problem is formulated so that it is independent of the delay and the power models used, allowing for more flexibility in applying the new technique to scaled technologies.

5. PIRT is capable of simultaneously routing and power optimizing the chip with runtime less than 0.1 second per net.

In conclusion, the goal of the novel multi-pin power optimization methodology is to devise a fast, yet accurate technique to reduce the power consumption of global interconnects while maintaining the chip's maximum clock frequency.

1.2 Thesis Organization

To formulate the concept of standby prediction and the multi-pin global routing problem, it is important to explore relevant topics. Accordingly, in Chapter 2, the physical nature of the VLSI power consumption problem is described.

Chapter 3 presents the architecture of generic superscalar and SMT processors, exploring the various building blocks of state-of-the-art microprocessors. After a discussion of the concepts of power consumption and microprocessor architecture, the most popular circuit-level leakage control techniques are reviewed in Chapter 3. This is followed by a description of the various system-level techniques available for the reduction of microprocessor power consumption. The focus is on low-power pipelines and low-power software design. Chapter 3 is concluded by reviewing some of the profile prediction literature.

In Chapter 4 after the concept of standby prediction is introduced, the results of applying the DSSG to the SPEC2000 benchmark suite on a Superscalar processor are

given. These results are obtained by modifying the SimpleScalar performance simulator in order to extract the functional units' usage patterns.

Chapter 5 details the results of running the DSSG on an SMT architecture using the SMTSIM simulator to extract the functional units' usage patterns. Chapter 5 also presents the results of applying the SSSG and the DSSG on several microprocessor architectures.

Chapter 6 starts by reviewing the state-of-the-art techniques for power optimal interconnect optimizations. Then, the chapter presents the new formulation of the low power global routing problem (PIRT). In addition, the results of running PIRT on the IBM and ISPD floorplan benchmarks is discussed.

Lastly, the thesis is concluded by outlining the major contributions of this research and the potential enhancement in future work.

Chapter 2

CMOS Power Consumption

CMOS circuits are the foundation for most modern microprocessors. The CMOS technology offers high density, high performance with low power consumption. To tackle the microprocessor power management problem it is imperative is to explore the physical nature of CMOS power consumption. Accordingly, this Chapter presents the fundamentals behind the sources of power consumption in CMOS circuits.

2.1 Technology Scaling

The semiconductor industry has been aggressively pursuing an ever shrinking minimum feature size. The shrinking minimum feature size allowed for lower cost per function, higher frequencies and compact light-weight electronics [23].

Constant field scaling proposed by Dennard et al. in [24] entails the continuous reduction of the vertical and horizontal dimensions while simultaneously scaling the applied voltages to maintain constant electric fields across the smaller devices [25]. The constant electric field scaling ensures the reliability of the scaled devices by limiting hot carrier injection. Table 2.1 shows the constant field scaling rules for some device and chip parameters.

Due to the continuous increase in die sizes the dynamic power consumption is increasing with each technology node. Moreover, the leakage power consumption is also exponentially increasing due to the reduced threshold voltages. In fact it is projected that a huge increase in the power density will occur with each technology node [26, 27]. Fig. 2.1 presents the projected increase in chip's power density [26]. Due to the impact of the increased dynamic and leakage power consumption modeling and understanding the

Table 2.1: Constant field scaling

Technology Parameter	Scaling Factor
Device Dimension (t_{ox},L,W)	1/S
Voltage	1/S
Doping Concentration	S
Electric Field	1
Carrier velocity	1
Gate Capacitance	1/S
Drift Current	1/S
Circuit Delay	1/S
Dynamic Power Dissipation	1/S^2
Leakage Dissipation	exponential
Circuit Density ($\alpha 1/A$)	S^2
Power Density	1

nature of the CMOS power consumption is an important prerequisite before any further discussion of processor power management.

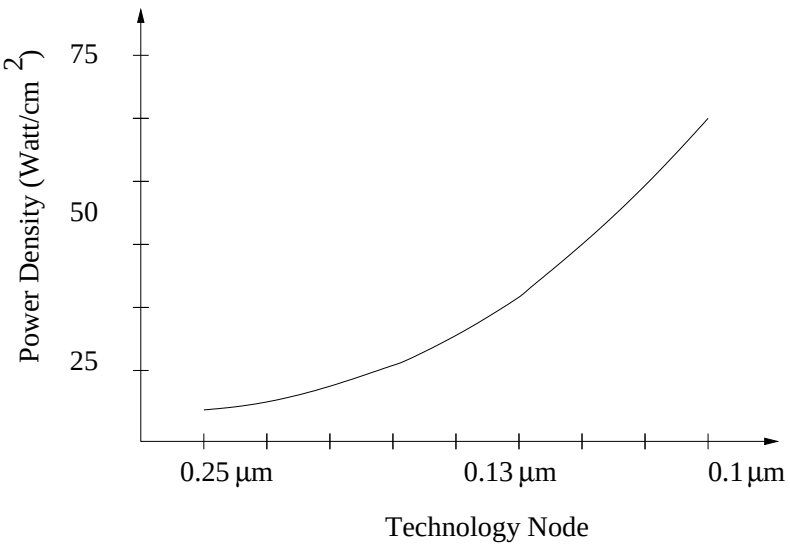


Figure 2.1: Trends of processors' power density [26].

2.2 Sources of Power Consumption

With the continuous scaling of CMOS technology, the semiconductor industry is slowly shifting towards more power centric designs. Typically, the total power dissipation of

CMOS circuits (P_{total}) is given by [11]

$$P_{total} = P_{switching} + P_{short-circuit} + P_{leakage}, \tag{2.1}$$

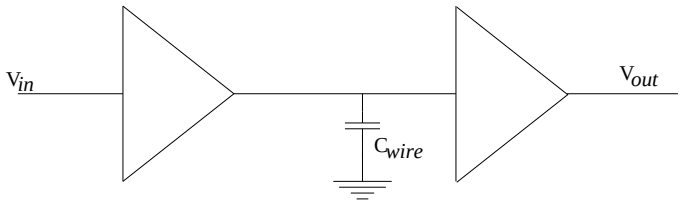
where $P_{switching}$ is the switching power, $P_{short-circuit}$ is the short-circuit power, and $P_{leakage}$ is the leakage power.

2.2.1 Switching Power

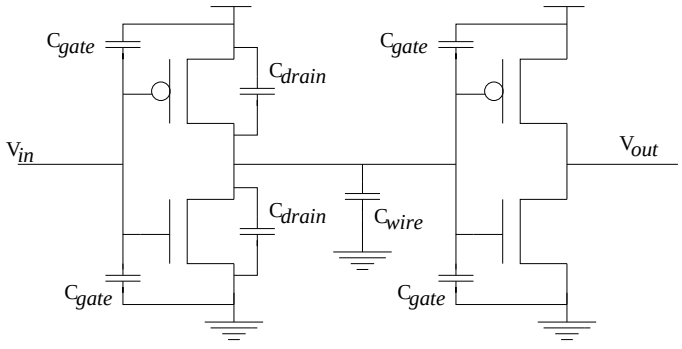
The switching power is the component of power consumption that is associated with the charging and discharging of the capacitances of the CMOS gate. A simple yet accurate estimation of the switching power of the gate is given by [6]

$$P_{switching} = \alpha V_{dd}^2 . f_{clk}(C_{gate} + C_{wire}), \tag{2.2}$$

where α is the switching factor of the gate, V_{dd} is the supply voltage, f_{clk} is the clock frequency, C_{gate} is drain diffusion capacitance at the output and the total input gate capacitances of the subsequent gates, and C_{wire} is the total wire capacitance at the fan-out of the gate. Fig. 2.2 presents the basic capacitive sources in a CMOS inverter.



(a)



(b)

Figure 2.2: Sources of output load capacitance.

It is interesting to note that the switching power consumption is quadratically dependant on the supply voltage [28]. This dependence motivates different approaches to reduce the switching power by scaling the supply voltage. However, the scaling of the supply voltage means an increase in the leakage power, since the reduction of the supply requires a corresponding reduction in the threshold voltage. This reduction in the threshold voltage causes an exponential increase in the subthreshold leakage current.

2.2.2 Short-Circuit Power

Due to the fact that for actual CMOS circuits there are finite rise and fall times the PMOS and NMOS transistors are briefly on, simultaneously. This results in short circuit current flowing due to the direct path through both transistors [11].

The average short circuit power can be calculated using

$$P_{short-circuit} = t_{sc} V_{DD} I_{peak} f_{clk} \tag{2.3}$$

where t_{sc} is the period of time when both devices are conducting, V_{DD} is the supply voltage, I_{peak} is the peak current flowing during the period of t_{sc}, and f_{clk} is the clock frequency

2.2.3 Leakage Power

The quadratic dependence of the switching power on the supply voltage can be utilized to reduce the switching power [28]. In fact, the reduction of the supply voltage is also bound to the scaling of the device's dimensions. For example, the reduction of the gate oxide thickness limits the value of the maximum electric field tolerated on the gate, which imposes an upper bound on the supply voltage. Nevertheless, reducing the supply voltage forces the reduction of the threshold voltage of the devices in order to maintain the overdrive voltage ($V_{GS} - V_{th}$). This leads to an exponential increase in the subthreshold leakage current [29].

Indeed, leakage power is expected to dominate the overall power consumption as the technology scales [28]. This is confirmed by the trend reflected in Fig. 2.3, where the leakage power will constitute more than 50% of the total power consumption for the technologies beyond $65nm$ if it is not properly addressed.

The leakage power consumption due to the subthreshold current can be calculated using the expression,

$$P_{leakage} = V_{dd} I_{sub}, \tag{2.4}$$

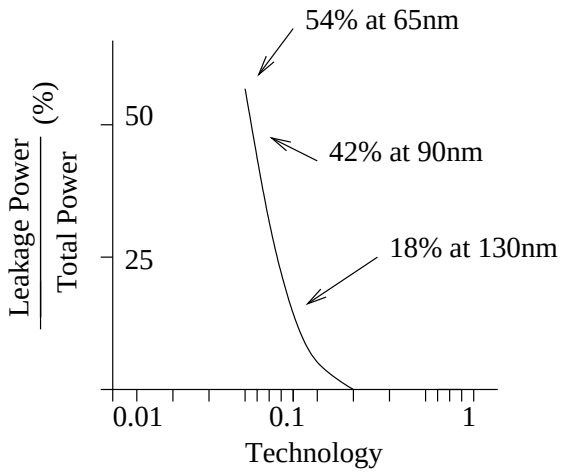


Figure 2.3: Leakage power consumption as a percentage of the total power for various technology nodes [29].

where I_{sub} is the subthreshold current which is computed by the following [30]:

$$I_{sub} = I_0 e^{(V_{gs}-V_{th})/mV_T}(1 - e^{-V_{ds}/V_T}), \tag{2.5}$$

where m is the subthreshold swing coefficient, V_{th} is the threshold voltage, and $V_T = KT/q$ is the thermal voltage.

Although this research focuses on subthreshold leakage control techniques, the following section describes briefly the various sources of leakage current in today's CMOS circuits.

2.2.3.1 Sources of Leakage Current

The various components of the leakage current can be summarized as follows [28]

- *PN junction reverse leakage*; this occurs due to the minority carrier drift near the depletion region, this current is usually small and can be neglected. However, if the electric field across the reverse biased PN junction reaches $10^6 V/m$, significant current will flow due to band to band tunneling [31].

- *Drain induced barrier lowering*; due to the relative size of the drain/source depletion width in the vertical dimension to the channel size, the source drain potential affects the band bending over a significant portion of the device width. Higher drain potential in effect reduces the transistor threshold, increasing its leakage [30].

- *Gate induced drain leakage*; this occurs at negative V_G and high V_D, which causes a high electric field under the gate drain overlap region causing significant band to band tunneling.

- *Gate oxide tunneling*; this is due to the tunneling of electrons through the gate oxide especially when the electric field is very high coupled with small oxide thickness.

- *Subthreshold leakage*; this is the current flowing between the source and the drain when the transistor is in the weak inversion and the gate voltage is below the threshold voltage. This component is exponentially dependant on the threshold voltage, currently making it the most critical component of leakage power consumption.

2.3 Managing CMOS Power Consumption

Several techniques have been envisioned to manage the increasing strain on the chip power budget. These techniques vary based on the time of application between design time, sleep mode and runtime power management schemes [11]. Table 2.2 summarizes some of the major vehicles of power management available to processor designers.

Table 2.2: Power management

	Design Time	Sleep Mode	Run Time
Dynamic Power	Lower V_{dd}, Multi-V_{dd}, Sizing, Logic optimization	Clock Gating	Dynamic Voltage Scaling
Leakage Power	Multi-V_{th}	VTCMOS, MTCMOS	VTCMOS

2.3.1 Dynamic Power Management

To reduce the dynamic power consumption of a microprocessor core chip designers resort to several techniques that focuses on manipulating the various parameters in equation 2.2. Transistor sizing and logic optimization targets the reduction of the total capacitance that is switched each cycle. Logic optimization can also help reduce the dynamic power consumption due to glitching.

Reducing the supply voltage as mentioned earlier quadratically influences the overall power consumption. Low-V_{dd} designs in portable and medical applications tremendously extends the battery lifetime at the expense of slower performance.

On the other hand Multi-V_{dd} designs allows for the reduction of the power consumption of non-critical components while maintaining the performance of mission critical logic. Multi-V_{dd} designs face challenging requirements with respect to their power delivery network, where multiple supply rails are required in addition to level converters between voltage islands.

Dynamic voltage scaling (DVS) allows the designers to scale the voltage down when the application requirements are low and increase it when the demand is high. Similar to Multi-V_{dd}, the power delivery and level converters are some of the limitation for DVS techniques. In addition to DVS, clock gating is a very effective technique to reduce the dynamic power when the logic is not utilized. However, clock gating does not tackle leakage power consumption.

Since both DVS and clock gating techniques act at the circuit level they require system level policies to ensure the power-performance trade off is maintained across various workload requirements.

2.3.2 Leakage Power Management

Multi-V_{th} technologies allow designers the flexibility of using low-threshold high-leakage devices to build the logic of critical paths while using high-threshold low-leakage devices in non-critical paths. Multi-V_{th} designs require careful attention to path selection since the slow down introduced to non-critical paths might cause them to become the critical paths in the design.

VTCMOS techniques allow the designer to change the threshold of the devices during runtime or sleep mode. Changing the threshold allows the logic to manage its leakage at the expense of some performance loss. MTCMOS designs allow for an aggressive shutdown of the supply network to limit the overall leakage. MTCMOS and VTCMOS design also require system level policies to ensure maximum performance.

In this research the dynamic sleep signal generator technique will target the management of leakage power in standby mode while the interconnect optimization effort targets the design time dynamic power minimization. In the following chapter, an overview of the state-of-the-art microprocessors' architectures is presented. In addition, a more detailed survey of the low leakage circuit and system-level techniques is conducted.

Chapter 3

Microprocessor Leakage Power Management

In order to establish the background necessary for the advanced leakage management technique proposed in this thesis, the study of the modern microprocessor architecture is crucial. In addition, modern leakage management techniques need to be investigated.

3.1 Microprocessor Architecture

State-of-the-art general purpose processors result from pursuing the latest techniques that can enhance the execution throughput. Microprocessors, from the software perspective, are designed to perform as a sequential engine,[1] capable of handling sequential sets of instructions and producing sequential output that matches the program order of instructions. This is simply described by considering pipelined processor architectures. These architectures divide the tasks, corresponding to the code execution, among a set of hardware blocks.

Single pipelined processors are especially useful for embedded applications, where the processors satisfy the computational requirements without the need for the more complicated superscalar architectures. In contrast to single pipelined processors, superscalar processers are capable of handling more than one instruction in each cycle. This nature allows the superscalar processor to increase its throughput by exploiting some of the inherent Instruction Level Parallelism (ILP) in the application code [1]. Finally, Si-

[1] This statement excludes explicitly parallel coding techniques that expect a multi-engine architecture. This multi-engine includes multi-processors and multi-core chips.

multaneous Multithreading (SMT) processors enhances the performance of superscalar processors by running multiple applications simultaneously.

3.1.1 Pipelined Processors

Pipelining is a means of fully utilizing the hardware capabilities. It allows for a slightly increased latency in the processor operation, while significantly increasing the processor throughput.

Fig. 3.1 illustrates a simple pipelined processor. Each stage is attached to the next by a latch that is capable of storing the results of stage n that are used by stage $(n + 1)$, while stage n is performing the next task.

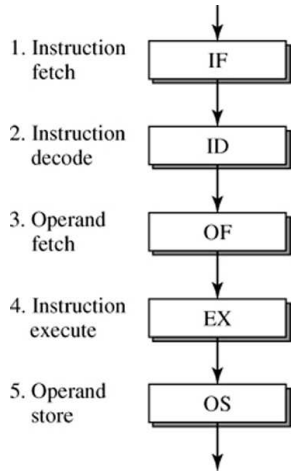


Figure 3.1: Pipelined processor [1].

Perfect pipelining requires that the executed code be divisible in a uniform set of steps that requires, approximately, similar execution times. Consequently, the latency of the various stages is expected to be approximately equal, ensuring that the pipelined stages are utilized to their maximum without any one stage completing the job ahead of the others. This is not naturally achievable, since some pipeline stages such as multipliers and dividers are inherently slower than other stages.

In addition to the divisibility, perfect pipelines assume that the code is generally repetitive. This translates into similar resource requirements for each executed instruction. In reality, executed instructions vary in their resource requirements, causing the pipeline stages to idle when not in use.

Finally, the major goal of the pipeline design is to minimize the inter-instruction dependencies. These dependencies cause the pipeline to stall, where stage n waits for

stage $(n-1)$ to finish its job, because stage n needs the results of stage $(n-1)$ [1].

In order to address some of the limitations of scalar pipelines, superscalar processors are discussed.

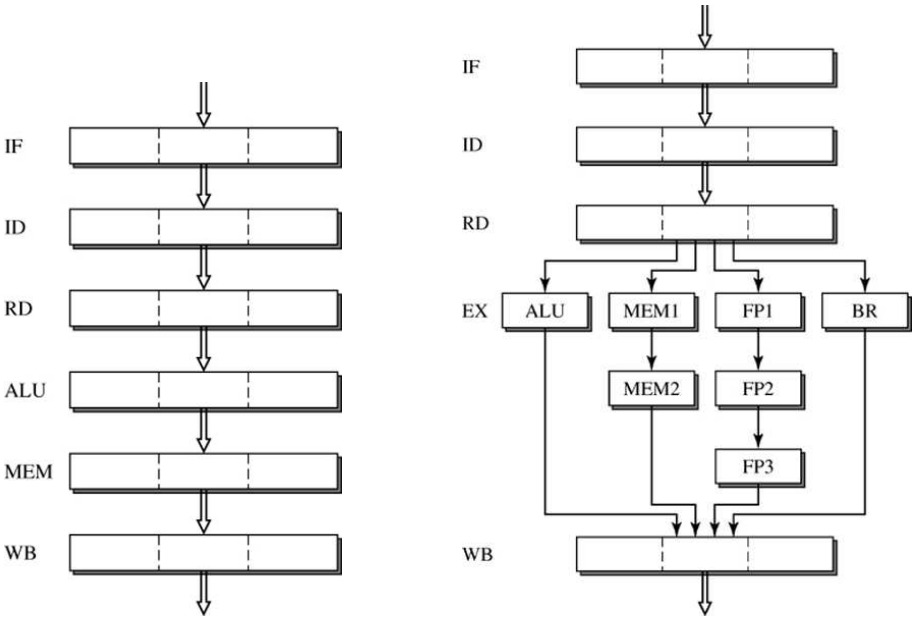
3.1.2 Superscalar Processors

Superscalar processors are an attempt by processor architects to increase the performance of the scalar pipeline. The main challenges of scalar pipelines can be summarized as follows [1].

- The performance of the scalar pipeline can be enhanced by reducing the number of instructions per program, reducing the number of Cycles per Instruction (CPI), or by increasing the clock frequency. However, the upper bound on the average Instruction per Cycle (IPC) is always one, since only one instruction can be fetched per cycle.
- The unified resource nature of a scalar pipeline proves inefficient when the instructions are not similar. For example, floating point, integer, and memory instructions are inherently difficult to unify, since they require fundamentally different resources.
- A pipeline that stalls due to data dependency is also a major deficiency. Since many of the stalls are avoidable if the pipeline is able to jump the stalled instruction and proceed to execute the non-dependent instructions.

The first approach to enhance the performance by increasing the IPC is to use simple replicates of a scalar pipeline, represented in Fig. 3.2(a). This ensures that the processor is capable of issuing more than one instruction per cycle, and in theory, raises the bound on the IPC to s, where s is the number of the parallel stages. However, simply replicating the pipeline does not solve the problem of the diversified requirements of the executed instructions. As a result, diversifying the pipelines is more adequate, since it allows for the customization of the instruction path to the instructions' resource requirements (Fig. 3.2(b)). To diversify the pipelines, the processor employs different functional units at the execution stage, allowing the processor to route the instructions in the pipeline depending on their requirements.

Early generation superscalars employed diversified pipelines, where the instructions are issued, decoded, and then forwarded to the execution stages without buffering, as



(a) Parallel pipeline (b) Diversified parallel pipeline

Figure 3.2: Generic superscalar pipeline [1].

long as the instructions are not dependent on other instructions. Processors such as the Alpha 21064, PowerPC 601, and the Pentium adopted such *direct issue* designs [32]. However, the major bottleneck for direct issue designs, is the blocking of the issue stage, when existing instructions are waiting for the unresolved dependencies.

Another major problem introduced by superscalar pipelines is the increased requirements in the cache bandwidth. Superscalar processors are capable of issuing more than one instruction per cycle, implying that for an s-wide pipeline, a corresponding s-wide Instruction Cache (I-Cache) is needed. In addition, a multi-port Data Cache (D-Cache) is needed, since more than one instruction may be requesting a read or write operation.

In addition to the memory bandwidth issue, branch misprediction penalties increase with wider pipelines. Branch penalties are the cost of recovery from mispredicted branches. The wider pipelines can increase the frequency of the branches per cycle, and thus, increase the chance of misprediction [32].

For later generations of superscalar processors, the solution for the problem of direct issue involved employing an Out-of-Order (O-o-O) execution. In the O-o-O scheme, the instructions, in theory, are only bound in their sequence of issuance, if they have data dependencies. In practice, Superscalar processors employ the hybrid technique denoted in Fig. 3.3, where the front end of the superscalar processor is an in-order fetch-decode-dispatch, followed by a reservation station, and then an out-of-order execution stage with

a reorder buffer, and finally, an in-order backend.

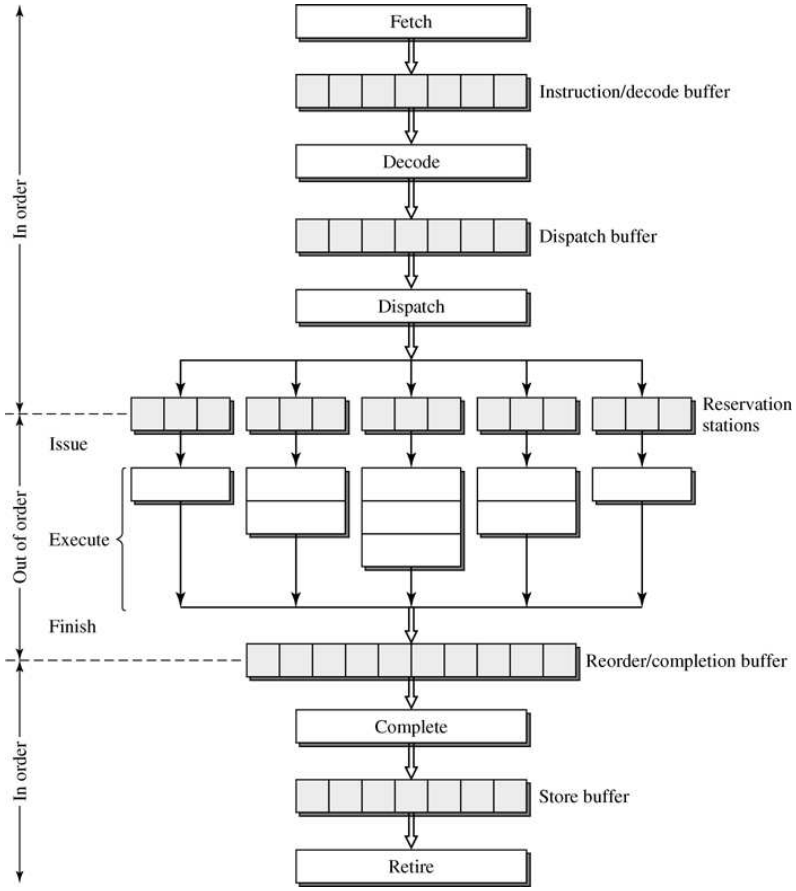


Figure 3.3: O-o-O superscalar processor [1].

3.1.2.1 Superscalar Pipeline Stages

The superscalar architecture can be explored by following the steps of an instruction execution in an s-wide processor [1]. This is composed of five steps

- Instruction Fetch (IF)
- Instruction Decode (ID)
- Instruction dispatch
- Instruction Execution (IE)
- Instruction completion and retirement

Step 1: Instruction Fetch (IF) The goal of the instruction fetch stage is to communicate with the I-Cache as quickly as possible to maximize the fetch bandwidth and attempt to fetch s instructions per machine cycle. The I-Cache is organized in blocks, containing consecutive instructions of the program. The program counter is used to search the I-Cache to determine if the requested instruction is in the I-Cache (cache hit) or not (cache miss) [33].

Typically, the fetch rate, measured in instructions per cycle, must match the maximum decode and execution rate, and in practice, is designed to be slightly higher to allow for cache misses or situations when fewer than the maximum instructions can be fetched. These situations arise when there is a misalignment of the fetched instructions block[2] with the cache boundaries. This causes the fetch stage to fetch only the instructions within the cache boundary in one cycle, then fetch the rest of the instruction block in the following cycle. This problem can be solved through a compile time I-Cache alignment or run-time hardware alignment [1].

Step 2: Instruction Decode (ID) The ID is responsible for identifying the instructions, determining the instruction types, and detecting the inter-instruction dependencies for the instructions that have been fetched but not yet dispatched.

In the Reduced Instruction Set Computer (RISC) architectures, the decode stage is simple, since it does not need to check for instruction boundaries. This substantially simplifies the decode operation and allows it to be merged with the register reading. In addition, the ID stage of a RISC processor is responsible for quickly identifying the control flow instructions in the fetch block in order to provide prompt feedback to the IF stage.

Contrary to the RISC, the Complex Instruction Set Computer (CISC) ID is quite complex. Besides the primary responsibility of the ID to identify the instructions, the CISC ID needs to identify the unfixed instruction boundaries. Moreover, the ID must translate the CISC instruction to its RISC-like operations, corresponding to the micro-operations (μOPs) in the Intel and RISC operations (ROP) in AMD.

Predecoding is sometimes useful to reduce the burden on the CISC ID. The CISC instructions are partially decoded on their way from the memory to the I-Cache. These instructions and a set of pre-decode bits are saved in the I-Cache, simplifying the decode operations. However, predecoding introduces two problems: an increased I-Cache miss penalty due to the increased overhead and extra storage requirements for the pre-decode bits.

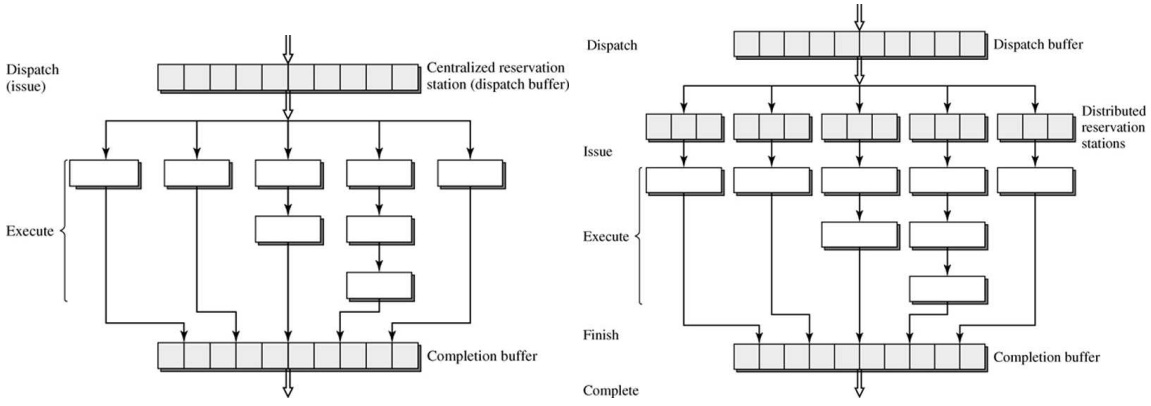
[2] A fetch instruction block is the set of instructions that needs to be fetched together by the IF stage.

Another task for the ID stage is register renaming. This is required because modern processors contain physical register files that are larger than the architectural register files [33]. The physical registers can be used to solve false data dependencies. Data dependencies will be revisited in more detail in Section 3.2.3.1.

Step 3: Instruction Dispatch Contrary to scalar processors, superscalar processors need an instruction dispatch stage to route instructions to the corresponding functional units, depending on the resource requirements of these instructions.

Instruction dispatching uses a reservation station which is a temporary buffer that holds the instructions that have been decoded, but not all its operands are ready. For example, those instructions that are dependant on an earlier instruction write to the register file (RF).

Reservation stations can be centralized, similar to the one denoted in Fig. 3.4(a). These stations provide the full capability of an O-o-O execution processor, since any instruction can be dispatched to any functional unit. However, the underlying complexity of the routing fabric limits the practicality of such stations in a wide superscalar processor. On the other hand, distributed stations in Fig. 3.4(b) are a partial solution to this problem. They allow the functional units to have their own buffers for queuing purposes, and keep a smaller dispatch buffer to perform the global routing of the instructions to the functional units' buffers. This reduces the routing complexity without eliminating the advantages of the O-o-O execution.



(a) Centralized (b) Distributed

Figure 3.4: Reservation station [1].

Step 4: Instruction Execution (IE) In this stage, the instructions utilize the resources of the functional units to perform the intended task of each instruction.

The trend is towards massively parallel IE stages, and the specialization of the functional units. This allows for more performance efficiency, since the dedicated functional units are used instead of a general purpose alternative.

The R10000 MIPS superscalar processor is a good example of such diversity. The integer execution stage of the R10000 contains two integer ALUs. The first ALU consists of a 64-bit adder and branch condition logic; the second ALU has a partial integer multiplier array and an integer divide logic. The floating-point execution stage contains a floating-point adder for addition, subtraction, comparison, and conversion, besides a floating-point multiplier for the multiplication and move operations. Finally, the R10000 contains a dedicated floating-point divide and a square root unit [34].

Step 5: Instruction Completion and Retirement After execution, the instructions are stored in the completion buffer, and are allowed to retire when they are done writing to the memory.

In this stage, the in-program order of the instructions is restored through the reorder buffer. The need for reordering is evident when the interrupt and exception handling is taken into account. Both interrupts and exceptions require that the processor retains its architectural machine state, and presents it as if the program is executed in order.

The reorder buffer is employed to handle this situation. The instructions enter the reorder buffer out-of-order and exit in-program order. The instructions that cause exceptions are flagged, and when they reach the exit of the reorder buffer, they are held until all the previous instructions are retired. Then, all ensuing instructions are discarded.

3.1.3 SMT Processors

To address the main limitation of Superscalar processors namely the absolute dependence on the instruction flow from a single application, SMT processors are devised. The applications running on a superscalar architecture do not make the best use of the available resources due to the latencies introduced by the memory system, branch mispredictions, and dependencies on high latency instructions.

Because of these latencies, the application remains idle, waiting for the data to arrive or the dependence to be resolved. This idleness results in a low usage of pipeline resources in superscalar processors that has led to the SMT architecture [35].

Such architectures execute several applications on the same pipeline and at the same time. This results in a better usage of the available resources, and serves also as a latency

hiding technique. When an application is on hold, waiting for the data to arrive from the memory, for example, another application continues to be executed. Therefore, the SMT architecture has a higher throughput with a system that combines a superscalar capability and multithreading.

However, when several applications are simultaneously run on the processor, the behavior is entirely different from that of a single application workload. When several applications share the pipeline resources, two things occur. The first is that the utilization of these resources is expected to increase, increasing the power dissipation. The second thing is that the behavior of these resources, in terms of the busy and idle periods, becomes less predictable.

3.1.3.1 SMT scheduling and resource sharing

SMT scheduling: The goal of multi-threaded processors is to maximize the sharing of the processor resources between requesting threads. Earlier implementation of fine-grained multithreading included round robin techniques which allotted specific time slots to each thread [1]. This implementation has maximum utilization of the resources if the number of threads are equal to the available time slots, otherwise the processor will idle during the empty time slots (Fig. 3.5). In addition, single thread performance is heavily impacted due to the unnecessary latency associated with the round robin implementation. To mitigate the drawbacks of fine-grained multithreading coarse-grained multithreading is introduced where the processor switches between threads when the pipeline stalls on long latency events. Hence, the processor pipeline is utilized by an alternate thread while the initial thread awaits the resolution of the stall (Fig. 3.6). The challenge for the course-grained processor is to make sure that each thread enjoys a fair chance in getting execution resources, a tough task when the stall profile of the executing threads is different.

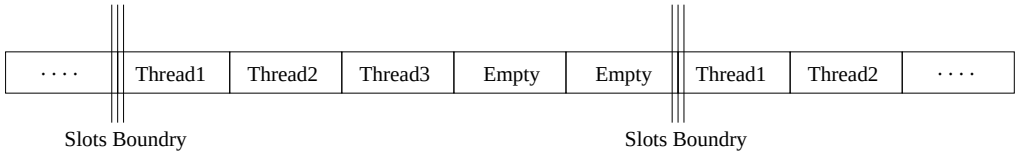


Figure 3.5: Fine grained Multithreading.

Simultaneous multithreading (SMT) combines a fine-grained implementation with the ability to dynamically switch between instructions from multiple threads as shown in Fig. 3.7. The SMT is very well suited to run on out-of-order processors since the instructions are already decoupled in their execution from the program order.

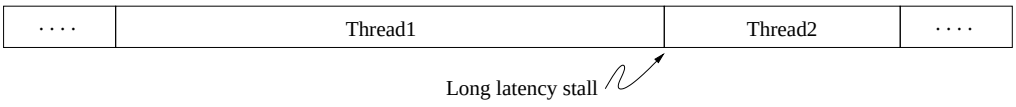


Figure 3.6: Coarse grained Multithreading.

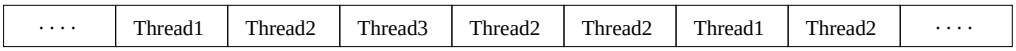


Figure 3.7: Simultaneous Multithreading.

Resource sharing: In order to implement an SMT processor the resources available to the threads has to be modified to allow multiple threads to proceed in the processor pipelines [1]. The choice between the various implementations in Fig. 3.8 is controlled based on the efficiency of sharing each processor block between the various threads.

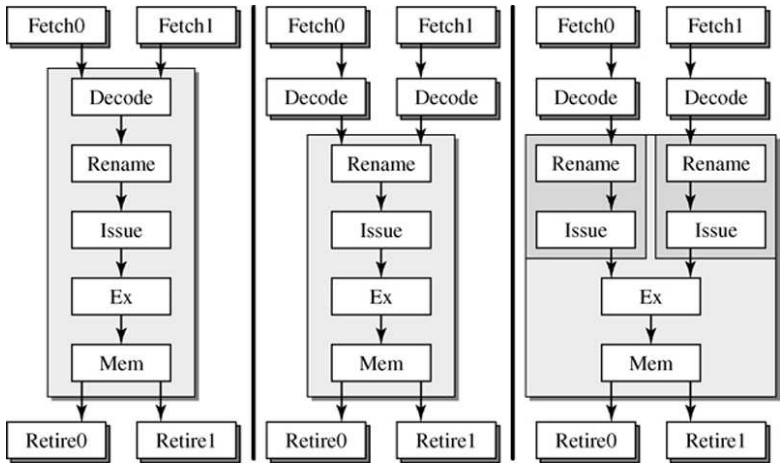


Figure 3.8: SMT resource sharing [1].

Designing the SMT pipeline requires the designers to choose between a single shared pipeline stage versus a partitioned stage. A single shared pipeline stage allows for maximum single thread performance since the pipeline stage is completely available to each thread. Partitioning the pipeline stage limits the single thread performance when there are no competing threads. However it significantly simplifies the design of the pipeline stage. The main limitation of the sharing of the fetch stage lies in the instruction cache [1]. Accordingly, modern SMT implantation similar to SUN's Niagara processor uses a shared fetch stage while multiplexing the requests to the instruction cache [36]. Using the same thread multiplexing logic, a decode stage can be shared overriding the complexity of extending a single interleaved decode stage between threads and the associated difficulty of resolving the instruction dependencies of unrelated threads [36]. Since the register re-

naming involves the translation of the architectural registers to the shared pool of physical registers, simultaneous accesses can be easily accommodated. The execution stage with its diversified resources can very well be simultaneously accessed from multiple threads with minimum overheads.

Sharing the memory is fairly straightforward using multi-port caches. However, the complexity of sharing the load-store queue might warrant the partitioning of the queues based on the number of threads. Finally, the retire stages is better partitioned between threads [1]. In summary, the decision of sharing each of these resources is a tradeoff between single thread performance and the hardware complexity. An interleaved stage allows for better single thread performance at the expense of a complex hardware to resolve the thread specific information and vice versa.

In order to complete the background of processor power consumption, the following section presents an overview of the leakage power management techniques available for state-of-the-art processors.

3.2 Leakage Power Management Techniques

As mentioned in Section 1.1, to successfully design a low power microprocessor, the designer needs to consider the combined effects of the system, the algorithm, and the circuits. Accordingly, to formulate the proposed standby prediction methodology, it is also important to explore the most popular circuit and system-level leakage control techniques. In the following Sections, various dynamic low leakage circuit techniques available to VLSI designers are explored. In particular, power gating is further discussed since it is later used as a test vehicle for the dynamic sleep signal generators. Then, in Section 3.2.3, the combined application of these circuit techniques and system-wide measures to exploit the power saving potential is explained. Finally, building on the circuit and system low-power foundation, Section 3.2.4 introduces some of the compiler and architecture-based program profiling techniques that are relevant to the task of standby prediction.

3.2.1 Leakage Power Management Overview

Referring to Fig. 3.9, the task of leakage control is divided between two major components; the leakage control mechanism, and the sleep signal generator.

Many leakage control mechanisms have been studied [3, 28, 30, 37–40]. Power gating circuit techniques are based on shutting down parts of the circuit which employ high

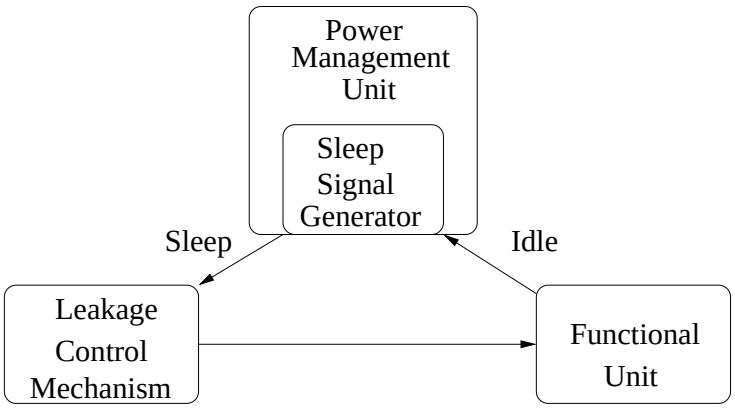


Figure 3.9: Dynamic leakage control.

threshold transistors between the functional units and the supply rail (Fig. 3.10(a)) [28]. The high threshold transistors exhibit 10 times less leakage than low threshold transistors [30]. Accordingly, turning off these transistors during standby reduces leakage of the whole functional unit [37]. Alternatively, input vector activation is based on changing the inputs of the circuit to reduce its leakage (Fig. 3.10(b)). The leakage reduction results from the dependence of the transistor leakage on the number of OFF transistor between the supply and ground. Accordingly, maximizing the number of OFF transistors in standby reduces the overall leakage [38]. Finally, since transistor leakage is exponentially related to the transistor threshold voltage, adaptive body biasing techniques reduce the functional units' leakage in standby by raising the transistor threshold voltage (Fig. 3.10(c)). Raising the threshold voltage is achieved by applying a reverse bias to the transistor body [3, 39, 40].

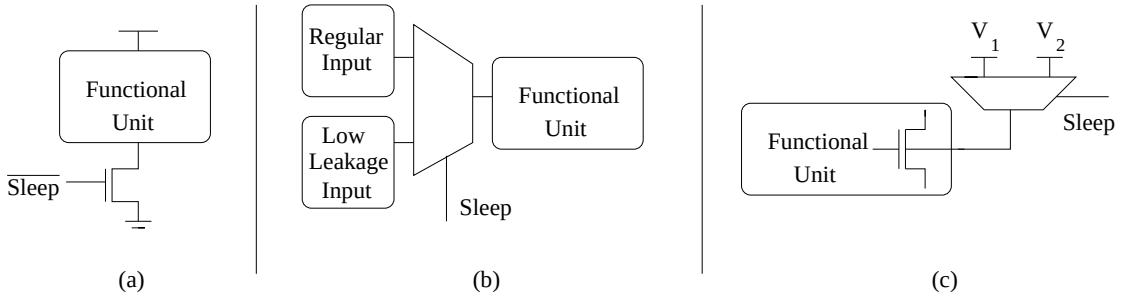


Figure 3.10: Dynamic leakage control mechanisms a) Power gating. b) Input vector activation. c) Adaptive body biasing.

In order to fully describe the operation of the leakage control mechanisms discussed so far, the timing diagram shown in Fig. 3.11 represents the major regions of operation for these mechanisms. The first region is where the functional unit is being used. In this region, the leakage control mechanism is disabled. The second region begins when the

functional unit idles. As soon as the sleep signal generator asserts the sleep signal, the third region begins. During this region the total power consumed initiating the leakage control mechanism is less than the total savings achieved through the reduction of leakage power. Finally, the fourth region begins when the power consumed by the circuit initiating the leakage control mechanism equals the power saved from leakage reduction.

From Fig. 3.11, the breakeven point is defined as the point in time when the energy consumed to generate the sleep signal is equal to the energy savings due to the leakage reduction. The breakeven point is illustrated by the onset of region 4. This breakeven point determines if enough power savings is achieved to warrant the use of any leakage management circuit (i.e., to assert the sleep signal).

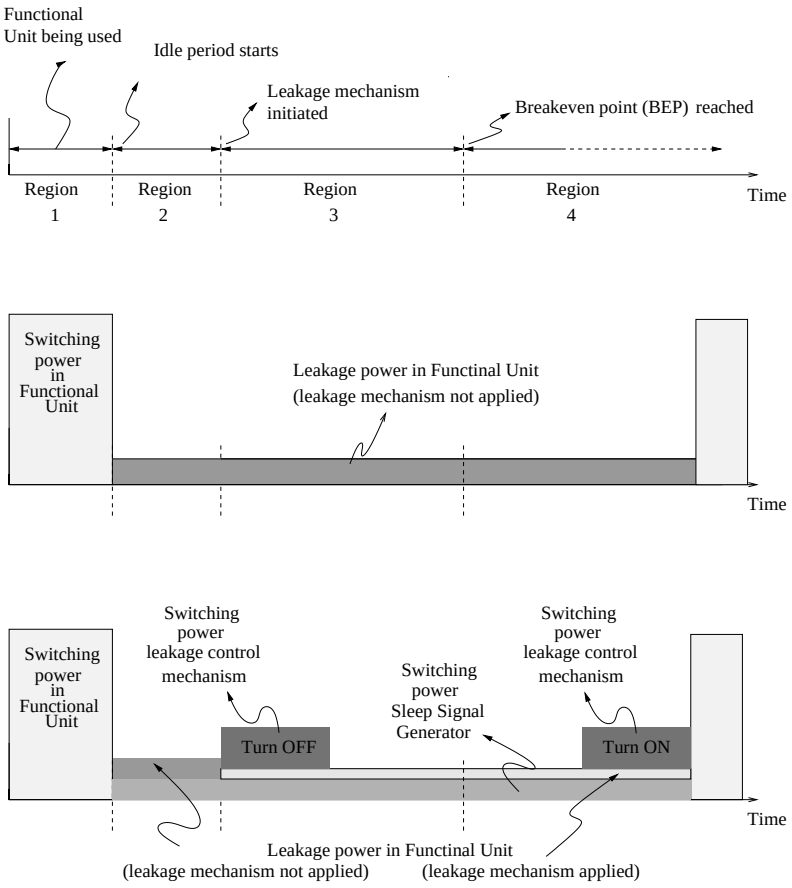


Figure 3.11: Breakeven point.

3.2.2 Circuit-Level Leakage Control Techniques

Leakage control circuits can be classified into static and dynamic techniques, depending on whether the control is statically applied at the design phase, or dynamically managed

during the circuit operation. In dynamic leakage control techniques, the circuit blocks enter a low leakage mode that can be short or long, according to the usage profile of the circuit. Sections 3.2.2.1 to 3.2.2.4 are a review of some of the dynamic leakage control techniques in the literature.

3.2.2.1 Input Vector Activation

In their earlier work, Ye et al. have introduced the concept of input vector activation by noting the dependence between the leakage current in a transistor stack and the input to these transistors [38]. Fig. 3.12 shows a two-input NAND gate to illustrate the relation between the inputs and the leakage power. With both NMOS devices OFF, the leakage current flowing through them is approximately an order of magnitude smaller than the leakage of a single transistor. This reduction in leakage is attributed to the negative gate-to-source biasing and the increase in the M1 transistor threshold due to the body effect [38].

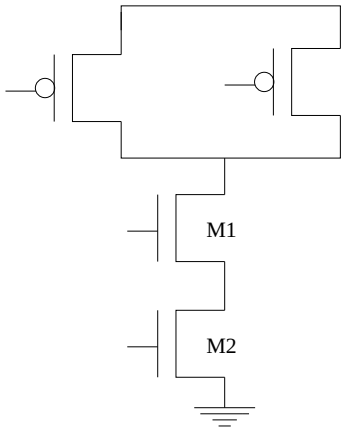


Figure 3.12: Two input NAND gate.

Capitalizing on this dependence of the leakage current on the number of OFF transistors between the V_{dd} and the ground, many researchers have focused on changing this path in the standby state to put the circuit into a low leakage mode. However, the problem of finding the input vector that minimizes leakage has been shown to be NP-complete [41]. NP-Complete problems are non deterministic polynomial hard problems. These problems are a subset of combinatorial problems that are not solvable in polynomial time. Accordingly, alternative heuristics have been implemented to identify the most effective input vectors [42], where the input vector activation is combined with transistor stacking to further enhance the leakage power performance.

One of the major challenges of input vector activation techniques is that the effect of the input vector diminishes with the logic depth, which, for example, translates into low savings for large multipliers [43]. This hurdle can be managed by using bypass multiplexers to control the internal nodes of the data path.

3.2.2.2 Dynamic Voltage Scaling (DVS)

In Dynamic Voltage Scaling (DVS) techniques, the DC supply of the circuit is adjusted on the fly to a lower value to reduce the leakage current [30]. The lower DC supply benefits both the switching and leakage power. The leakage is reduced because the subthreshold leakage, due to drain-induced barrier lowering (DIBL), decreases as the supply voltage scales down [44]. It has been shown that reductions in the subthreshold and gate leakage are proportional to V_{dd}^3 and V_{dd}^4, respectively [45]. However, it is noteworthy that the performance of the circuits becomes more sensitive to the variation of V_{dd} and V_{th} at lower supply voltages. In addition to the change of the supply voltage, DVS techniques are accompanied by a change of the circuit speed. Earlier research reports that up to a three-fold reduction in the frequency can be observed, if the supply is scaled from 1V to 0.5V [46].

To implement the DVS, an efficient DC-DC converter is required, adding to the complexity and area overhead. Gutnik et al. have employed a PLL-based DC-DC conversion [47]. This technique has the advantage of controlling the supply voltage and the system frequency simultaneously.

3.2.2.3 Adaptive Body Biasing

Adaptive body biasing techniques are based on changing the threshold voltage of the devices according to the circuit state. These techniques apply a forward bias to the body, when the circuit is active, to enhance performance, and a reverse bias, when the circuit is idle, to reduce leakage. V_{bias} shown in Fig. 3.13 is used to control such a technique. Reducing the V_{bias} causes an increase in the circuit threshold, reducing its leakage. Conversely, increasing V_{bias} reduces the threshold, which increases the circuit speed [39]. This operation is governed by the following equation:

$$V_{th} = V_{tho} + \gamma(\sqrt{|V_{SB} - 2\phi_f|} - \sqrt{2|\phi_f|}), \tag{3.1}$$

where γ is the body effect coefficient, ϕ_f is the fermi potential, and V_{SB} is the source to bulk potential represented by V_{bias}.

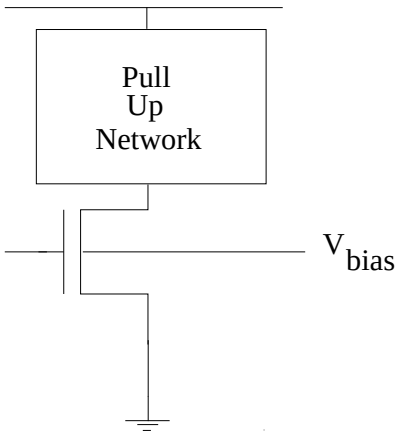


Figure 3.13: Adaptive body biasing technique.

However, this approach is limited in its use to low temperature operation, since higher temperatures increase the leakage current between the n-well and the p-well in the triple well structure needed to isolate the various transistors [40].

Another limitation to the adaptive body biasing techniques is their diminishing performance as CMOS technologies scale down. Research shows that the maximum achievable leakage power reduction is reduced four-fold per technology generation [48]. This is attributed to the increased band-to-band tunneling for smaller transistors.

3.2.2.4 Power Gating

Referring to Fig. 3.14, the task of leakage control for any circuit that employs power gating[3] is divided between two major components; the sleep transistor network, and the sleep signal generator. The basic idea behind power gating techniques is to shut down parts of the circuit when they are not utilized, using a high threshold device between the functional unit and the supply rail [3, 28, 30, 37]. These devices are setup in a sleep transistor network as shown in Fig. 3.14. The high threshold transistors exhibit 10 times less leakage than low threshold transistors [30]. Turning off these transistors during standby reduces leakage of the whole functional unit [37].

During the active mode, when sleep signal is low and the sleep transistor is ON, the sleep transistor can be modeled as a resistor between the functional unit and the supply rail. This generates a small voltage drop V_X across the resistor (Fig. 3.15). The voltage drop V_X has two effects [49]. First, it reduces the driving capabilities of the gates inside the functional units from V_{dd} to $(V_{dd} - V_X)$ and second, it causes the threshold

[3]Power gating is also known as Multi-threshold CMOS (MTCMOS).

voltage of the low threshold pull-down devices to increase due to the body effect [38,50]. Both effects degrade the speed of the circuit. Therefore, the resistor should be made small, and consequently, the size of the sleep transistor be made large. This comes at the expense of extra area and power. On the other hand, if the resistor is sized too large (i.e., the sleep transistor is sized small), the circuit speed will degrade and the loading on the sleep signal generator circuitry will decrease. Therefore, a tradeoff exists between achieving sufficient performance and low power values. This tradeoff becomes even more evident in the DSM regime. In DSM technologies, the supply voltage is scaled down aggressively, causing the resistance of the sleep transistor to increase dramatically, requiring even larger size sleep devices. This will cause leakage and dynamic power to significantly increase in the standby and active modes, respectively. Therefore, an important design criterion is sizing the sleep transistor to attain sufficient performance.

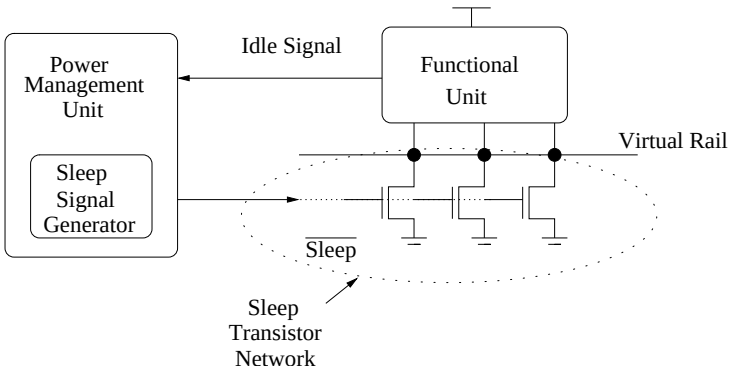


Figure 3.14: Power gating within the framework of a power management unit.

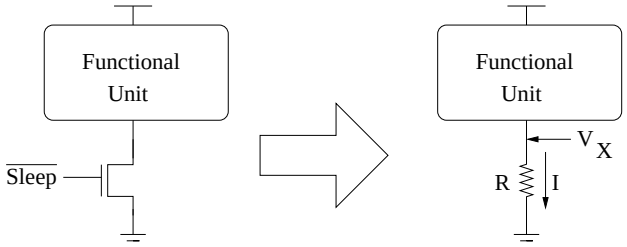


Figure 3.15: Sleep transistor in power gating circuits.

For power gated circuits the breakeven point is defined as the point in time when the energy consumed to generate the sleep signal and turn off the sleep transistor network is equal to the energy savings due to the leakage reduction. The breakeven point determines if enough power savings is achieved to warrant the use of power gating. The value of the breakeven point is highly dependent on the amount of energy consumed going in and out of the low leakage mode and the time needed for the DC operating point to converge

to its final value [3]. Hu et al. in [4] use a highly parameterizable analytical model to show that the breakeven point for power gating techniques can reach as low as 10 cycles. However, in real implementations such as the integer ALU in [3] the breakeven point is achieved around 100 cycles. This difference is mainly attributed to the block size (course vs. fine granularity) and the value of the decoupling capacitance as it adds to the total capacitance that needs to be discharged. Furthermore, tests in Section 4.3.2.2 will discuss the value of the breakeven point in more details through circuit simulations of Fanout of 4 (FO4) inverter networks and a standard cell implementation of an open source floating point unit [51]. These tests indicate that a 10 to 500 cycles is a reasonable range for the breakeven points, the exact value of which depends on the functional unit being power gated.

Another issue for MTCMOS implementation is the data loss, when the circuit is switched to the sleep mode. Data retention high-threshold buffers are available to retain the data through the sleep state. However, these buffers increase the complexity and can result in leakage sneak paths [52].

Beside the need for data retention buffers, MTCMOS circuits are constrained by the overhead needed to charge and discharge the virtual rail seen in Fig. 3.14. This constraint further limits the use of MTCMOS circuits to reduce leakage during long standby periods. Variants of the MTCMOS scheme introduce soft sleep states, where the data is retained by clamping the virtual rail to V_{th}, allowing for a faster recovery with lower power overhead [53].

3.2.2.5 Discussion

The principal challenge in all the leakage control techniques is the identification of the circuit's usage patterns during operation. For example, the performance of dynamic leakage control techniques depends heavily on the length of time the circuit can be put in a standby mode. This time is a major factor, since the overall savings in leakage need to be calculated accounting for the overhead of forcing the circuit into the standby mode, waking up the circuit when an operation is pending, and the impact of the delay required to wake up the circuit on the overall performance. Consequently, these techniques work best when there is an accurate estimation of the standby times, and when there is flexibility in changing the task schedule in order to maximize the total sleep time.

To estimate the circuit's usage patterns, system-level power management is needed. System-level power controllers are able to observe the various system components and their interactions.

3.2.3 System-Level Leakage Control Techniques

The low power circuits presented in the previous section are the basic building blocks for system-wide low power design. System-level power management has the potential of overseeing multiple circuit blocks and their interactions. Thus, more power savings can be achieved by optimizing the overall performance of the microprocessor at the system level.

In addition to general purpose microprocessors, there are also their embedded counterparts and programmable DSPs that are used for specific applications such as automation, video and audio processing, coding/decoding, and filtering. Fortunately, these applications have characteristic performance needs. For example, video processing tends to be differential in nature, calculating only the difference between the successive frames. This fact allows for more system-level optimization than what is available for general purpose processors.

In this section, the various system level techniques present in the literature are discussed. Section 3.2.3.1 presents some of the challenges and techniques available to superscalar and SMT processors, whereas Section 3.2.3.2 introduces software-level power consumption control.

3.2.3.1 Power Management for Superscalar and SMT Processors

As mentioned in Section 3.1.2, superscalar pipelining and SMT are two of the most prevalent approaches for enhancing processor performance. However, these architectures introduce issues that need to be considered for the processor operation to be successful. To describe some of the challenges and low power potential for superscalar and SMT processors, the generic superscalar processor presented earlier in this Chapter is used. This processor is represented in Fig. 3.16 for clarity. In the following sections, some of the pipeline aspects are presented with the objective of exploring some of the potential situations for reducing the overall power consumption.

Pipeline Stall By using the pipeline structure in Fig. 3.16, instructions are fetched in each clock cycle. In an ideal situation, the instructions utilize 100% of the processor all of the time. However, in reality, many situations arise that force one or more pipeline stages to be idle. These situations, which are presented in the following sections, can be considered for potential power savings.

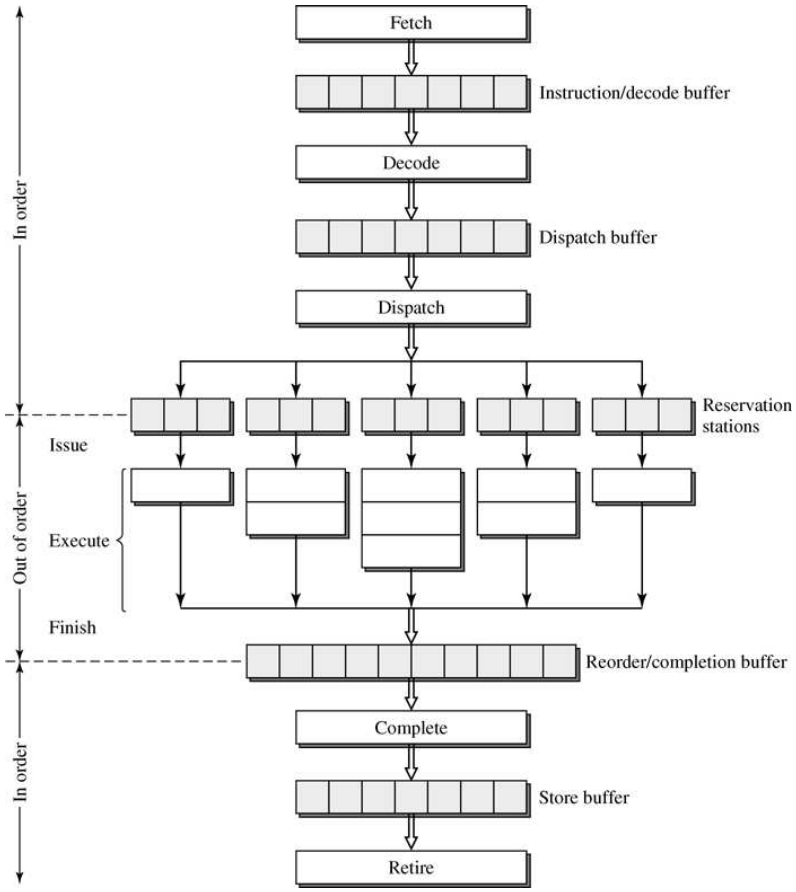


Figure 3.16: O-o-O superscalar processor [1].

Read After Write (RAW)

$$I1 : R_3 \leftarrow R_1$$
$$I2 : R_5 \leftarrow R_3$$

Consider two consecutive instructions $I1$ and $I2$. $I1$ writes to a memory or register file location R_3, and $I2$ needs to read from the same location. $I1$ will only write to R_3 at the Retire stage. However, $I2$ is following $I1$, and $I2$ needs to read R_3 before being executed. This situation shows the Read after Write data hazard that forces the pipeline to stall, waiting for $I1$ to write back before $I2$ can proceed for execution.

Write After Write (WAW)

$$\begin{aligned} I1 : R_3 &\leftarrow R_1 \\ I2 : R_5 &\leftarrow R_3 \; op \; R_x \\ I3 : R_3 &\leftarrow R_7 \end{aligned}$$

Consider three instructions: $I1$, $I2$, and $I3$. $I1$ and $I3$ write to the same memory or the RF location R_3. $I2$ reads R_3 after $I1$, but before $I3$. This situation presents a Write after Write data hazard. In this case, if $I3$ writes to R_3 before $I2$ is at dispatch, the data for $I2$ is corrupted. This case can arise, if the core has more than one execution unit and a scheduler that may allow $I3$ to execute faster than $I2$. This can also happen if $I3$ does not need the ALU attention, and is only a write operation to the RF; in this case, $I3$ is stalled while waiting for $I2$ to finish. This situation is more prevalent in O-o-O execution processors, where the instructions in flight are not in the same program order.

Write After Read (WAR)

$$\begin{aligned} I1 : R_3 &\leftarrow R_1 \; op \; R_x \\ I2 : R_1 &\leftarrow R_y \end{aligned}$$

Consider two instructions $I1$ and $I2$. In the first instruction, R_1 is being written to R_3. In the second instruction, a value is being written to R_1. Since $I2$ modifies the R_1 value, this instruction cannot be allowed to write back before $I1$ does. Again, this situation can arise with processors with multiple execution units or O-o-O processors.

Cache Miss A key potential power saving situation is the cache misses. In this case, instructions try to read nonexistent data from the cache, resulting in a multiple cycle wait for the higher-level cache or the external memory to return the data. A methodology, similar to the work by Zhu et al. [54], can be used to predict memory accesses and manage the execution stages, depending on the memory access profile.

Resource Contention Another aspect that can cause the pipeline to stall is the resource contention. The reservation station can overflow, causing a dispatch stall. The cross bar switch, responsible for routing the instructions to the various functional units, can also become overloaded, causing a stall. Finally, the rename register is also a shared resource that can cause a stall, when it is unavailable [1].

In all these situations, power gating can be used to set the pipeline stages into low power mode, when the pipeline is stalled. In addition, the DVS can be used to slow down the pipeline stages in order to equalize the execution time, or to prevent RAW, WAW, and WAR hazards. Also, the DVS can be used in the cache miss situation to slow down the circuit operation, until the higher-level cache returns the data [55, 56].

Branching Typically, branch instructions and subroutine calls are detected in the decode stage. However, the Taken/Not Taken (TK/NT) branch is not resolved, until the instruction reaches the execution stage. In addition, the branch target[4] is calculated in the execution stage. Accordingly, any instructions that are fetched in the shadow of the branch must be flushed, in case the branch is mispredicted. This is illustrated in Fig. 3.17, where I1, I2 and I3 are discarded, if the Branch (Br) is taken.

To mitigate the impact of the branch TK/NT problem, accurate branch prediction techniques have been developed to accurately guess the direction of the branch. Earlier work has focused on the low power implementation of the branch prediction [57]. Researchers have investigated the supply gating of inactive prediction arrays[5]. Later, the accuracy of the branch predictor was used as a gauge of power consumption [58]. More accurate branch predictors are less prone to prediction error, saving the power consumption of the processor core.

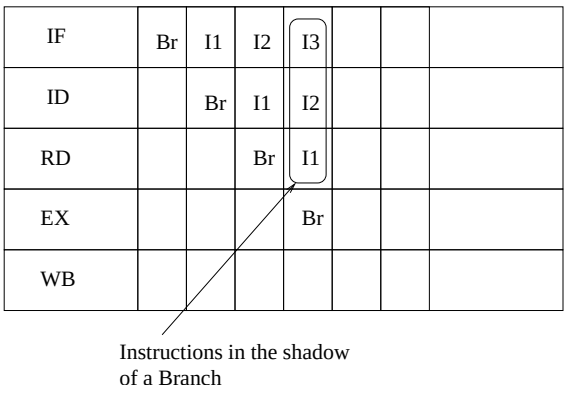


Figure 3.17: Branching in pipelined processors.

[4]The branch target is the memory address that the branch is jumping to.

[5]Branch prediction is preformed through a set of two-bit state machines arranged in an array capable of predicting more than one branch.

3.2.3.2 Low-Power Software Design

Most of the effort in low power processor design is concentrated on the hardware, since it is the actual vehicle performing the various operations [59]. However, the impact of software design on the total power consumption is substantial. This is obvious since the program code contains information about the sequence of events that take place in the underlying hardware. In the following section, some of the sources of software power consumption and popular low power software design techniques are investigated.

3.2.3.3 Sources of Software Power Consumption

The power consumption of general purpose processors is the collective contribution of many components: Memory, ALU, multipliers, buses, clock, and control logic. The following components are the principal contributors to the power consumption of the processor [59].

Memory Subsystem: Typically, memory accesses are power hungry, since they involve the switching of large capacitance word/address line and activating row/column decode logic. For example, the L2 cache of the dual core UltraSPARC microprocessor consumes up to 14% of the chip's power consumption [60]. Accordingly, software efforts that utilize data locality and minimize memory accesses can be extremely beneficial in minimizing power consumption.

LOAD/STORE are the main instructions accessing the memory, the order of which can be used to power gate some of the execution units, predict pipeline stalls, or slow down the execution, using techniques similar to the DVS [59].

Buses: Buses are also a source of power consumption, since many components are connected to the same bus, increasing the overall bus capacitance. Reducing the bus activity using the information available at compilation time can be very helpful in minimizing power consumption. This can be achieved through compiler directives or instruction rearrangement with the objective of minimizing the bus activity [61].

Functional Units: Functional units are another major source of power consumption. These units consume approximately 38% of the UltraSPARC core's power consumption [60]. Accordingly, efforts to reduce their power consumption greatly enhance the processor performance. Although an exact prediction of the inputs to the functional units

is not possible, the sequence of instructions is available at the compilation time. Knowledge of the sequence of the instructions can be used to minimize the power consumption by changing the execution blocks used. For example, if a sequence of instructions is not dependent on a multiply instruction, this instruction can be transformed to a regular software multiplication. This reduces the overall power consumption of the multiplier unit, or keeps the unit in the sleep mode longer [59].

3.2.3.4 Software Power Reduction

Power Estimation: The first task for attaining low power software is the modeling of the power consumed by the application code. This is achieved by the following steps [59]

1. *Base Cost Calculation*: In this step, the power consumed by each instruction, assuming that it is the only instruction in the processor, is calculated. This power is independent of the sequence of events and bypasses the cache misses, pipeline stalls, branch mispredictions, and bus switching due to the opcode differences. This step is achieved by inserting the instruction in a loop that is long enough to eliminate the branch overhead at the end of the loop, but not too long to cause cache misses.

2. *Circuit State Cost Calculation*: Here, pairs of consecutive instructions are examined to determine the following.

   - The Opcode change and the associated change on the instruction bus.
   - The switching of control lines to account for the alternation between the various functional units.
   - The switching associated with the rerouting of the data from the ALU through the bypass paths to the register file.

3. *Total Program Power Calculation*: In this step, the following equation is used to quantify the total power consumed by each program [59].

$$E_p = \sum_i (B_i \times N_i) + \sum_{i,j} (O_{i,j} \times N_{i,j}) + \sum_k E_k, \tag{3.2}$$

   where E_p is the overall power consumed by the code. $\Sigma_i(B_i \times N_i)$ is the base cost calculation, where B_i is the base cost of instruction type i, and N_i is the number of instructions of type i in the code. $\Sigma_{i,j}(O_{i,j} \times N_{i,j})$ is the circuit state cost, where $O_{i,j}$ is the cost of the instruction of type i, followed by an instruction of type j, and $N_{i,j}$ is the number of occurrences of the $\{$i,j$\}$ sequence. Finally, $\Sigma_k E_k$ is the cost of the pipeline stalls, cache misses, branch mispredictions, and interrupts.

Power Optimization The second aspect of developing low power software is to explore the power saving potential of the application code and determine the appropriate solutions. This step is especially beneficial for embedded processors since they are usually tied to a specific application, allowing the use of application specific measures to combat the power consumption.

Minimizing the memory access cost is one of the promising solutions to the power consumption problem. Minimizing the memory access cost can be achieved by the following [59].

- Minimizing the number of accesses to the memory by using data locality. This is achieved by code reordering to localize the computations to the data available in the register file before requesting new data.
- Ordering the memory references by proximity, by accessing the registers first, the caches, and then the main memory.
- Using multiple word LOADs[6] to further minimize the number of required memory accesses.

Another approach for software power reduction is related to instruction selection and ordering. In this case, the code is modified to minimize the power consumed in the processor core by the following:

- **Instruction Packing**: In this technique, more than one instruction is packed in an instruction bundle. This packing minimizes the overhead attached to individual instructions and allows for more control over the power consumption [62].
- **Instruction Ordering**: By using the information available from the power estimation step, certain code sequences can be identified as a potentially power hungry combination. This is due to excess bus switching or the changing of the functional units. Code reordering can be employed to eliminate these sequences at the expense of increased compilation time [63].
- **Data Flow Management**: Alternately, if the objective is to minimize the leakage of a DSP core, data flow management techniques can be used to prevent the over switching of some circuits which results in shorter standby times. In essence, data flow management techniques throttle data flow to the execution units such that an objective is achieved. For example, this objective can be maximizing the standby time [64].

[6] A regular LOAD instruction loads only one word from the memory; multiple word LOAD reads more than one word per cycle.

3.2.4 Program Profiling

After a discussion of the circuit and system-level techniques that attempt to manage the leakage power, it is crucial to review the previous research related to program profiling. The objective of program profiling is to identify the trends in the executed code. Trends such as the number of loops in the executed code and when they are executed, or the impact of the code execution on the IPC. Understanding program profiling is an essential factor in the work towards standby prediction.

In this section, two approaches that attempt to identify program profiles are discussed. In Section 3.2.4.1, compiler-based approaches are presented, followed by the higher-level approach of software and hardware phase extraction in Section 3.2.5.

3.2.4.1 Compiler-Oriented Approach

The idea of compiler support for managing the different aspects of hardware is not new. However, the main drawback of compiler-based modifications is the need to recompile much of the existing code to benefit from such techniques. In addition, as the hardware implementation changes, some of the older techniques or microarchitectural features can become ineffective or even undesirable. If some of these older features are promoted to the Instruction Set Architecture (ISA) level, then they become part of the ISA, resulting in an installed software base or legacy code containing these features. Thus, all future implementations must support the entire ISA to ensure the portability of the existing code [1].

Rele et al. have used compiler directives to manage leakage power consumption [65]. The technique is based on modifying the instruction set by adding an OFF/ON instruction pair to each of the processor's functional units. For example, for the floating-point multiplier, the compiler is assigned the task of analyzing the program offline, and identifying the program regions that do not use the floating-point multiplier. The compiler then inserts a pair of OFF/ON instructions between the boundaries of these regions. In order to prevent OFF/ON pairs that are close in time, the OFF instruction only sets the functional unit status to pending-OFF. If an ON instruction for the same functional unit is encountered before the OFF instruction retires, the functional unit is kept on, otherwise it is powered down.

To extend this work, Kim et al. have used the compiler to tune the instruction per cycle (IPC) in order to manage the leakage periods more efficiently [66]. This technique exploits the inherent variations of the demand for functional units in an application to change the amount of instruction parallelism to reduce the functional units' usage. The

idea behind this technique is to tag loops according to the amount of parallelism they require, and using these tags to power down the unused functional units.

3.2.5 Phase Extraction

In addition to compiler-based approaches, hardware and software-based techniques that extract the usage profile at a coarse granularity have been studied. Contrary to the compiler-based techniques discussed in Section 3.2.4.1, these techniques do not focus on managing leakage power. Instead, they attempt to extract the trends of major code sections. These trends are called program phases.

Hardware Assisted Phase Extraction: Hardware assisted techniques are implemented by periodically probing the processor utilization. Such techniques extract the usage profile and submit this information to the operating system in order to perform tasks such as dynamic voltage or frequency scaling. The probing can be performed by the processor itself or by a profiling co-processor [67, 68].

Narayanasamy et al. have chosen hardware-based tables to set aside important code characteristics such as LOADs that frequently miss and are frequent in the code. This information is helpful for techniques such as multipath execution and cache prefetching [67]. Similar techniques use long execution segments (10 million instructions) to identify similar program phases aimed at DVS and phased-based task scheduling [68].

Due to the slow nature of such techniques, they are only useful when the overhead of the decision and the power savings are large. This is the case for some of the dynamic voltage and frequency scaling techniques, where the decision to vary the supply voltage or frequency requires a series of steps that are time consuming.

Software Assisted Phase Extraction: Similar to the hardware techniques, the program code is analyzed to extract long-term trends. Sherwood et al. have reported an especially important implementation of these extraction techniques [63]. By defining the Basic Block Vector (BBV) as a section of code that is executed from start to finish with one entry and one exit. The researchers have managed to use the BBV as a metric to compare the behavior of the program across various BBVs. The concept of the BBV has been valuable to reduce the simulation time of cycle accurate performance simulators [69]. This is performed by identifying the program phases and tagging them with their IPC, cache misses, and branch mispredictions. Searching through these tags, they manage

to identify patterns in the executed code. These patterns are later used to prevent the re-simulation of code segments that are similar, reducing the total simulation time.

3.2.6 Discussion

To implement all of the system-level power management measures, mentioned in this chapter, the designer needs two assets

Cycle Accurate Simulator: This set of performance simulators is used to simulate the processor execution on a cycle-by-cycle basis. Furthermore, these simulators have the ability to estimate the flow of instructions in the processor. Also, these simulators are capable of generating usage reports that contain information about pipeline stalls. The SimpleScalar and SMTSIM are two of the most accurate performance simulators available [70].

Benchmarks: To perform cycle accurate simulations, the designer needs accurate benchmarks that can predict the behavior of an average program. The SPEC2000 benchmarks [71] are suitable for general purpose processors, whereas MiBench is more appropriate for embedded processors [72].

3.3 Summary

In summary, in order to maximize the power saving potential of any leakage management technique the system designers need to consider the combined effect of the application, the architecture and the circuits. In order to build a leakage management system that is capable of tracking the application behavior the following Chapter presents the concept of predictive sleep signal generation.

Chapter 4

Predictive Sleep Signal Generation

General purpose and embedded processors represent the cornerstone of many consumer electronics applications [1]. Processors in portable equipment are faced with a very stringent power envelope due to the battery dependence. On the other hand, server class processors are also power-bound by the available heat sinking capabilities and the increasing cost of operation due to the power consumption in server clusters. These facts have slowly shifted the focus of the processor design process to a significantly more power aware strategy as opposed to performance centric designs [73].

The semiconductor industry's drive towards aggressive scaling of the silicon technology to match the continuous increase in chip requirements is accompanied by a drastic increase in chips' leakage power consumption. Indeed, as mentioned in Chapter 2 leakage power is slowly dominating the overall power consumption (Fig. 2.3). Accordingly, the concept of predictive sleep signal generation aims to tackle the leakage power management problem while maintaining the processor performance. The objective is to utilize the observed behavior of the running application code to help alleviate the overheads associated with leakage power consumption in modern processors.

4.1 Sleep Signal Generation

Chapter 3 summarized several leakage control techniques that have been proposed to deal with leakage power [4, 65, 66]. In these techniques, when the functional units are not used, an idle signal is sent to a sleep signal generator which is located in the power management unit. The sleep signal generator is responsible for asserting the sleep signal to a leakage control mechanism that sets the functional unit to a low leakage mode (Fig. 3.9). Applying the leakage control mechanism consumes switching power in both

the functional unit and the sleep signal generator circuit. In addition, the application of the leakage control mechanism introduces a delay overhead due to the need to restore the state of the functional unit after the idle period ends. Accordingly, the main challenge for all leakage control techniques is to balance the power and the delay overhead incurred by asserting the sleep signal, with the leakage power savings gained by setting the functional unit to the low leakage mode.

In order to generate the sleep signal, many system level techniques have been proposed. Compiler-based leakage control techniques discussed in Section 3.2.4.1 are based on offline analysis of application code to assist the generation of sleep signal [65, 66]. This is achieved by using the application code to identify the periods in time when the functional unit is not used, and shutting them down accordingly. However, the main drawback of compiler-based techniques is the need to recompile existing applications in order to capitalize on these modifications as well as the enforced hardware backward compatibility that would limit future enhancements [1].

In addition to compiler-based approaches, hardware and software-based techniques that extract major program trends have been studied [63, 67–69]. Although these techniques can dynamically track the utilization of the functional units and in turn assist the task of sleep signal generation, these techniques suffer from large power and delay overhead.

Branch prediction based techniques were studied [4]. In these techniques, when a branch misprediction is encountered, the functional unit is powered down. However, these techniques do not guarantee that the functional unit will stay idle long enough to recoup the power consumed in the shut down period.

Finally, Hu et al. have proposed the use of a simple counter based technique that shuts down the functional unit after being idle for a fixed time interval [4]. Although this technique is characterized by a low power overhead, it suffers from very low accuracy predicting the length of the standby period that will follow.

This research tackles the task of sleep signal generation for microprocessors' functional units. However, compared to earlier attempts that focused on static generation of the sleep signal when the functional unit is idle for a specified number of cycles [4], the proposed technique introduces a dynamic approach that has a better accuracy predicting the length of the sleep period with minimal power overhead. This maximizes leakage savings by applying the sleep signal when it is most likely for the functional unit to stay idle long enough for the power savings to exceed the power overhead introduced by the application of the sleep signal. In this work, the proposed dynamic approach uses an instruction level analysis of the utilization of the functional units. Accordingly, the

functional units are treated as a whole block as opposed to much finer granularity techniques at the sub-block level. The technical contributions in of the dynamic sleep signal generator can be summarized as:

1. The proposed dynamic sleep signal generator (DSSG) is capable of tracking the executed program behavior across different time segments and predicting the length of the standby periods. This translates into high accuracy delaying the generation of the sleep signal when the program is frequently utilizing the functional units. It is also capable of early sleep signal generation when the program rarely utilizes the functional units.

2. Contrary to compiler-based low leakage techniques that are bound to a specific Instruction Set Architecture (ISA), the proposed hardware based DSSG depends on only the information about current and previous standby periods. Accordingly, the DSSG can be easily incorporated in existing microprocessor designs.

3. Compared to program phase extraction techniques that depend on a small memory structure to save program trend information, the DSSG only keeps track of the program behavior through a simple finite state machine. This translates into smaller area and lower power consumption.

4. The DSSG provides a general framework for predicting the length of the standby period independent of the kind of implemented low leakage circuit mechanism. This applies to dynamic low leakage circuit control techniques such as power gating, adaptive body biasing and input vector activation.

5. The DSSG exhibits low power consumption, in the order of 300 μW to achieve accuracies up to 80% in predicting the length of the standby period.

4.2 Proposed Dynamic Sleep Signal Generation

After reviewing the information in Chapter 2, it is clear that all the dynamic leakage control circuit techniques require the external generation of the sleep signal. In addition, high level compiler and phase extraction based techniques are limited in their applicability due to the large overhead they entail. Thus, a fast and power efficient sleep signal generator would be an asset.

Defining the breakeven point as the point in time when the total overheads entering a sleep mode is equated to the total leakage savings achieved, the most simple approach to

deal with sleep signal generation would be the counter-based Static Sleep Signal Generator (SSSG) implemented in [4]. The control state machine of this technique is presented in Fig. 4.1. In this technique whenever the functional unit is idle, a counter is started. When the counter reaches a predefined threshold [1] dictated by the breakeven point, the sleep signal is generated.

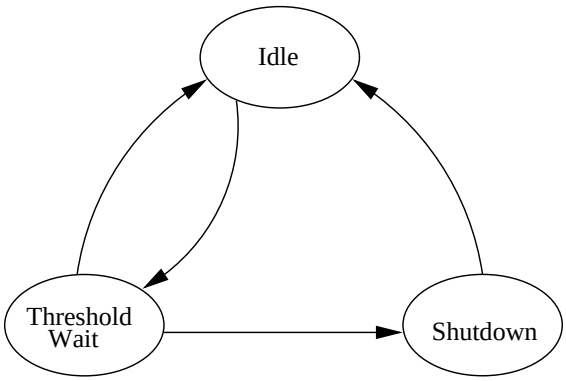
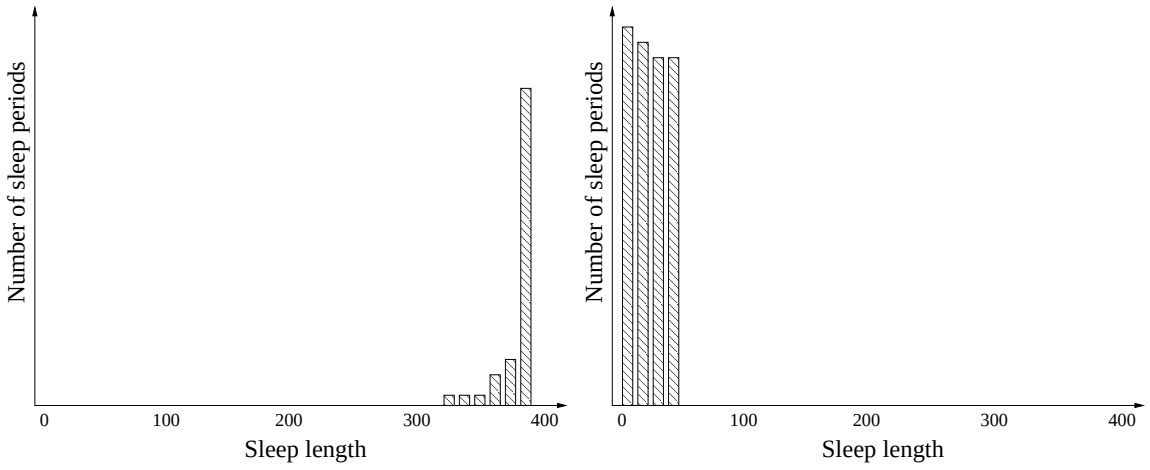


Figure 4.1: Static Sleep Signal Generator state machine.

This technique is caught between two mutually exclusive alternatives. First, use a long threshold and miss potential leakage savings through relatively short standby periods that are longer than the breakeven point. Second, use a short threshold and run at a higher risk of generating the sleep signal for periods that are shorter than the breakeven point. Fig. 4.2 presents a sample histogram of the sleep periods on an SMT processor. Examining the histograms further clarifies the limitation of the static approach. Varying the software mix on the same processor resulted in a widely varying profile. Furthermore, the number of sleep periods that are potentially longer than the threshold but shorter than the breakeven point can be a significant fraction of the overall sleep profile (Fig. 4.3). Accordingly, the technique's inability to adapt the threshold to the running application severely limits the accuracy of such a technique in predicting the length of the sleep period before generating the sleep signal. Hence, this technique is more prone to initiating the sleep cycle for short standby periods that do not reach the breakeven point.

In order to overcome such limitation, the proposed DSSG dynamically changes the threshold according to the requirements of the running application. This approach limits the mutual exclusivity between the two alternatives. When the application is utilizing the functional unit in short repetitive bursts, i.e. the functional units idles frequently but for very short periods, the DSSG raises the threshold to limit the sleep signal generation in this application phase. On the other hand, when the application uses the functional

[1] This threshold should not be confused with the transistor threshold voltage.



(a) 2 threads (b) 4 threads

Figure 4.2: Floating point multiplier unit standby trace histogram. The x-axis represents the sleep length, and the y-axis is the number of sleep instances with the corresponding sleep length.

unit infrequently, the DSSG lowers the threshold to cash-in on the ability to generate the sleep signal early on, maximizing the potential leakage savings.

The DSSG success is based on the fact that application code tends to fall into repetitive patterns. For example, the usage patterns in Table 4.1 and Table 4.2 are repeatedly present in the usage profiles of the integer multiply unit for the GZIP benchmark and the integer divide units for the MESA benchmark, respectively. These patterns are represented by the pair (The length between the successive uses/the length of usage), for each functional unit. In these patterns the functional units are repeatedly utilized in a similar pattern just before an extended sleep period. Due to these repetitive patterns, it is expected that the DSSG stabilizes the threshold above the short standby periods but sufficiently below the long standby periods to maximize the leakage power savings.

4.2.1 The DSSG architecture

In order for the DSSG to predict the length of the standby periods, the DSSG uses a set of internal counters to keep track of how many times the standby period reached the breakeven point and vice versa. Using these counters, the DSSG bases its decision whether to raise the sleep signal, reduce or increase the threshold, on the history of utilization of the functional unit. This enables the DSSG to assert the sleep signal when it is most likely for the idle period to stay beyond the breakeven point.

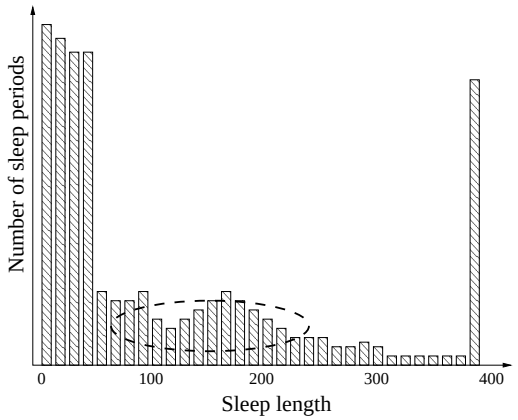


Figure 4.3: Sample standby trace.

Table 4.1: SPEC:GZIP integer multiply usage patterns.

	Event 1	Event 2	Event 3	Event 4
Pattern	12/5	12/5	13/5	12/5
	7/1	7/1	6/1	7/1
	6/1	6/1	6/1	6/1
	3/1	3/1	4/1	3/1
	4/1	4/1	3/1	4/1
	3/1	3/1	3/1	3/1
	4/1	4/1	4/1	4/1
	1726469/3	740910/3	196027/3	1717387/3

Table 4.2: SPEC:MESA integer divide usage patterns.

	Event 1	Event 2	Event 3	Event 4
Pattern	25/1	25/1	19/1	25/1
	19/1	19/1	24/1	19/1
	24/1	24/1	19/1	24/1
	228/1	193/1	258/1	228/1

The idea behind the DSSG can be described using the six operating regions depicted in the state machine shown in Fig. 4.4, and the timing diagram shown in Fig. 4.5.

1. *Idle*: In this state, the DSSG is idle while the functional unit is being used.

2. *Threshold Wait*: This state is initiated when the functional unit idles. In this state, the DSSG counts the number of clock cycles since the functional unit idled and compares them to the threshold. As soon as the threshold is reached, the DSSG

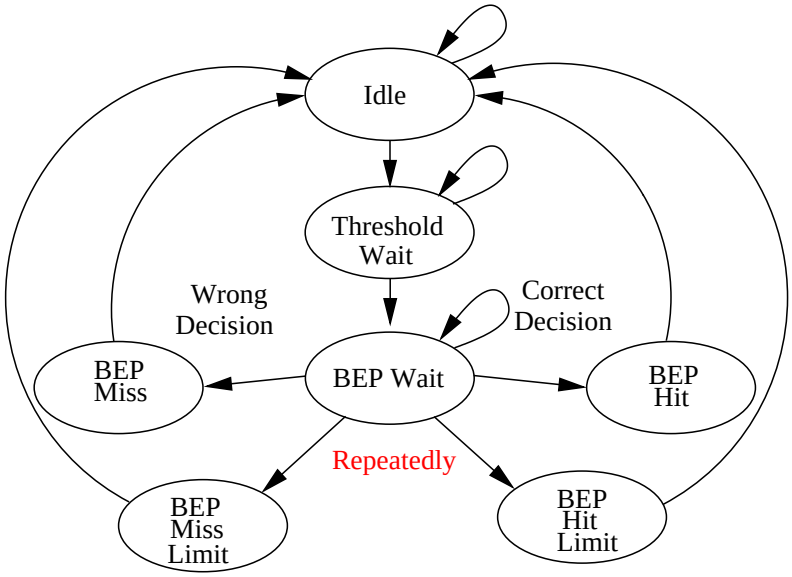


Figure 4.4: Dynamic Sleep Signal Generator state machine.

raises the sleep signal to high. After the sleep signal is asserted the DSSG waits in the *BEP Wait* state until the sleep period either reaches the breakeven point or a pre-breakeven wake-up request is detected.

3. *BEP Hit*: This state is reached when the functional unit stays idle longer than the breakeven point. Whenever the DSSG reaches this state, the total leakage savings exceeds the overheads [2].

4. *BEP Hit Limit*: A small counter is attached to the DSSG to count the number of times the BEP Hit state is reached. When the BEP Hit is reached a predefined number of times denoted by `hit-limit` the DSSG reduces the threshold by a step equal to `step-`.

5. *BEP Miss*: This state is reached if the threshold is reached but the functional unit does not stay in the idle state longer than the breakeven point.

6. *BEP Miss Limit*: Similar to the BEP Hit Limit, the DSSG keeps track of the number of BEP misses. When they reach a predefined value denoted by `miss-limit`, the DSSG increases the threshold by a step that is equal to `step+`.

In order to describe the various parameters of the DSSG, the quartet (`step+`, `step-`, `miss-limit`, `hit-limit`) is used. As an example, if these parameters take the values (5, 20, 3, 15), the operation of the DSSG will proceed as follows:

[2]These overheads include the switching power consumed by both the DSSG and the leakage control technique chosen from the alternatives presented in Section 3.2.2

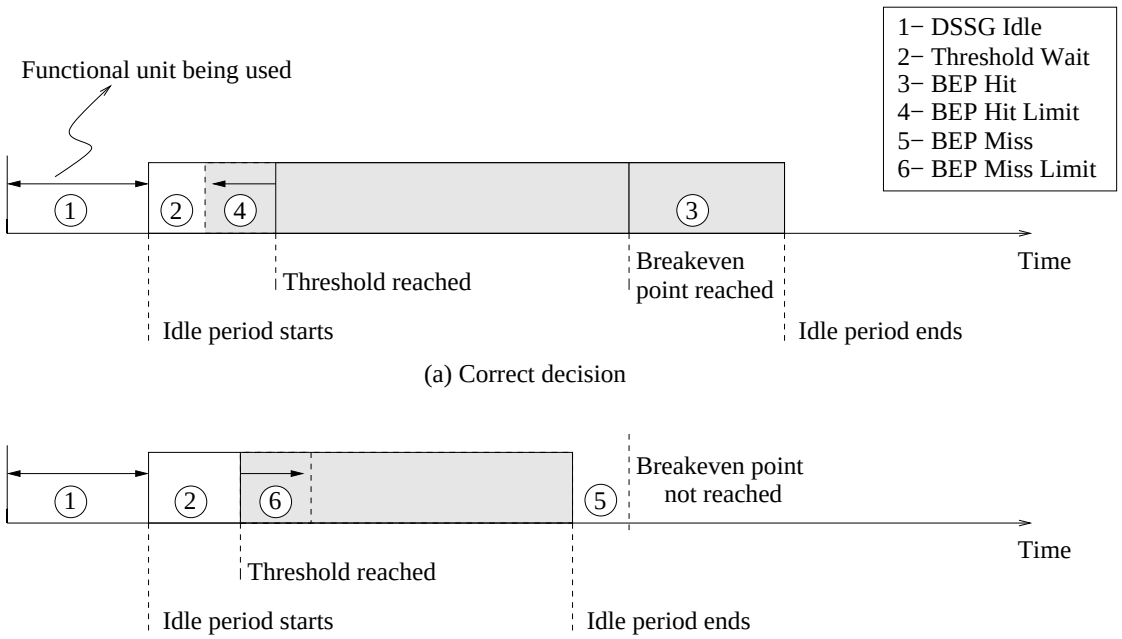


Figure 4.5: The DSSG timing diagram.

- Idle period $>$ breakeven point (Fig. 4.5(a)).
  1. The DSSG starts at state 1.
  2. The idle period starts and the sleep signal is generated at end of state 2.
  3. The idle period stays beyond the breakeven point, and the DSSG goes to state 3.
  4. If state 3 is reached 15 times, the threshold is reduced by 20 cycles (state 4).
- Idle period $<$ breakeven point (Fig. 4.5(b)).
  1. The DSSG starts at state 1.
  2. The idle period starts and the sleep signal is generated at end of state 2.
  3. The idle period does not stay beyond the breakeven point, and the DSSG goes to state 5.
  4. If state 5 is reached 3 times, the threshold is increased by 5 cycles (state 6).

It is important to note that checking if the sleep period reaches the breakeven point is calculated from the start of the idle period independent of the threshold value. This

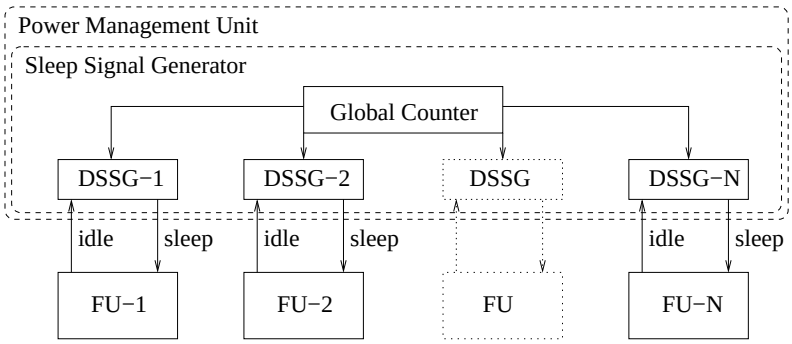


Figure 4.6: The DSSG implementation.

in turn puts an upper bound on the value of the threshold at the breakeven point minus `step+`. Otherwise, the functional unit will miss power saving opportunities if the threshold exceeds the breakeven point.

Implementing the DSSG technique in a microprocessor will require that a DSSG is attached to each functional unit. In order to reduce the switching power overhead introduced by the DSSG, a single global clock counter is used by all the DSSGs in the processor. Accordingly, each DSSG captures the value of the global counter at the beginning of the Threshold Wait state. The DSSGs can then be implemented as shown in Fig. 4.6.

In order to assess the potential of the DSSG scheme presented so far, the following Sections presents the experimental results of applying the DSSG to the SPEC benchmarks using the superscalar SimpleScalar simulator [70].

4.3 Sleep Signal Generation for Superscalar Processors

In order to compare the DSSG and the SSSG, three major issues need to be tackled. Firstly, the accuracy of the DSSG and the SSSG in matching the sleep signal assertion to the breakeven point of the functional units needs to be investigated. However, adding a sleep signal generator to the functional units affects the value of the breakeven point by adding to the overhead of the sleep cycle. Accordingly, to guarantee that the total power is reduced due to the savings achieved by the DSSG the design and the power consumption of the DSSG versus the SSSG is the second issue that needs to be analyzed. In the following sections, the design of both the DSSG and the SSSG using a 90nm process will be described with special emphasis on the dynamic power consumption overhead. Thirdly, this will be followed by the analysis of the accuracy of the DSSG versus the SSSG in predicting the length of the sleep period and the ability to match the

sleep signal generation to the different applications being executed while emphasizing the potential leakage savings achievable for both techniques.

4.3.1 Superscalar Simulation Environment

To test the sleep signal generation schemes, the SimpleScalar [70] simulator is modified to report the usage profiles of the various functional units. These simulations use a MIPS-like architecture as a test vehicle. After the modified SimpleScalar simulator is presented, the test microprocessor architecture is described.

4.3.1.1 Modified SimpleScalar

The SimpleScalar tool set includes an open source processor simulator, a cross compiler, and a set of precompiled test executables [70]. The simulator implements the SimpleScalar architecture, which is a close derivative of the MIPS architecture [74]. The tool set provides a set of processor simulators that range from simple, `sim-fast`, to the more complete, `sim-outorder`. `Sim-outorder` is a superscalar, O-o-O simulator that is capable of performing cycle-by-cycle instruction simulation. In addition, `sim-outorder` supports speculative execution.

The `sim-outorder` simulator is modified in order to track the usage patterns, associated with the various function units.

4.3.1.2 `Sim-outorder`

The `sim-outorder` simulator depends on the Register Update Unit (RUU) [75] to support the out-of-order issue and execution. The RUU uses a reorder buffer to perform register renaming and to hold the results of the pending instructions; then, in each cycle, the reorder buffer retires the completed instructions in-program order. This architecture is similar to the one described earlier in Section 3.1.

The simulator memory system employs a load/store queue that allows for speculative STOREs. The LOADs are satisfied either from the memory system or through an earlier STORE value in the queue, if the addresses match. LOADs are dispatched to the memory when the addresses of all the previous STOREs are known, preventing the reading of obsolete data.

4.3.1.3 `Sim-outorder-trace`

In order to perform the tests and extract profile information, the `sim-outorder` main execution loop, given next, is slightly modified based on the power update pointers extracted from the Wattch SimpleScalar based simulator [76].

```
%ruu_init();
for (;;)
{
    ruu-commit();
    ruu-writeback();
    lsq-refresh();
    ruu-issue();
    ruu-dispatch();
    ruu-fetch();
}
```

This loop is executed for each simulated machine cycle. When the simulated program ends by issuing an `exit()`, this loop is broken by a `longjmp()` enabling the `main()` to generate the program usage statistics. Note that the loop pipeline is traversed in the reverse order to enable the simulator to handle the inter-stage latch synchronization correctly in one pass [70]

To extract the usage profiles, the `ruu-issue()` function is modified by adding software probes that can identify the used/not used state of each of the functional units available to the processor. The `ruu-issue()` is selected because it contains information about the availability of the functional units, in addition to the status of the memory dependencies of the simulated instruction. The software probes keep track of the successive accesses of the various functional units in the processor, recording the time span of usage and the time difference between each two successive accesses to the functional unit.

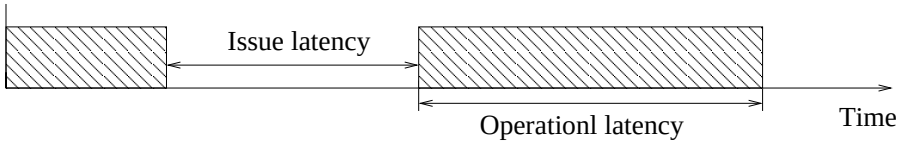


Figure 4.7: Issue and operational latency.

The `sim-outorder-trace` uses the `sim-outorder` resource pool definition of the functional units. This definition is given in Table 4.3, where the operational latency

is the time between the instruction issuance and the availability of the results, whereas the issue latency is the minimum time between each two successive instructions (Fig. 4.7).

Table 4.3: Functional units in the superscalar test processor.

Functional Unit	Operational Latency (cycles)	Issue Latency (cycles)	Available Units
Integer ALU	1	1	4
Integer Multiply	3	1	1
Integer Divide	20	19	1
Floating-Point Adder	2	1	4
Floating-Point Multiply	4	1	1
Floating-Point Divide	12	12	1
Floating-Point SQRT	24	24	1

To configure the remaining options of the test processor, the simulator is setup with the model parameters of Table 4.4. These parameters are chosen to be similar to the test processor in [5].

The following section reveals the test results of running the modified simulator on the various benchmarks and applying sleep signal generation to the extracted traces.

4.3.2 DSSG and SSSG Circuit Implementation and Power Consumption

In order to design the DSSG, the MIPS-like processor architecture shown in Table 4.4 is assumed. The execution core of this processor contains the functional units in Table 4.3. Besides the DSSG, the SSSG is designed assuming the same processor architecture and execution core. This ensures the fairness of the comparison between both techniques with respect to the associated power overhead.

The following section describes the design of both the SSSG and the DSSG while Section 4.3.2.2 presents the test structures that are used to emulate the functional units' circuits for the assessment of the breakeven point variation due to the introduction of the sleep signal generators. Finally, Section 4.3.2.3 compares the DSSG and the SSSG with respect to their power consumption overhead.

Table 4.4: Superscalar test processor architecture.

Parameter	Value
Fetch Queue	8 entries
Branch predictor	comb of bimodal and 2-level gshare; bimodal/gshar level 1/2 entries- 2048, 1024 (hist. 10), 4096 (global), resp.; Combining pred. entries - 1024; RAS entries - 32 ; BTB - 4096 sets, 2-way
Branch misprediction latency	10 cycles
Decode and Issue Width	4 instructions
Reorder Buffer	128
Load Entries	32
Store Entries	32
Instruction TLB	256 entry 4-way, 8K pages, 30 cycles miss
Data TLB	512 entry 4-way, 8k pages, 30 cycles
Memory Latency	80 cycles
L1 I-Cache	64KB, 4-way, 64B line, 2 cycles
L1 D-Cache	64KB, 4-way, 64B line, 2 cycles
L2 Unified	2 MB, 8-way, 128B line, 12 cycles

4.3.2.1 SSSG and DSSG circuits for Superscalar Processors

In order to compare the SSSG and the DSSG, both sleep signal generators are custom designed using the STMicroelectronics 90nm CMOS process. Both designs are set to run at 500MHz and 2GHz to match the implementation of the functional units that will be discussed in Section 4.3.2.2. Additionally, to further ensure fairness, both the DSSG and the SSSG adopted the structure of the power management unit shown in Fig. 4.6. This design divides the impact of the global counter among the different functional units that need to be managed through sleep signal generation.

Fig. 4.8 shows the internal design of the DSSG. In this design, the circuit operation of the DSSG closely matches the finite state machine depicted in Fig. 4.4. As soon as the functional unit idles, an idle signal goes high triggering the DSSG to capture the global counter into a local register. The DSSG then uses an XOR array to compare the value of the global counter to that of the local register. When the number of clock cycles elapsed since idle went high is equal to the current threshold, the DSSG raises the sleep signal. If the functional unit stays idle beyond the breakeven point, the DSSG updates its

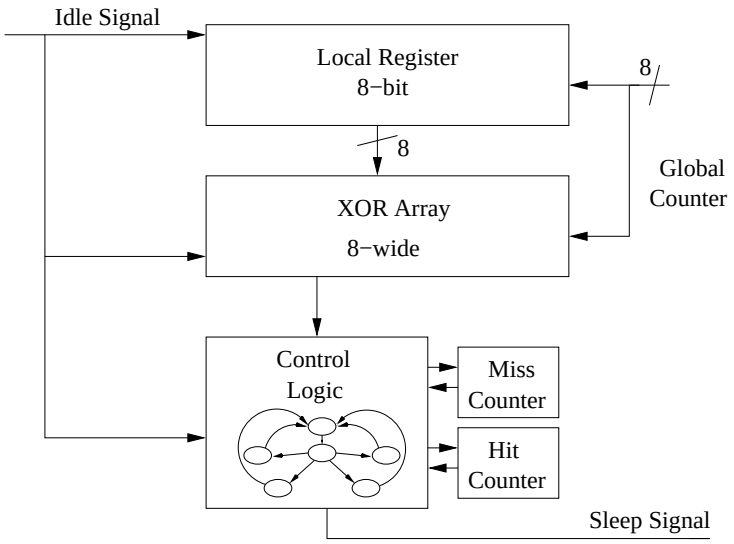


Figure 4.8: DSSG circuit implementation.

hit counter; otherwise the miss counter is incremented. When the idle signal goes low, the DSSG checks to see if either the hit or miss counters reached their respective limits (`hit-limit` and `miss-limit`), and updates the states accordingly by reducing or incrementing the threshold.

On the other hand, the SSSG has the simpler structure shown in Fig. 4.9. In this design, similar to the DSSG, the operation starts when the idle signal goes high. As soon as the idle signal goes high, the SSSG local register captures the counter signal and compares it using an XOR array. When the number of elapsed cycles since the idle signal went high is equal to a predefined threshold, the SSSG raises the sleep signal.

Since the XOR array represents an important component affecting both the speed and the power consumption of the sleep signal generators, the low power 8 transistors XOR presented in [77] is used. In addition, all the internal counters in the DSSG are ripple counters, since they are not on the critical path. This allows the DSSG to consume less dynamic power[3]. The rest of the design is implemented in standard static CMOS.

4.3.2.2 Functional Units Emulation

In order to fully quantify the impact of the DSSG versus the SSSG on the sleep signal generation process, the breakeven point of the functional units in Table 4.3 needs to be identified. This is achieved using two different testbenches. The first testbench (FO4)

[3]Flip-flops do not receive a direct clock signal, mimicking the effect of clock gating.

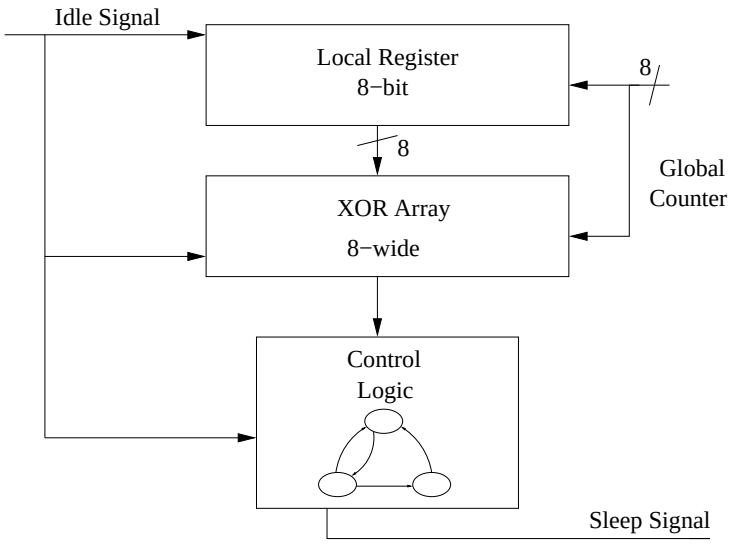


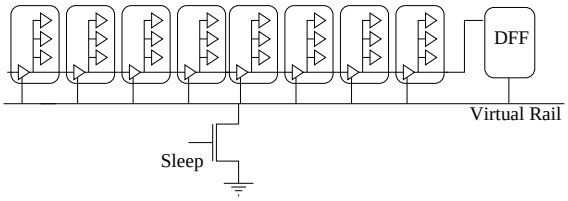
Figure 4.9: SSSG circuit implementation.

uses FO4 inverter chains to emulate the functional units. The second testbench (FPU100) is a standard cell implementation of a floating point unit based on the work in [51].

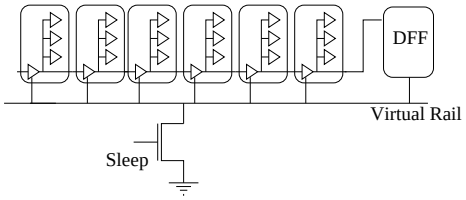
FO4 Inverter Chain Implementation (FO4) FO4 inverter delay is a widely used delay metric that is used to identify various parameters of VLSI circuits [78, 79] and processor performance [80]. Additionally, FO4 inverter chains are further used for power optimal pipelining [81]. According to Hrishikesh et al. in [80] the optimum clock period is 8 FO4 inverter delays for an integer unit, and 6 FO4 inverter delays for a floating point unit. Based on this conclusion, the structures shown in Fig. 4.10 are used to emulate the integer and floating point functional units in Table 4.3. These structures correspond to a single pipe stage each. In these structures, the inverter chains are implemented with devices having low threshold voltage, while the sleep transistor connected between the virtual rail and the ground is a high threshold device. The inverter chains are sized for 500MHz and 2GHz operation. Additionally, the sleep transistor is sized to ensure that the maximum delay penalty due to the introduction of the series resistance is less than 3% [82]. This sizing also ensures that the maximum ground bounce is less than 30mV.

Assuming that the functional units are designed for 32-bit operation, the structures in Fig. 4.10 are replicated in parallel 64 times for each functional unit representing the 64 inputs that will be needed for the operation. Finally, for each functional unit, the operational latency in Table 4.3 is used as an indicator for the pipeline depth[4]. For example, the floating point multiply has an operational latency of 4 cycles; this corresponds

[4]The FP SQRT and Integer and FP divide are not fully pipelined



(a) Integer Unit



(b) Floating Point Unit

Figure 4.10: FO4 inverter chain for integer and floating point units representing 1 pipe stage.

Table 4.5: Breakeven point of testbench functional units without sleep signal generators' overhead.

Functional Unit	Testbench	Breakeven Point (cycles)	
		0.5GHz	2GHz
Integer ALU	FO4	38	136
Integer Multiply		37	134
Integer Divide		33	120
Floating-Point Adder		50	178
Floating-Point Multiply		46	167
Floating-Point Divide		44	160
Floating-Point SQRT		39	142
Floating-Point Adder	FPU100	22	-
Floating-Point Multiply		25	-
Floating-Point Divide		49	-
Floating-Point SQRT		11	-

to four stages from Fig. 4.10-b cascaded in series. The breakeven point of each of the integer and floating point units without the overhead from the sleep signal generator is highlighted in Table 4.5. This corresponds to the overhead of shutting down the sleep transistor network compared to the leakage savings that can be achieved.

Standard Cell Based FPU (FPU100) The second testbench is based on the design of

the 32-bit FPU in [51]. The FPU is subdivided into four subcomponents representing the floating point add, multiply, divide, and square root operations. The design is synthesized using Synopsys design compiler[5] and the $90nm$ STMicroelectronics standard cell library. Because the FPU100 critical path has 37 gates, the maximum operational frequency is 500MHz. The second part of Table 4.5 also includes the breakeven point of the second testbench without the sleep signal generators' overhead.

4.3.2.3 Superscalar DSSG and SSSG Power Consumption Overhead Comparison

Table 4.6: Power consumption of the sleep signal generators.

Frequency	DSSG (μW)	SSSG (μW)	Global Counter (μW)
2GHz	266	104	277
500MHz	184	48.3	143

In order to compare the power consumption overhead between the DSSG and the SSSG, Table 4.6 establishes the baseline by summarizing the dynamic power consumption of both generators extracted from spice simulations. Comparing the dynamic power of both generators to a modern 64-bit ALU in the dual core UltraSPARC microprocessor running at approximately 400 mW[6] highlights the minimal impact that both techniques have on the power budget of the processor core [60].

However, more importantly the impact of both sleep signal generators on the breakeven point is given in Table 4.7. Additionally, the table shows the percentage increase in the breakeven point value comparing both generators combined with their respective global counters.

The breakeven point is calculated using the following equations.

$$Breakeven\ (Cycles) = E_{overhead}/E_{saved} \tag{4.1}$$

where

$$E_{overhead} = P_{STN} \times T_{trans} + P_{sleep} \times T_{cycle} \tag{4.2}$$

$$E_{saved} = P_{leakage\ savings} \times T_{cycle} \tag{4.3}$$

[5]Although the standard cells are rarely used in processor functional units, the use of such emulation platform gives an important insight on the power trade-off between the SSSG and the DSSG

[6]Calculated based on 2 ALUs consuming approximately 16% of the 5.3 Watt core power.

Table 4.7: Breakeven point of testbench functional units with sleep signal generators' overhead.

Functional Unit	Testbench	Breakeven Point (cycles)		Percentage Increase
		SSSG	DSSG	
Integer ALU	FO4 @ 2GHz	197	266	34.86
Integer Multiply		155	177	14.66
Integer Divide		123	126	2.4
Floating-Point Adder		219	264	20.74
Floating-Point Multiply		187	210	11.97
Floating-Point Divide		166	173	4.29
Floating-Point SQRT		145	148	2.13
Integer ALU	FO4 @ 0.5GHz	67	125	86.38
Integer Multiply		47	66	40.63
Integer Divide		35	37	7.16
Floating-Point Adder		69	107	55.5
Floating-Point Multiply		56	75	33.7
Floating-Point Divide		47	53	12.67
Floating-Point SQRT		41	43	6.37
Floating-Point Adder	FPU100 @ 0.5GHz	25	31	25.26
Floating-Point Multiply		26	29	11.98
Floating-Point Divide		53	60	13.75
Floating-Point SQRT		13	15	21.71

where P_{STN} is the total power to shutdown and turn on the sleep transistor network, T_{trans} is the total time needed to fully turn ON or OFF the sleep transistor network, P_{sleep} is the total dynamic power needed to generate the sleep signal for either sleep signal generators and $P_{leakage\ savings}$ is the difference between the leakage of a non-power gated and power gated functional unit.

Additionally, the breakeven point is calculated assuming that there are 7 functional units to control, where the overhead of each sleep signal generator is calculated using the following equation.

$$P_{sleep} = P_{counter}/N + P_{SSG} \tag{4.4}$$

where N is the number of functional units to be managed, $P_{counter}$ is the dynamic power of the unified counter and P_{SSG} is the dynamic power of the sleep signal generator. Table 4.7 shows that the breakeven point can increase between 2 to 86% by applying

the DSSG instead of the SSSG. However, this piece of information does not allow for a complete comparison between both generators, since it ignores an important factor which is the accuracy of each generator in predicting the standby period length. This accuracy is a major factor, since a low overhead operation that is 40% accurate, is worse than a slightly higher overhead operation that is 80% accurate. In order to complete the comparison, the following section compares the operation of both sleep signal generators under SPEC2000 benchmarks workload to identify their respective accuracies.

4.3.3 Accuracy of the Sleep Signal Generators on Superscalar Processors

This section will discuss the accuracy of both techniques with special emphasis on the implications of the breakeven points presented earlier in Table 4.7. In order to calculate the accuracy of the DSSG, the SimpleScalar simulator is modified to report the usage data of each functional unit. Additionally, to match the earlier power analysis the SimpleScalar is configured with the same architecture as in Table 4.4, while the execution core contains the functional units in Table 4.3.

Using this processor model, the modified superscalar out-of-order processor implemented in SimpleScalar is used to trace the usage profiles of the processor's various functional units for the SPEC2000 benchmarks. Since the full SPEC2000 inputs require very long simulation times, the tests use the MinneSPEC [83] input workload to reduce the run time. Table 4.8 indicates the subset of the SPEC benchmarks used and the corresponding instruction count.

Using the traces extracted from the modified SimpleScalar, the proposed DSSG and the SSSG in [4] are implemented in C++. For both techniques the breakeven point is varied from 10 to 290 cycles. These values correspond to the breakeven points extracted earlier from the circuit simulations of both the DSSG and the SSSG in Section 4.3.2. Additionally, these values explore the design space for all the functional units in the test processor shown in Table 4.3 while maintaining an 8-bit wide counter as a test vehicle similar to Fig. 4.8 and Fig. 4.9.

Comprehensively comparing the results of the DSSG and the SSSG, regarding their abilities to predict the length of the standby period ahead of time, requires a common accuracy metric. The accuracy is defined as the percentage of the correct decisions to the total number of decisions made by the sleep signal generator. In both the SSSG and the DSSG, a correct decision is made when the sleep signal generator raises the sleep signal and the functional unit indeed stays idle beyond the breakeven point. This is considered

Table 4.8: SPEC2000 benchmarks.

Benchmark Name	Number of Instructions (Billions)	Nature
ART	7.7	Floating-Point
APPLU	0.95	Floating-Point
BZIP2	15.2	Integer
GCC	6.4	Integer with some floating-point operations
GZIP	12.2	Integer
MCF	1.7	Integer
MESA	1.3	Floating-Point
PARSER	5.6	Integer
SWIM	1.3	Floating-Point
VORTEX	1.5	Integer
VPR	5.3	Integer with many floating-point operations

a hit. The total number of decisions includes all the hits and misses. Misses are encountered when the sleep signal generator raises the sleep signal but the functional unit does not stay idle beyond the breakeven point. In effect, the accuracy can be calculated as:

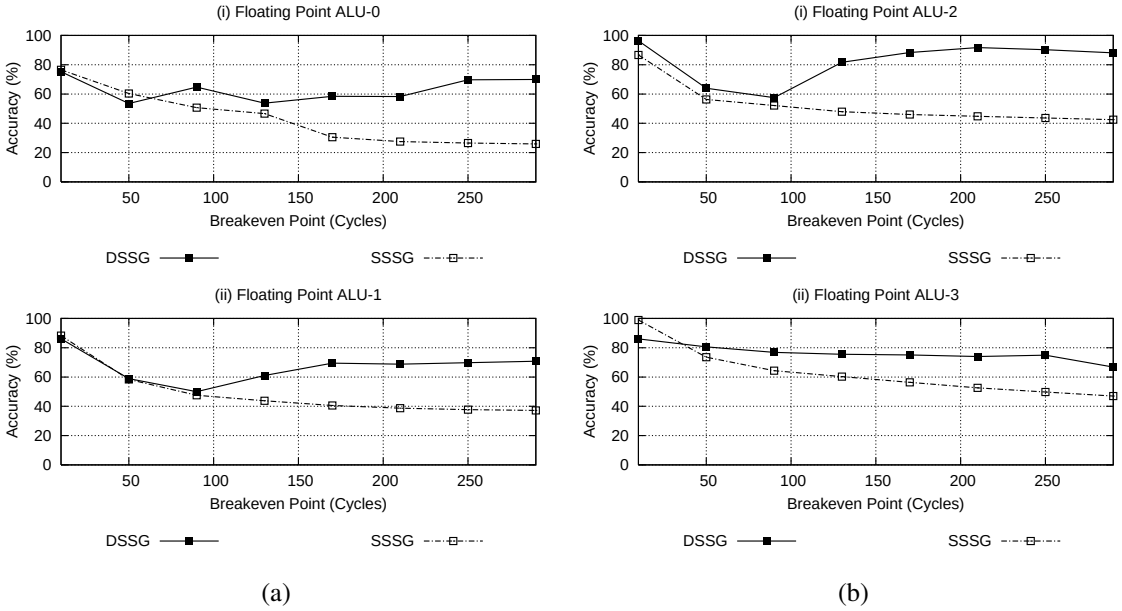
$$\text{Accuracy } (\%) = \frac{\text{Number of hits}}{\text{Number of hits + Number of misses}} \tag{4.5}$$

In order to compare the SSSG and the DSSG techniques, the results for both techniques for the floating point ALUs, the integer and floating point multiply and divide will be presented in the following sections.

4.3.3.1 Floating Point ALU

Testing the floating point ALU required the simulator to track the (FP) ADD/SUB operations as well as the integer to floating-point conversion and floating-point comparison. Using the accuracy metric in Equation (4.5), Fig. 4.11 depicts the results of applying both SSSG and DSSG to the standby traces of all the floating point ALUs. Fig. 4.11 shows that the DSSG maintains a comfortable lead over the SSSG across a wide range of breakeven points.

As a further example, Fig. 4.12 shows the results for floating point ALU-2, the vertical dashed lines correspond to the DSSG and SSSG breakeven points from Table 4.7



(a) (b)

Figure 4.11: DSSG and SSSG average prediction accuracy for the floating point ALUs.

while the horizontal arrows define the direction of the increase in the breakeven point.

It is clear that the tremendous increase in the prediction accuracy for the DSSG especially at higher breakeven points overshadows the increase in the breakeven point due to the DSSG overhead. For example, the floating point ALU with SSSG breaks even at 219 cycles, with prediction accuracy of 45%. On the contrary, with the DSSG it breaks even at 264 cycles with prediction accuracy of 90%. This is one of the major advantages of the DSSG, since it is capable of compensating for its misses by manipulating the sleep generation threshold.

4.3.3.2 Integer and Floating Point Multiply/Divide

Figs. 4.13(a) and 4.13(b) show the maximum achievable accuracy running both the SSSG and the DSSG on the integer and floating point multiply and divide. Similar to the previous graphs the vertical lines correspond to the breakeven points in Table 4.7. These units are generally less utilized compared to the integer and floating point ALUs. Less utilized functional units stay idle longer and hence result in better hit to miss ratio. These results are consistent with the DSSG behavior for the ALUs. Applying the DSSG slightly increases the breakeven point, but tremendously enhances the quality of the standby prediction. However, it is interesting to note that DSSG advantage diminishes for lower frequencies and the associated lower breakeven points. Fig. 4.13(a) shows that the DSSG and the SSSG are equal in their prediction accuracies for such low breakeven points. This

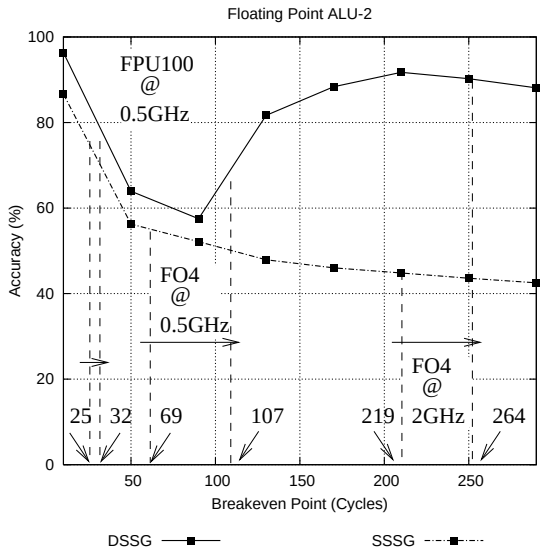
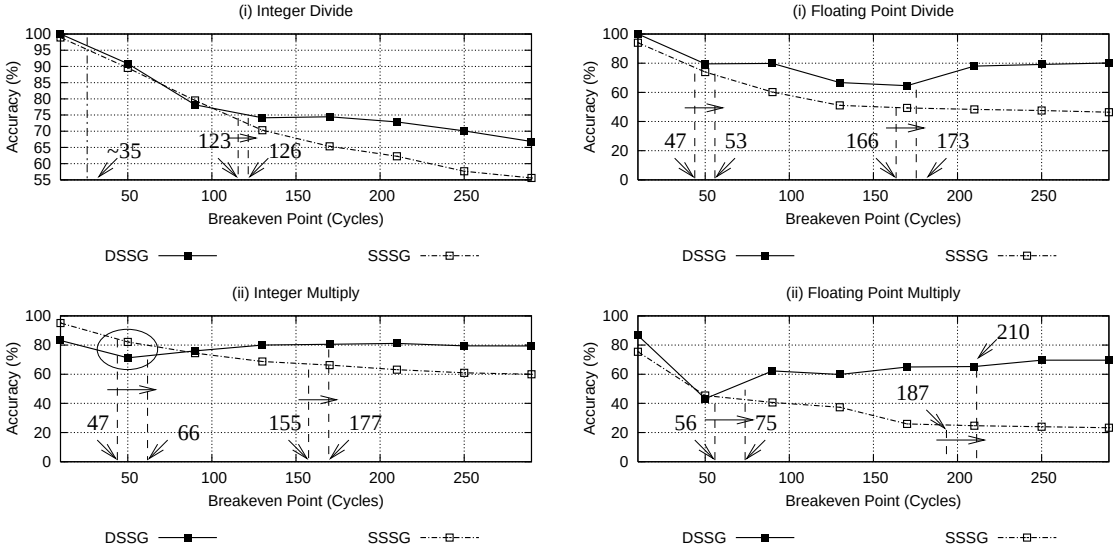


Figure 4.12: DSSG and SSSG average prediction accuracy for the floating point ALU 2. Vertical lines correspond to breakeven points from Table 4.7 and the horizontal arrows depict the changes in the breakeven points.



(a) Integer multiply and divide.

(b) Floating point multiply and divide

Figure 4.13: Average DSSG and SSSG prediction accuracy. Vertical lines correspond to breakeven points from Table 4.7. The arrows depict the changes in the breakeven points.

can be attributed to the small threshold needed to reach such breakeven point leaving the SSSG with less mistakes compared to the higher breakeven points.

4.3.4 DSSG and SSSG Design Issues

In order to determine the maximum accuracy for both the DSSG and SSSG in the tests in Sections 4.3.3.1 and 4.3.3.2, the four parameters `step+`, `step-`, `miss-limit`, and `hit-limit` for the DSSG are required. Accordingly, the simulations are designed to explore the impact of the variation of these parameters in the range between 1 and 20. This range results in the most accurate predictions using the DSSG. On the other hand, for the SSSG, the threshold is varied between 10 and 80 cycles in steps of 10 cycles. Beyond 80 cycles the SSSG power saving potential as discussed in Section 4.3.5 is dropped significantly since it eliminates more sleep periods that would have been otherwise included.

The final choice of the DSSG parameters is part of the DSSG design process which can only be settled during the actual design of the processor. In order to explore the impact of these parameters on the DSSG each parameter is given the values 1,5 and 20 yielding 81 combinations. Fig. 4.14 shows the impact of changing the `step+`, `step-`, `miss-limit` and `hit-limit` on the DSSG accuracy by presenting the five best combinations. Across all the functional units the quartets (20 5 1 1) and (20 20 1 1) leads to the highest DSSG accuracy. At this configuration, the DSSG quickly increases the threshold after a miss and a hit with slightly varying step sizes. The final values for these parameters should be assigned based on each functional unit's breakeven point.

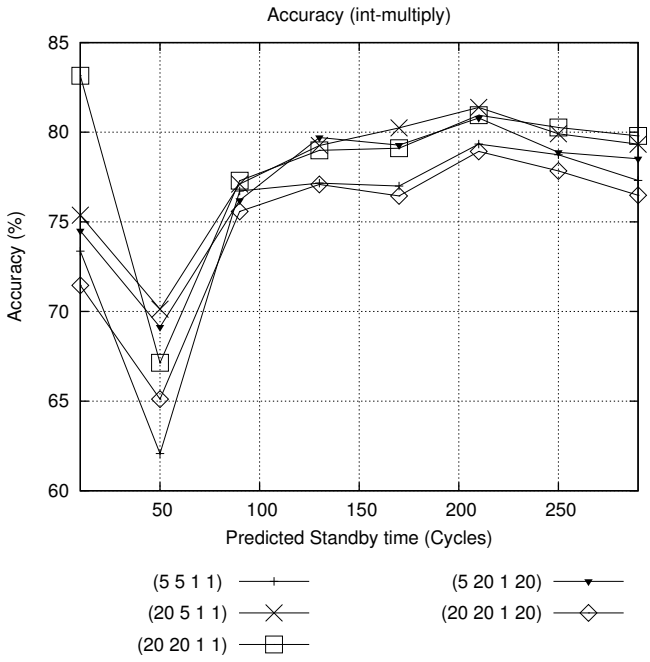


Figure 4.14: Impact of DSSG parameters on accuracy.

In addition, as an example, Fig. 4.15 visualizes the dynamic threshold trends throughout the execution of the MESA benchmark program. This graph is compiled by sampling the traces of the MESA program for the integer ALU-3 unit. In this graph, the breakeven point is hypothetically set at 250 cycles. Each point in this graph represents a change in the threshold, hence stable threshold periods are only represented by a single point. It is instructive to see that the DSSG adapts the threshold depending on the program execution phase.

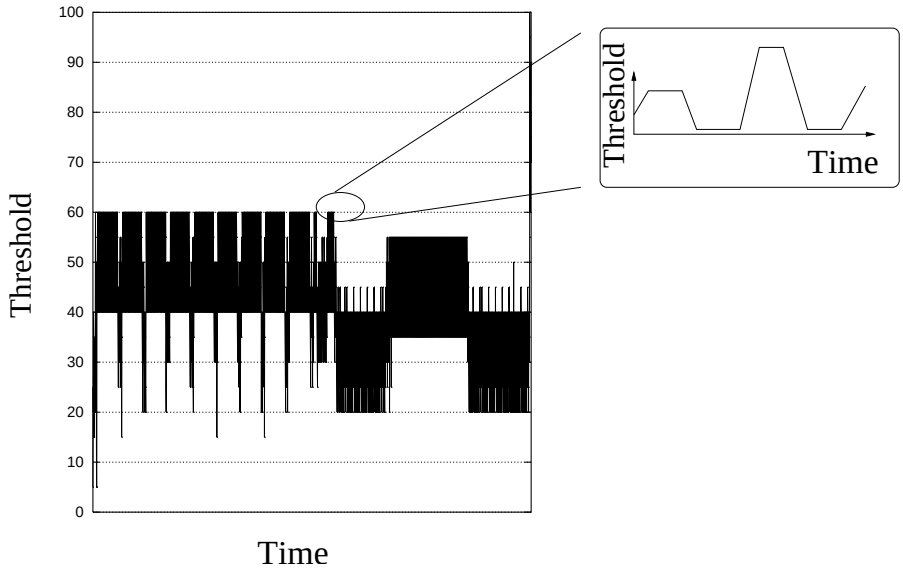


Figure 4.15: The threshold value for the integer ALU unit running the MESA program.

Finally, the results show that although it is tempting to quickly assert the sleep signal for the functional units when they are idle to maximize the leakage savings, the total power overhead quickly increases when the functional units regularly fail to stay idle beyond the breakeven point. Accordingly, the ability of the DSSG to postpone the assertion of the sleep signal when it is less likely for the functional unit to stay idle beyond the breakeven point greatly enhances the leakage saving performance of power gating techniques.

4.3.5 Superscalar Leakage Saving Potential for Sleep Signal Generators

A detailed discussion of the DSSG will not be complete without describing the power saving potential of such technique. Referring to the definition of a hit as a correct decision, and a miss as a wrong decision, each hit or miss will be associated with a number of

cycles that correspond to the total power savings or losses. For a hit, any cycles beyond the breakeven points adds to the total savings, while for a miss any cycles between the pre-breakeven wake-up and the breakeven point correspond to a total loss. Assuming constant leakage savings per power-gated cycle the Hit-Cycle-Count (HCC) and Miss-Cycle-Count (MCC) can be defined as follows

$$HCC = Sleeplength - BreakevenPoint \tag{4.6}$$

$$MCC = BreakevenPoint - Sleeplength \tag{4.7}$$

where $Sleeplength$ is the actual number of cycles the functional units stayed idle, HCC will correspond to the total leakage savings represented in number of cycles beyond the breakeven, and MCC will correspond to the total power losses due to wake-up before the breakeven. Subtracting the MCC from the HCC shows the worthiness of either the DSSG or the SSSG. A positive number indicates net power savings and vice versa. Also a larger positive number indicates better power savings. Accordingly, the net savings of either sleep signal generators can be calculated by computing

$$NetSavings = HCC - MCC. \tag{4.8}$$

Using this equation in conjunction with the traces extracted from the processor simulator and the breakeven points presented in Table 4.7 for the 2GHz implementation of the functional units, Table 4.9 presents the HCC and MCC running all the benchmarks in series. The table also presents the net savings as a number of cycles for each sleep signal generator besides the ratio of savings of the DSSG to the SSSG. Although it is expected that the results will vary depending on the software benchmark mix, the trends in Table 4.9 should be maintained across different benchmark mixes. This is expected since both the DSSG and the SSSG have a very limited history dependence, which translates into results that are mostly independent of the order the benchmarks are run.

Table 4.9 highlights many important aspects regarding the operation of the SSSG versus the DSSG. Firstly, the results of the net savings in the dark gray cells show that neither sleep signal generators are usable for the integer ALU 0 or 1, both sleep signal generators consume more power than their respective savings. However, for integer ALU 2 and 3 the SSSG still fails to achieve any saving while the DSSG has a considerable ability to achieve net positive power savings. Secondly, for most floating point units the DSSG outperforms the SSSG in its power savings capability achieving up to 80% more savings for the floating point adder as shown in the Saving Ratio Column. Finally, for several benchmarks the total savings in terms of number of cycles is the same (Saving

Table 4.9: Leakage savings potential for SSSG and DSSG.

Functional Unit	Number	SSSG			DSSG			Saving Ratio
		HCC (cycles)	MCC (cycles)	Net Savings (cycles)	HCC (cycles)	MCC (cycles)	Net Savings (cycles)	
Integer ALU	1	96417	5.57E8	-5.57E8	2870	3.80E5	-3.77E5	-
	2	1.08E7	6.80E8	-6.69E8	9.50E6	2.04E6	7.46E6	-
	3	8.24E8	1.39E9	-5.61E8	7.87E8	3.71E7	7.50E8	-
Integer Multiply	-	2.48E10	8.10E8	2.40E10	2.43E10	3.09E8	2.40E10	1
Integer Divide	-	2.69E10	4.49E8	2.64E10	2.68E10	2.46E8	2.65E10	1
Floating-Point Adder	0	4.48E9	2.38E9	2.10E9	4.19E9	4.08E8	3.78E9	1.8
	1	1.09E10	2.18E9	8.70E9	9.19E9	5.97E8	8.60E9	0.99
	2	1.22E10	6.71E8	1.16E10	1.23E10	5.40E7	1.22E10	1.06
	3	1.36E10	3.48E8	1.33E10	1.36E10	4.37E7	1.35E10	1.02
Floating-Point Multiply	-	1.12E10	2.60E9	8.61E9	1.04E10	5.24E8	9.85E9	1.14
Floating-Point Divide	-	1.48E10	2.88E8	1.45E10	1.44E10	1.09E8	1.43E10	0.98
Floating-Point SQRT	-	8.06E9	3.40E5	8.06E9	8.06E9	3.65E3	8.06E9	1

Ratio = 1). However, careful consideration of the values of the MCC for each sleep signal generator shows that the DSSG achieves the same leakage savings with proportionally less mistakes than the SSSG. This is further confirmed by the results of Table 4.10 which shows the number of misses for each functional unit and the ratio between the SSSG and the DSSG. The lower number of misses associated with the DSSG has an important effect on the total performance of the functional unit, less misses means lower ground and supply rail bounce, which in return considerably reduces the number and values of area and power hungry decoupling capacitances [29]. In addition, the lower number of misses translates into higher performance due to the reduction of the number of wake-up events that invariably will affect the IPC of the processor due to the unnecessary unavailability of the functional units.

Evaluating the exact value of the leakage savings depends heavily on the value of the leakage savings per cycle. Using the emulation structures of FO4 that were introduced earlier in Section 4.3.2.2, these values are listed in Table 4.11. These values correspond to the subtraction of the leakage power when the functional unit is not power gated versus a power gated unit. This Table also multiplies these leakage savings per cycles by the net savings presented in Table 4.9 which correspond to the number of cycles with real savings after removing the overheads and the clock period to show a sample of the achievable leakage energy savings. Research in [3] shows that power gating of the functional units can save up to 18% of the total dynamic power. Since the floating point units in the UltraSPARC presented in [60] consume up to 22% of the core's dynamic power, power gating these functional units can save approximately 4% of the core's power.

The last aspect that should be considered for evaluating the performance of the DSSG versus the SSSG is the penetration ratio. The penetration ratio is defined as the ratio of

Table 4.10: Miss ratio.

Functional Unit	#	Saving Ratio	SSSG Misses	DSSG Misses	Miss Ratio
Integer ALU	1	-	2.89E6	2060	1402.77
	2	-	3.61E6	11940	302.14
	3	-	7.76E6	2.33E5	33.26
Integer Multiply	-	1	7.97E6	2.98E6	2.67
Integer Divide	-	1	6.39E6	3.16E6	2.02
Floating-Point Adder	0	1.8	1.65E7	2.56E6	6.46
	1	0.99	1.61E7	4.22E6	3.81
	2	1.06	3.63E6	4.39E5	8.27
	3	1.02	2.00E6	2.93E5	6.85
Floating-Point Multiply	-	1.14	2.21E7	4.55E6	4.86
Floating-Point Divide	-	0.98	2.59E6	1.03E6	2.51
Floating-Point SQRT	-	1	2978	312	9.54

Table 4.11: Leakage savings for the DSSG.

Functional Unit	#	Leakage Savings per Cycle (μW)	DSSG		SSSG	
			Net Savings (Cycle)	Total Energy Savings (μJ)	Net Savings (Cycle)	Total Energy Savings (μJ)
Integer ALU	1	2.35	-3.77E5	-	-5.57E8	-
	2	2.35	7.46E6	0.01	-6.69E8	-
	3	2.35	7.50E8	0.88	-5.61E8	-
Integer Multiply	-	7.1	2.40E10	85.25	2.40E10	85.25
Integer Divide	-	54.7	2.65E10	725.89	2.65E10	725.89
FP Adder	0	3.55	3.78E9	6.71	2.10E9	3.73
	1	3.55	8.60E9	15.26	8.70E9	15.43
	2	3.55	1.22E10	21.64	1.16E10	20.58
	3	3.55	1.35E10	23.95	1.33E10	23.6
FP Multiply	-	7.19	9.85E9	35.43	8.61E9	30.97
FP Divide	-	22.6	1.43E10	161.58	1.45E10	163.84
FP SQRT	-	52.31	8.06E9	210.82	8.06E9	210.82

hits to the total available hits. The total available hits are all the sleep periods longer than the breakeven point. This analysis is useful in showing future enhancement potential for

both techniques. Results in Table 4.12 shows relatively high penetration ratio for both techniques for all the functional units except the integer ALUs. However, the penetration ratio of the DSSG is mostly lower than the SSSG, this is mainly due to the cautious nature of the DSSG. The DSSG favors less penetration over more mistakes, which considerably increased the return on investment of the DSSG as shown earlier in Table 4.9

Finally, it should be noted that the results and breakeven points have been computed at room temperature ($25°$C). However, in reality, high performance functional units that have high activity are likely to have higher temperatures (more subthreshold leakage) compared to other functional units with less activity (less subthreshold leakage) [30]. That would translate to a drop in the breakeven point for high activity units.

Table 4.12: Hit penetration.

Functional Unit	#	DSSG Penetration (%)	SSSG Penetration (%)
Integer ALU	1	1.62	13.2
	2	15.57	32.39
	3	41.8	56.72
Integer Multiply	-	60.33	71.9
Integer Divide	-	15.66	47.85
Floating-Point Adder	0	16.67	53.03
	1	62.12	79
	2	98.79	97.29
	3	65.14	84.84
Floating-Point Multiply	-	30.8	67.91
Floating-Point Divide	-	76.4	84.13
Floating-Point SQRT	-	94.25	94.25

4.4 Summary

So far, the results of running the DSSG on a Superscalar processor shows that the proposed technique is a simple yet effective method for the accurate generation of the sleep signal. The DSSG is capable of adjusting its behavior to match the running application requirements. This is opposed to the static counter based techniques that generate the sleep signal at a fixed interval after the functional unit idles. In effect, the DSSG extends the static techniques by dynamically changing the generation interval to match the

requirements of the applications.

The DSSG exhibits accuracy ranging from 60 - 80%, while consuming approximately $266\mu W$ of power at $2GHz$. In addition to the low power consumption introduced by the DSSG at $2GHz$, the DSSG is capable of operating at much higher frequencies by using a clock divider to interface the DSSGs global counter. This in effect, extends the range of applicability of the DSSG to future processor implementations. Finally, the ability of the DSSG to capture execution profiles of the application allows it to bridge the gap between the architecture and the low level leakage management circuit techniques.

Chapter 5

Application of Sleep Signal Generation on SMT Processors

In SMT architecture, the processor executes several threads simultaneously in the same pipeline. Therefore, for SMT cores, the hardware multithreading is combined with superscalar technology by issuing several instructions from different threads per cycle, increasing the throughput and making better use of the pipeline resources [33]. The primary goal of SMT architecture is to amortize the cost of the microarchitectural structures, such as branch predictors, execution units and reorder buffers, over a higher number of Instructions Per Cycle (IPC), extracted from multiple threads.

Although this concept leads to a better usage of the processor resources, the power consumption is substantially increased, a major issue in SMT and multicore chips. Since the SMT architecture utilization of the functional unit is different than the Superscalar architecture it is interesting to determine the impact of the SSSG and the DSSG on an SMT processor.

To fully quantify the impact of the SSSG and the DSSG on an SMT processor, several key parameters similar to the Superscalar implementation need to be investigated. First, the accuracy of both generators should be investigated taking into account the impact of the generators on the value of the breakeven point. Accordingly, the design and the power consumption of the DSSG and the SSSG when attached to the SMT processor is the second issue that needs to be analyzed. Thirdly, their performances under various workloads must be evaluated.

In addition to the results of applying the SSSG and DSSG to a certain SMT processor architecture, it is instructive to study the impact of the processor architecture on the performance of the SSSG and DSSG.

In the following section, the SMT experimental setup is detailed with emphasis on selection of the workload benchmarks. Section 5.2 presents the results of applying the SSSG and DSSG on an SMT processor. This is followed in Section 5.3 by a detailed study of the impact of the processor architectural parameters on the performance of the sleep signal generators.

5.1 SMT Experimental Setup

To conduct the experiments to establish the idle and active periods of the functional units, a modified version of the cycle-accurate SMTSIM simulator [84] is chosen. It faithfully implements a simultaneous multithreaded processor [35]. Tables 5.1 and 5.2 list the typical parameters of an SMT high performance processor. The simulator is modified to generate the required statistics.

For benchmarks, two groups of multiprogrammed SPEC2000 workloads are selected. The first group consists of several combinations of four benchmarks, and the second group consists of combinations of two benchmarks. Each combinations runs for 5 billion instructions, and all the benchmarks are compiled to the Alpha binary with the default compiler settings of the suite.

The number of benchmarks in the workload groups is chosen since they are representative of the traditional workloads of current desktops. To form these groups of heterogeneous workloads, similar combinations of memory-oriented and processor-oriented workloads, as in [85–87] are used. Table 5.3 summarizes the workload combinations. Each combination has a different proportion of memory-oriented workloads and processor-oriented workloads. For instance, most of the workloads in `apsi-art equake-ammp` and `applu-ammp` are memory-oriented; that is, they tend to access the memory system more often than the functional units. This is due to either a large working set, or a large portion of I/O. However, such combinations as `crafty-vortex wupwise-mesa` and `fma3d-mesa` are mostly processor-oriented. Therefore, they are expected to access the functional units more frequently. With these various combinations and their different characteristics, the efficiency of the SSSG and DSSG with different access patterns is tested.

Table 5.1: SMT test processor architecture.

Parameter	Value
Fetch Queue	32 entries
Branch Predictor	comb of bimodal and 2-level gshare; bimodal/gshar level 1/2 entries- 2048, 1024 (hist. 11), 4096 (global), resp.; combining pred. entries - 1024; RAS entries - 32 ; BTB - 512 sets, 4-way
Branch misprediction latency	10 cycles
Decode and Issue Width	8 instructions
Reorder Buffer	128
Load/Store Queue Entries	64
Instruction TLB*	256 entry 4-way, 8K pages, 30 cycles
Data TLB*	512 entry 4-way, 8k pages, 30 cycles
Memory Latency	150 cycles
L1 I-Cache*	64KB, 4-way, 64B line, 2 cycles
L1 D-Cache*	64KB, 4-way, 64B line, 2 cycles
L2 Unified	2 MB, 8-way, 128B line, 12 cycles

Table 5.2: Functional units in the SMT test processor.

Functional Unit	Operational Latency (cycles)	Issue Latency (cycles)	Available Units
Integer ALU	1	1	3
Integer Multiplier	3	1	1
Floating-Point ALU	2	1	2
Floating-Point Multiplier	4	1	1

5.2 Predictive Sleep Signal Generation on SMT processors

In order to quantify the performance of predictive sleep signal generation on the SMT processor the following sections present the power consumption, accuracy and leakage saving potential of the sleep signal generators. In addition, the dependence of the performance on the workloads is studied.

Table 5.3: SMT simulation workloads.

Mix (4 threads)	Mixture (2 threads)
apsi art equake mesa	applu ammp
ammp mesa swim vortex	crafty vortex
apsi crafty mcf mesa	applu vortex
ammp crafty vortex wupwise	mcf twolf
crafty vortex wupwise mesa	fma3d mesa
apsi art equake ammp	art crafty

5.2.1 DSSG and SSSG Circuit Implementation and Power Consumption for SMT processors

In order to design the DSSG, the SMT processor architecture, shown in Table 5.1 is chosen. The execution core of this processor contains the functional units in Table 5.2. The SSSG is designed with the same processor architecture and execution core. This ensures the fairness of the comparison between both techniques with respect to the associated power consumption overhead.

The following section describes the design of both the DSSG and the SSSG. Section 5.2.1.2 compares the DSSG and the SSSG with respect to their power consumption overhead. The SMT tests use the functional units emulation structures presented earlier in Section 4.3.2.2.

5.2.1.1 DSSG and SSSG Circuits

In order to test the impact of both generators the experiments utilize the 90nm CMOS designs of the SSSG and the DSSG described in Section 4.3.2. In addition to the emulation circuits described in Section 4.3.2.2. In order to explore the impact of the sleep signal generators at higher frequencies, the designs are set to run at 2 and 4GHz.

To quantify the impact of the DSSG and the SSSG on the sleep signal generation process, the breakeven point of the functional units in Table 5.2 must be identified. However, due to the higher operating frequency set for the SMT tests only the emulation testbenches that depend on FO4 inverter chains to emulate the functional units is used.

5.2.1.2 Power Consumption Overhead Comparison

To compare the power consumption overhead of the DSSG and the SSSG, the baseline in Table 5.4 is compiled by summarizing the dynamic power consumption of both generators, extracted from SPICE simulations. The table takes into account an extra bit for the global counter in order to count up to 512 cycles to cover the higher breakeven points associated with the higher frequencies.

Table 5.4: Power consumption of the sleep signal generators.

Frequency	SSSG (μW)	DSSG (μW)	Global Counter (μW)
2GHz	104	266	277
4GHz	208	531	554

Table 5.5: Breakeven Point of Testbench Functional Units without Sleep Signal Generators' Overhead.

Functional Unit	Testbench	Breakeven Point (cycles)	
		2GHz	4GHz
Integer ALU	FO4	136	271
Integer Multiplier		134	269
Floating-Point Adder		178	357
Floating-Point Multiplier		167	334

Table 5.5 presents the breakeven points of the various functional units without adding the overhead of the sleep signal generators. However, more important is the impact of the two sleep signal generators on the breakeven points in Table 5.6. Additionally, the table shows the increase in the breakeven point values for both generators and their respective global counters. The breakeven point is calculated by assuming that there are four functional units to control[1], where the overhead of each sleep signal generator is calculated using the overhead equation (4.4). Table 5.6 indicates that the breakeven points can increase between 12% to 34% by applying the DSSG instead of the SSSG. However, again this does not allow for a complete comparison between both generators, since it ignores the accuracy of each generator in predicting the standby period length. The following section analyzes the operation of both sleep signal generators under the benchmark workloads in Table 5.3 to identify their respective accuracies.

[1]The integer ALUs are heavily utilized, rarely remaining in standby beyond 50 cycles. As a result, no sleep signal generation circuit is attached to them.

Table 5.6: Breakeven Point of Testbench Functional Units with Sleep Signal Generators' Overhead.

Functional Unit	Frequency	Breakeven Point (cycles)		Percentage Increase
		SSSG	DSSG	
Integer ALU	2GHz	210	278	32.71
Integer Multiplier		159	182	14.26
Floating-Point Adder		227	273	19.95
Floating-Point Multiplier		191	214	11.69
Integer ALU	4GHz	395	531	34.86
Integer Multiplier		309	355	14.66
Floating-Point Adder		438	529	20.74
Floating-Point Multiplier		374	419	11.97

5.2.2 Accuracy of the Sleep Signal Generators on SMT Processors

By adopting the traces extracted from the modified SMTSIM, the C++ implementation for both the DSSG and the SSSG introduced earlier in Section 4.3.3 is used. For both techniques, to accommodate the higher operating frequency, the breakeven point is changed from 10 to 500 cycles. These values correspond to the breakeven points, extracted earlier from the circuit simulations of both the DSSG and the SSSG in Section 5.2.1.2. In addition, these values explore the design space for all the functional units in the test processor in Table 5.2. On the other hand, for the SSSG, the threshold is varied between 10 and 80 cycles in steps of 10 cycles.

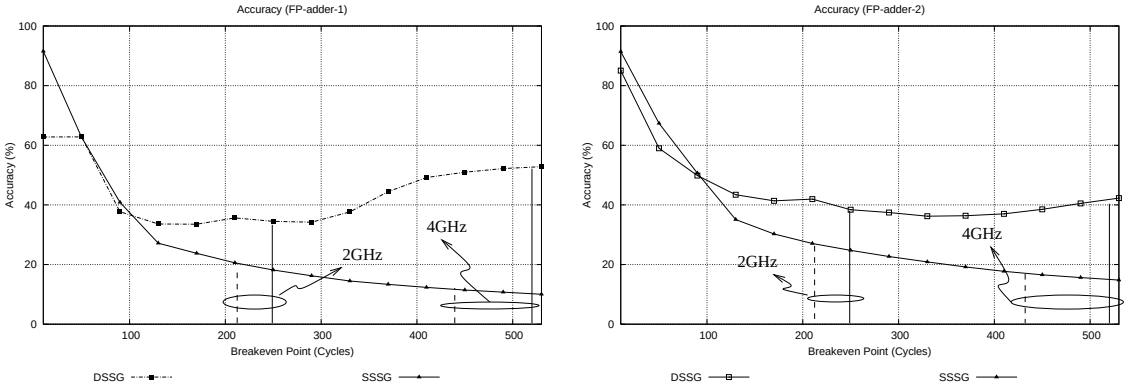
Using the accuracy metric set in Section 4.3.3 the results of the DSSG and the SSSG, regarding their capability to predict the length of the standby period can be studied. This accuracy metric is defined as the percentage of correct decisions to the total number of decisions, made by the sleep signal generator. To compare the SSSG and the DSSG techniques, the results for both techniques for the integer and floating point ALUs and multiplier units will be introduced in the following sections.

5.2.2.1 Integer and Floating Point ALU

By referring to Table 5.2, the test processor has three integer and two floating point ALUs. In order to test the integer ALUs, the modified SMTSIM simulator is instructed to keep track of all the instructions that utilize any of the three integer ALUs in the test

processor core. This creates a standby trace that contains the information about all the periods of time during which the ALUs are employed. The evaluation of the accuracy metric in Equation (4.5) indicates that the DSSG and the SSSG have a very low accuracy in tracking the integer ALU units. This is expected since these units are heavily utilized and rarely stay idle long enough to allow for a shutdown.

For the floating point ALU, Fig. 5.1 reveals that the DSSG has a higher accuracy for predicting the standby length, especially at the higher breakeven points. The two vertical lines at 2 and 4GHz show the breakeven values from Table 5.6. Clearly, the small increase in the overhead, due to the application of the DSSG, is counter-balanced by the increased accuracy.



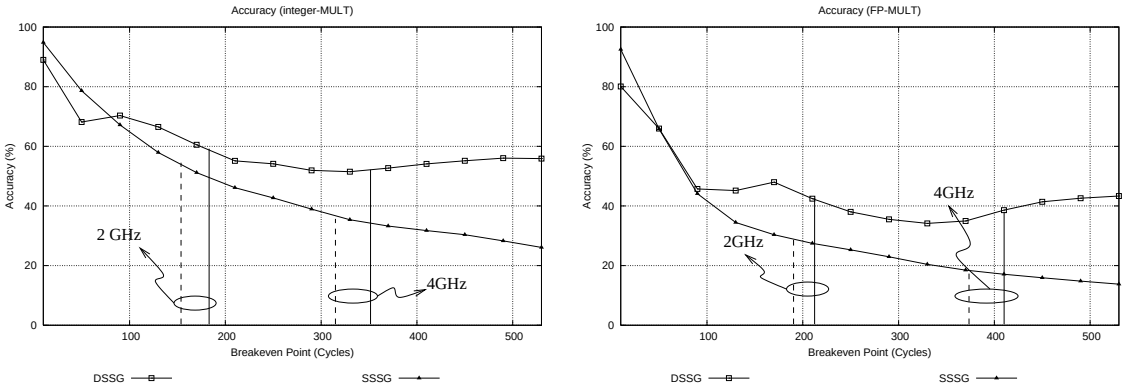
(a) (b)

Figure 5.1: DSSG vs. SSSG prediction accuracy for the floating point ALU (a) unit 1 and (b) unit 2.

5.2.2.2 Integer and Floating Point Multipliers

By comparing the results for the integer and floating multipliers in Fig. 5.2, it is clear that the DSSG outperforms the SSSG in accurately predicting the sleep length for the higher breakeven points. And similar to Fig. 5.1, the vertical lines are the actual breakeven points from Table 5.6.

In addition to the superior average accuracy of the DSSG over the SSSG, Fig. 5.3 draws attention to a significant aspect of the decision to use either of the sleep signal generators. This figure depicts a set of histograms for the different benchmark combinations run on the SMTSIM simulator. The x-axis represents the sleep length, and the y-axis is the number of sleep instances with the corresponding sleep length. The most important aspect in Fig. 5.3 is the dynamic nature of the sleep traces. A single static value



(a) (b)

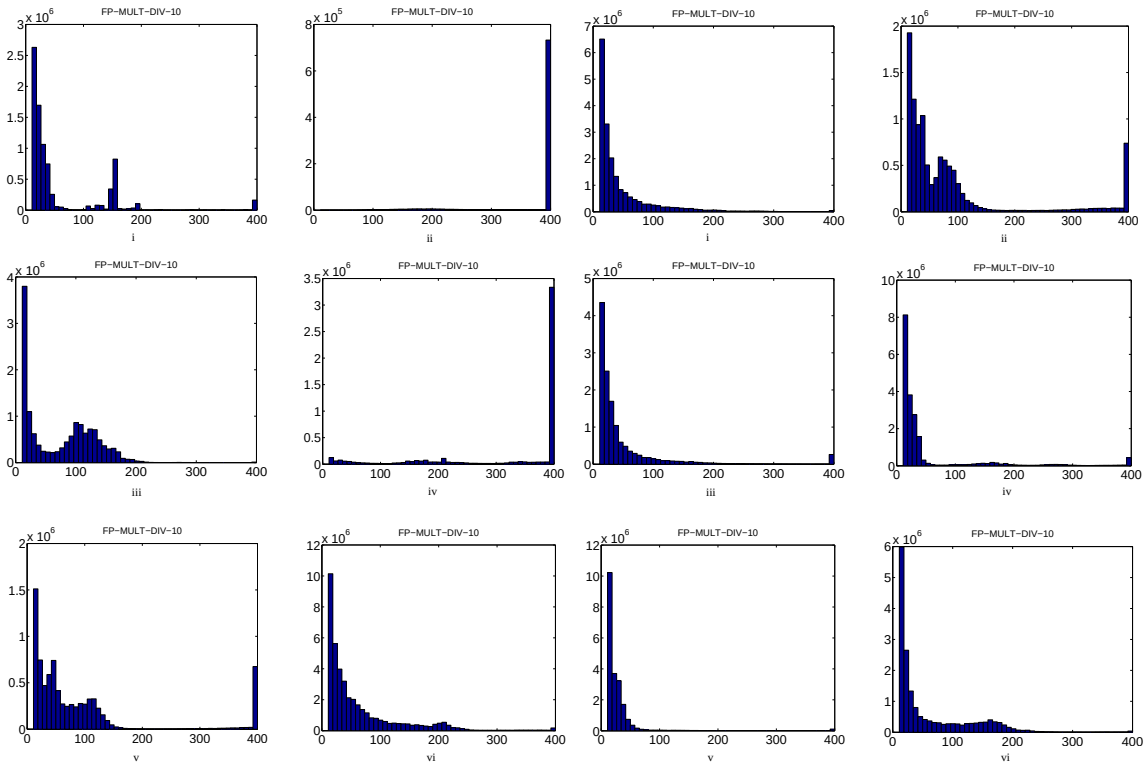
Figure 5.2: DSSG vs. SSSG prediction accuracy for the (a) integer multiplier unit and (b) floating point multiplier unit.

for the sleep signal generation threshold consistently underperforms that of the dynamic approach that matches the threshold to the current application. For example, setting the threshold to 80 cycles and assuming a 200 cycle breakeven, the SSSG will always trigger a shutdown, and will almost always miss since many sleep periods reach the 80 cycles mark but never reach the 280 cycles required to achieve any savings. Setting the fixed threshold lower than 80 cycles will reduce the required 280 cycles but will substantially increase the misses, raising it will eliminate any saving opportunities. On the other hand, with the same 200 cycle breakeven, the DSSG will either increase the threshold to prevent this erroneous shutdowns when the multiplier is used similar to Fig. 5.3(b).v where there is no sleep period that reaches the 200 cycles. Alternatively, the DSSG can reduce the threshold to zero to maximize the savings when the multiplier is used similar to Fig. 5.3(a).ii where all the sleep periods reach 200 cycles. This ability of the DSSG to change the threshold based on the application profile is its greatest advantage.

5.2.3 The Sleep Signal Generators Workload Dependence on SMT Processors

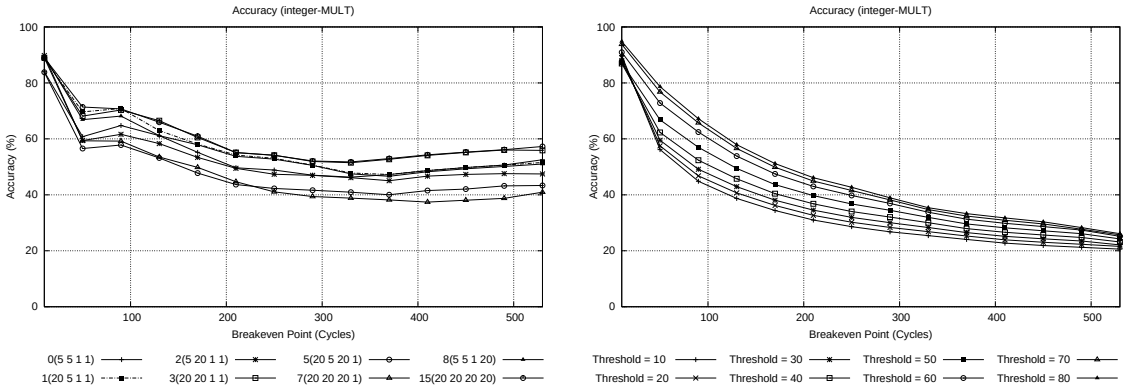
In this section, the effect of the workload characteristics on the accuracy of DSSG and SSSG is explained.

Figures 5.4(a) and 5.4(b) depict the accuracy of the DSSG and SSSG for one of the two integer multiplier units. These figures show the average of all the workloads, and reveal several things. First, the DSSG has a consistently higher accuracy than the SSSG, especially at the high breakeven points. Secondly, from the breakeven point of



(a) 2 threads (b) 4 threads

Figure 5.3: Floating point multiplier unit standby trace histogram. The x-axis represents the sleep length, and the y-axis is the number of sleep instances with the corresponding sleep length.

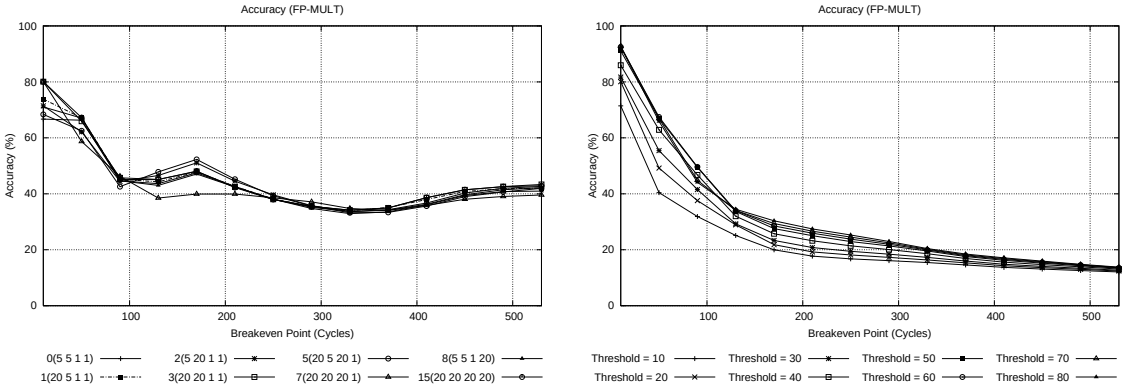


(a) (b)

Figure 5.4: (a) DSSG and (b) SSSG prediction accuracy for the integer multiplier unit.

100, the SSSG accuracy decays. However, the DSSG behavior can be divided into two intervals. From the breakeven points of 100 to 300 DSSG the accuracy decreases, but at a rate that is slower than that of the SSSG in the same interval. The DSSG saturates at the

breakeven points above 300. Recall that in Table 5.6, higher frequency processors tend to have higher breakeven points. This implies that for high performance systems and higher clock frequencies, the DSSG is the better choice At the breakeven points below 100, the behaviors of the DSSG and SSSG differ. The SSSG accuracy decreases consistently, whereas the accuracy of the DSSG alternates between saturation and decline.



(a) (b)

Figure 5.5: (a) DSSG and (b) SSSG prediction accuracy for the floating point multiplier Unit.

The situation is different for the floating point multiply unit in Figures 5.5(a) and 5.5(b). The utilization of this unit is lower than that of the integer counterpart. The accuracy of the DSSG does not only saturate, but also slightly increases at the high breakeven points. The SSSG accuracy does not increase at the high breakeven points, but the decay does slow down.

Due to the very high utilization of the integer ALU units, the accuracy of both the DSSG and SSSG is low, and saturates for the breakeven points above 200. But still the accuracy of the DSSG is higher than that of the SSSG.

The workloads to test the DSSG and the SSSG are categorized into processor-bound workloads, memory-bound workloads and neutral workloads. Since Processor-bound workloads are expected to have high utilization of execution units, the prediction accuracy is expected to be low. Alternatively, since Memory-bound workloads access the memory system more often, the utilization of the execution units will be lower and the prediction accuracy higher. The neutral workloads fall somewhere in between as displayed in Figure 5.6. The other execution units exhibit similar behavior. The SSSG exhibits similar behavior but with a lower accuracy. It is important to note here that the behavior of the neutral workload differs with the execution unit and the number of threads. So sometimes the neutral has better accuracies than the memory and sometimes

it is worse.

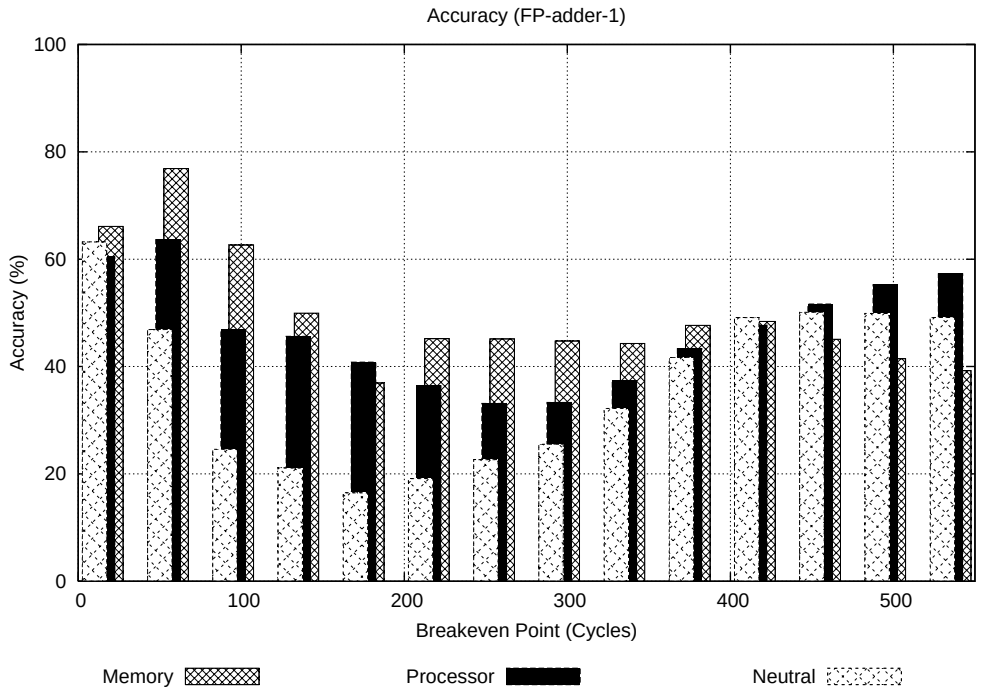


Figure 5.6: DSSG prediction accuracy for floating point add/sub. unit running processor/memory/neutral-bound workloads.

Another factor that affects the utilization of the execution units is the number of threads in each workload. Figure 5.7 portrays the DSSG accuracy for a floating point adder/subtracter with two-thread and four-thread workloads. As it is clear from the figure, the accuracy for the two-thread workload is higher due to the lower utilization compared to that of the four-thread workload.

It can be concluded that as the functional unit utilization increases, it becomes more difficult to make a prediction. However, the accuracy of the DSSG remains higher than that of the SSSG, especially at the high breakeven points.

5.2.4 SMT Leakage Savings Potential for Sleep Signal Generators

To fully characterize the performance of the DSSG and the SSSG on an SMT processor, their leakage savings for the microprocessor's functional units in Table 5.1 needs to be identified. Similar to Section 4.3.5 the number of cycles the functional units are idle after the breakeven point is used to represent the leakage savings. The number of cycles after the breakeven is the real performance metric, since the accurate prediction

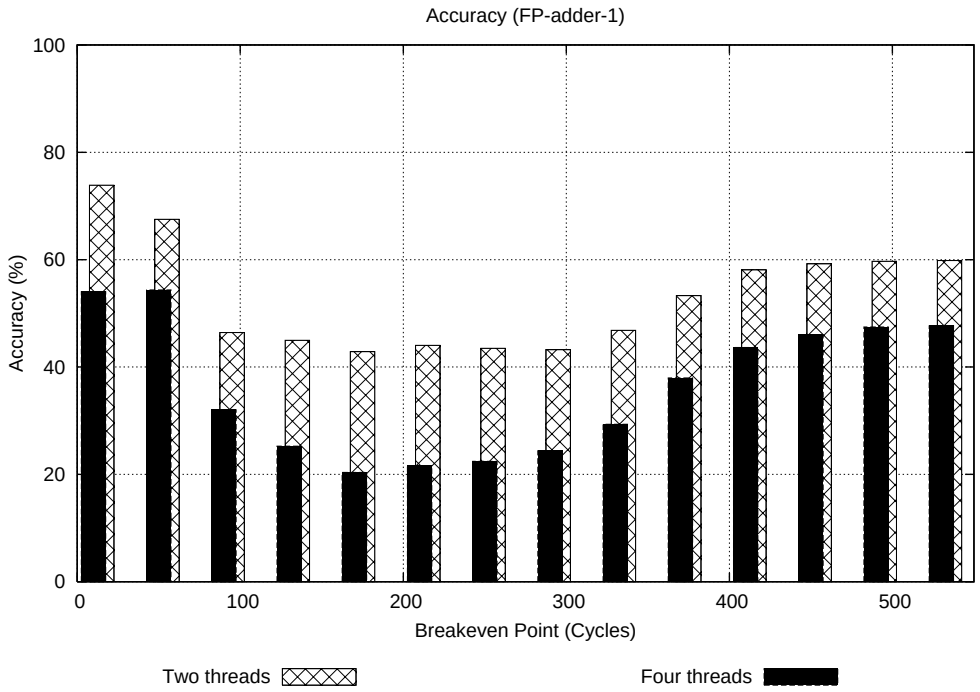


Figure 5.7: DSSG prediction accuracy for floating point add/sub. unit running two and four thread workloads.

to determine if the functional units reach the breakeven point guarantees only that the processor is not losing power. It is the counting of the cycles beyond the breakeven that will determine the actual leakage savings. However, the number of cycles, where there is a pre-breakeven wake-up, must be deducted from the savings, since these cycles represent wasted power. Accordingly, the net savings of either sleep signal generators can be calculated by evaluating equation (4.8).

Equation (4.8) is used in conjunction with the SMT traces and the breakeven points in Table 5.6 to calculate the savings in Table 5.7. Table 5.7 also provides the difference in savings between the DSSG and the SSSG. Again, the SMT traces are used to simulate all the benchmarks running in series and due to the limited history dependence of both the DSSG and the SSSG, the trends in Table 5.7 are expected to remain independent of the order in which the benchmarks are run.

From the results in Table 5.7, it is obvious that both the DSSG and the SSSG cannot be used to power gate integer ALUs, since they consume more power during the misses than the total power savings. However, it is noteworthy that the absolute value of the power losses for the DSSG, highlighted by the gray cells in Table 5.7, are between 1 and 7 orders of magnitude lower than those of the SSSG. This is an incentive for future modifications to the DSSG to better predict the ALUs performance, creating power gating

Table 5.7: Leakage savings potential for SSSG and DSSG on SMT processors.

Functional Unit	#	Frequency	Net Savings (Cycles)		Percentage
			SSSG	DSSG	Difference
Integer ALU	1	2GHz	-1.56E10	-5.54E6	-
Integer ALU	2		-1.38E10	-1.16E9	-
Integer Multiplier			4.46E10	4.46E10	-0.14
Floating-Point Adder	1		3.01E10	3.46E10	15.17
Floating-Point Adder	2		1.82E10	2.26E10	23.88
Floating-Point Multiplier			1.64E10	1.92E10	16.77
Integer ALU	1	4GHz	-3.61E12	-8.85E5	-
Integer ALU	2		-4.96E10	-1.26E6	-
Integer Multiplier			3.96E10	4.18E10	5.57
Floating-Point Adder	1		2.05E10	3.23E10	57.03
Floating-Point Adder	2		8.66E9	2.13E10	146.32
Floating-Point Multiplier			7.67E9	1.77E10	131.44

opportunities.

The DSSG and the SSSG are equivalent for the integer ALU-1 at 2GHz. In this case, the SSSG is better suited for the task due to its simplicity. This trend is broken for the other functional units at 2GHz, where the DSSG outperforms the SSSG by 15% to 23%. At 4GHz, the DSSG consistently outperforms the SSSG for all the functional units. This is attributed to the higher breakeven points, associated with the shorter clock period, and the relatively lower accuracy of the SSSG at these breakeven points.

Another significant aspect in quantifying the leakage savings of the two signal generators is the absolute value of the energy savings. However, determining this value depends substantially on the values of the leakage savings per cycles. By using the emulation circuits of FO4 introduced in Section 4.3.2.2, the values in Table 5.8 are determined. They correspond to the subtraction of the leakage power, when the functional unit is not power gated, from that of a power gated unit. Also, Table 5.8 lists the leakage savings per cycle, multiplied by the net savings in Table 5.7 and the clock period to exemplify the achievable leakage energy savings.

Table 5.8: Energy savings for SSSG and DSSG on SMT processors.

Functional Unit	#	Frequency	Leakage Savings (μW)	Total Energy Savings (μJ)		Percentage Difference
				SSSG	DSSG	
Integer Multiplier		2GHz	7.10	158.42	158.42	0
Floating-Point Adder	1		3.55	53.40	61.38	14.95
Floating-Point Adder	2		3.55	32.29	40.09	24.18
Floating-Point Multiplier			7.19	58.99	69.06	17.07
Integer Multiplier		4GHz	7.10	70.33	74.24	5.56
Floating-Point Adder	1		3.55	18.18	28.65	57.56
Floating-Point Adder	2		3.55	7.68	18.89	145.96
Floating-Point Multiplier			7.19	13.79	31.83	130.77

5.3 Architectural Dependence of Predictive Sleep Signal Generation

After studying the performance of the sleep signal generators on a model processor it is instructive to investigate the changes in the performance of the sleep signal generators for various modifications of the processor architecture. In order to ensure the fairness of the comparison, a base processor modeled similar the processor in Table 5.1 is used. The base processor is modified and the SPEC benchmarks introduced earlier in Table 5.3 are rerun on the modified processor.

Table 5.9: Architectural modifications to the base processor

Architectural Unit	Base Value	Modified Value
Floating Point ALU	2	4
Floating Point Multiply	1	2
Integer Multiply	1	2
Load Store Queue	64	32 - 128
Memory Latency	150	100 - 300

Table 5.9 summarizes the modifications introduced to the processor. These modifications are applied to the processor separately in order to identify the impact of each modification. The modifications in the execution stage listed in Table 5.9 are limited to the subset of functional units that experienced performance gain using either sleep signal

generators.

5.3.1 Floating Point ALU

In order to examine the performance of the DSSG relative to the number of the floating point ALUs available to the processor, two additional floating point units are added to the base architecture. Fig. 5.8 presents the histograms of the sleep periods of each FP-ALU in the processor. The x-axis represents the sleep length, and the y-axis is the number of sleep instances with the corresponding sleep length. Comparing the histograms of ALU-1 and ALU-2 in Fig. 5.8 shows that ALU-2 has a flatter distribution of the sleep periods corresponding to the lower utilization of this ALU. Accordingly, the DSSG has better performance tracking ALU-2 compared to ALU-1 as shown in Fig. 5.9. Fig. 5.9 shows the leakage savings ratio between the DSSG and the SSSG. The ratio of the DSSG to the SSSG for the ALU-3 and 4 is lower compared to ALU-1 and 2. This is attributed to the fact that the number of long sleep periods for ALU-3 and 4 is more compared to ALU-1 and 2. The higher number of long sleep periods allow the SSSG to compensate for its mistakes which can go up to 5 to 10 times more than that of the DSSG. However, this does not account for the disruption of the processor due to the erroneous shutdowns performed by the SSSG.

Comparing the results of the ALU-1 and 2 between the base architecture and the modified version shows little change with respect to the leakage savings achieved by the DSSG. This is due to the fact that the added ALUs slightly affected the number of sleep periods in the region between 50 and 100 cycles in Fig. 5.8 while preserving the overall trends. The change in the lower regions of the histograms results in no change in the predictive behavior of the DSSG since it ignores these periods when targeting the higher breakeven points.

5.3.2 Floating Point Multiply

Increasing the number of floating point multiply units is limited to an extra one functional units due to the low utilization of this functional unit compared to the ALUs and the limitation on the simultaneous requests for multiply instructions. Comparing the results of the base architecture to the modified architecture reveal slight variation in the DSSG performance as compared to the SSSG (Fig. 5.10). This is attributed to the comparable histogram of both architectures. Although the second multiplier increases the performance by allowing multiple multiply instructions to execute simultaneously, it has

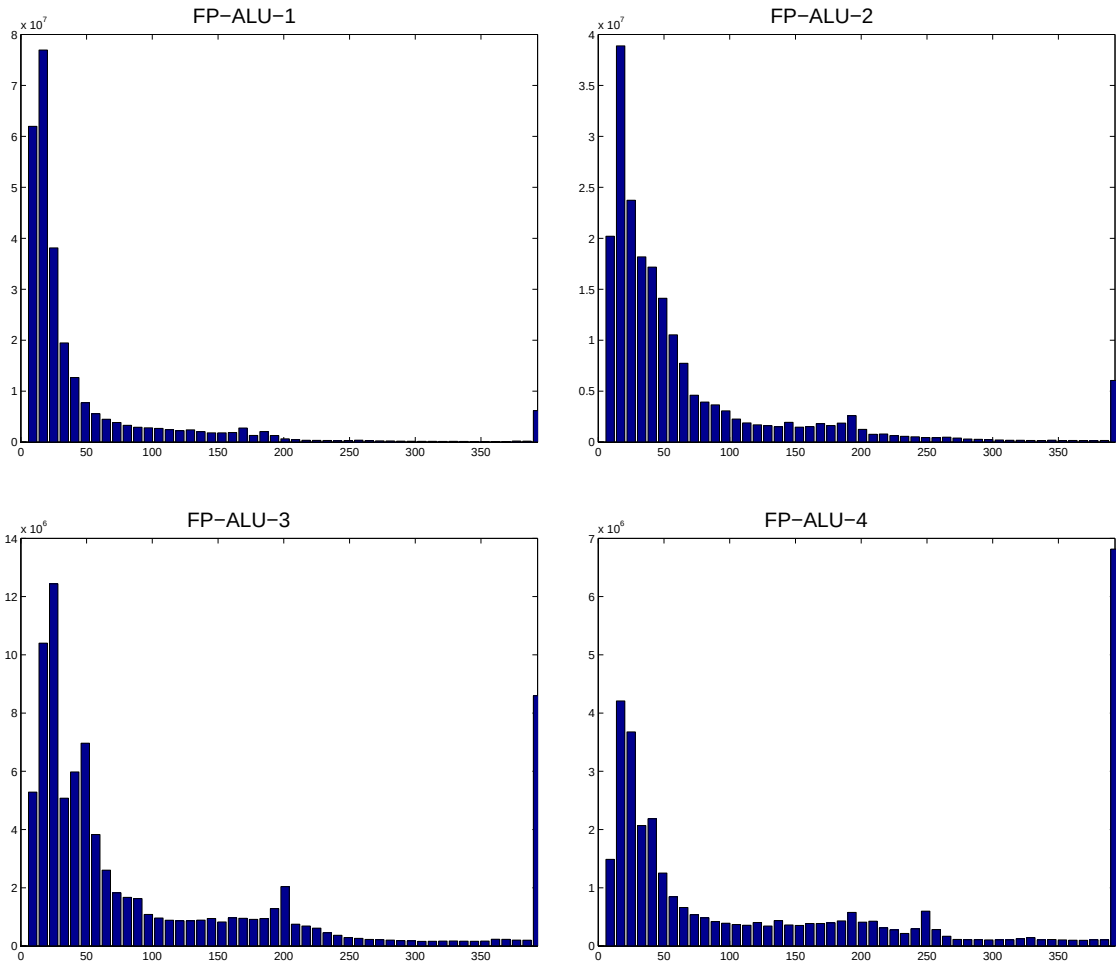


Figure 5.8: Histograms of the execution traces on the FP-ALUs.

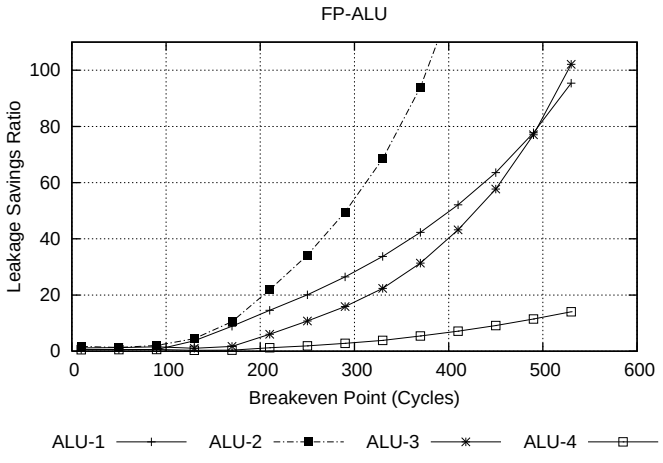


Figure 5.9: Leakage Savings for FP-ALU.

little effect on the first multiplier sleeping patterns. This can be explained if we take for example two pending multiply instructions. These instructions will run consecutively on a single multiply unit and then the unit will go into a standby period depending on code behavior. In a two multiplier architecture the instructions will be split between both. However, this split has little impact on the standby period as it is mainly controlled by the succession of requests from the executing code.

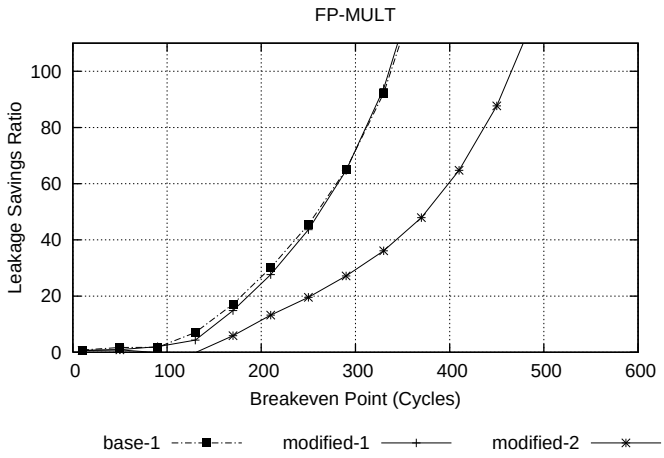


Figure 5.10: Leakage Savings for FP-MULT.

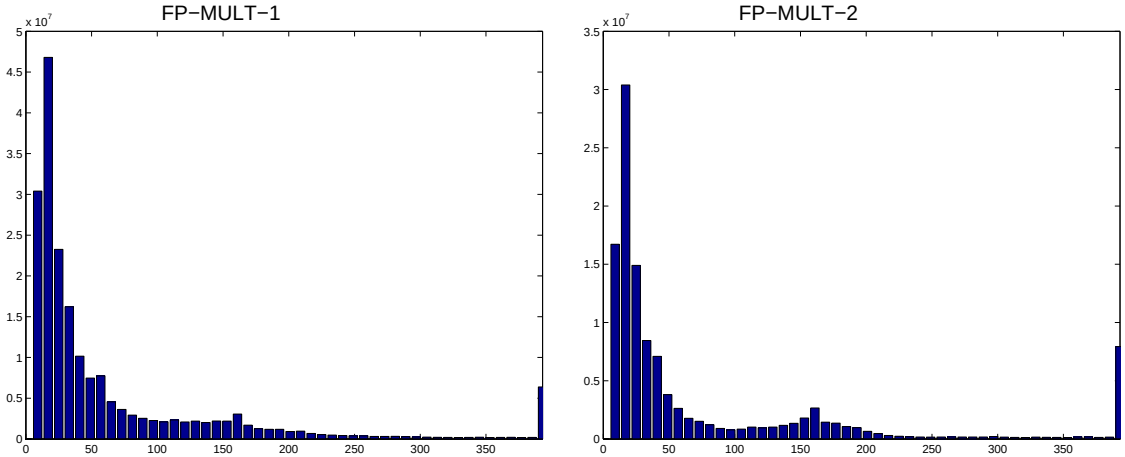


Figure 5.11: Histograms of the execution traces on the FP multiply.

Comparing the results of the first and second multipliers (modified-2) shows that the histograms in Fig. 5.11 has an increased number of long sleep periods compared to the first multiplier. The increase in long sleep periods allows the SSSG to recover some of its losses.

5.3.3 Integer Multiply

The integer multiply unit was augmented by a second multiplier. In the case of the integer multiplier the savings achieved by the DSSG as compared to the SSSG is relatively lower compared to the savings achieved for the floating point ALU and multipliers (Fig. 5.12(a)). This is mainly attributed to the heavy presence of long standby periods in the case of the integer multiply (Fig. 5.13). However, the savings achieved by the DSSG and the corresponding shutdown periods are in the order of $4 \sim 5 \times 10^{10}$ cycles (Fig. 5.12(b)). This means that the DSSG was capable of successfully shutting down the multipliers for approximately $72 \sim 90\%$ of the total simulation time in cycles.

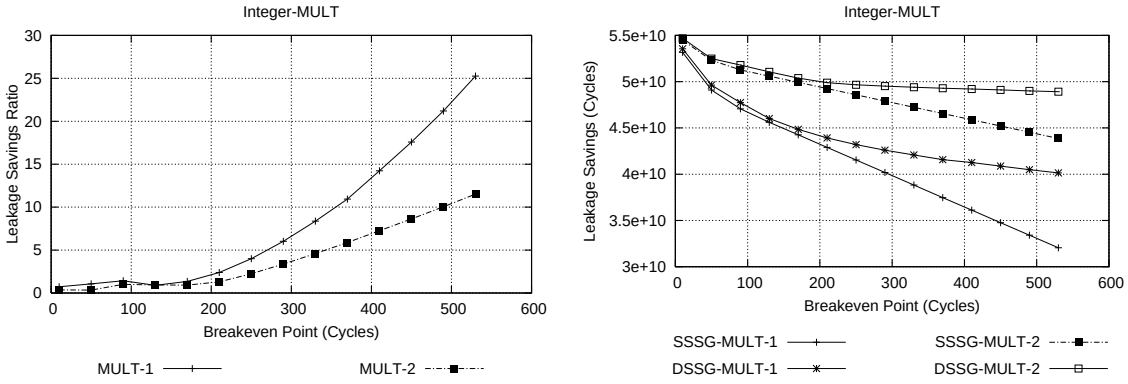


Figure 5.12: a) Ratio of savings between the DSSG and the SSSG. b) Leakage savings represented in number of cycles.

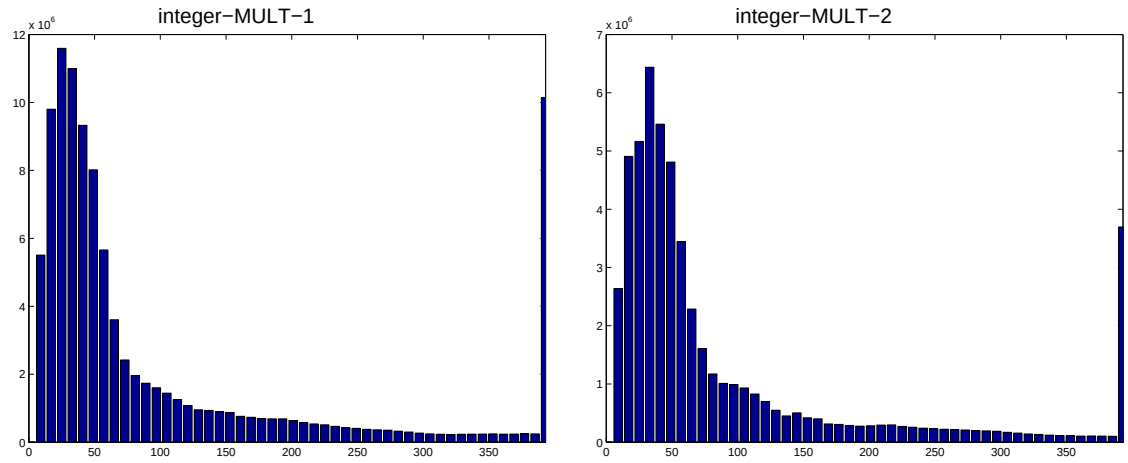


Figure 5.13: Histograms of the execution traces on the integer multiply.

5.3.4 Memory Latency

An interesting aspect of the processor design is the memory latency. The memory latency was set to 100, 150 and 300 cycles. Although the actual memory latency is gradually increasing the use of ”latency hiding” techniques, such as prefetching and the trend to move more parts of the memory on-chip results in an effective reduction of the memory latency as seen from the perspective of the functional units. Tracking the behavior of the DSSG versus the SSSG shows that the DSSG performance increases with the reduction of the effective memory latency as seen by the functional units. Fig. 5.14 shows the behavior of the DSSG targeting the floating point ALUs. The lower memory latencies causes more sleep periods to occur at the shorter sleep lengthes. This shift in the length of sleep periods when the memory latency is moved from 300 to 100 is depicted in Fig. 5.15.

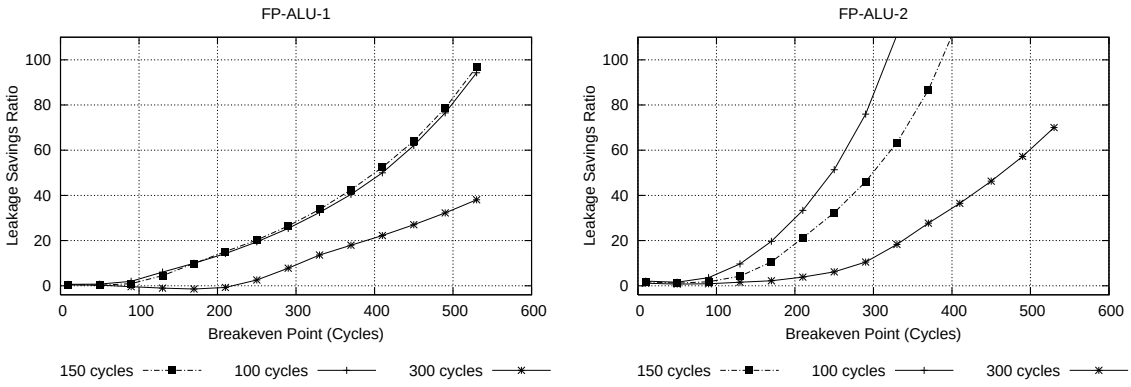


Figure 5.14: DSSG vs. SSSG with the change of memory latency (floating point ALUs)

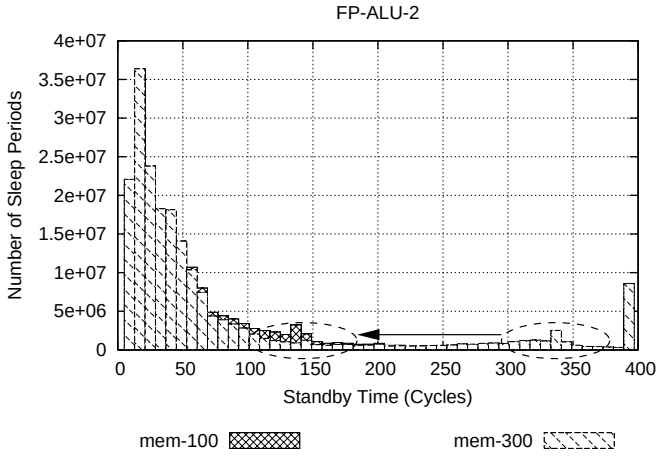


Figure 5.15: Histograms of the execution traces with memory latency variation.

5.3.5 Load Store Queue (LSQ)

The LSQ size was changed between 32, 64 and 128 entries. The relevance of the size of the LSQ is most pronounced when there is burst of load/store instructions. Accordingly, it is expected that the functional units with the highest utilization and corresponding load/store requests will be affected since the time to satisfy the request will increase. The histograms in Fig. 5.16 show such trend. The smaller LSQ size caused the heavily utilized ALU-1 to stall more frequently especially for shorter sleep lengths.

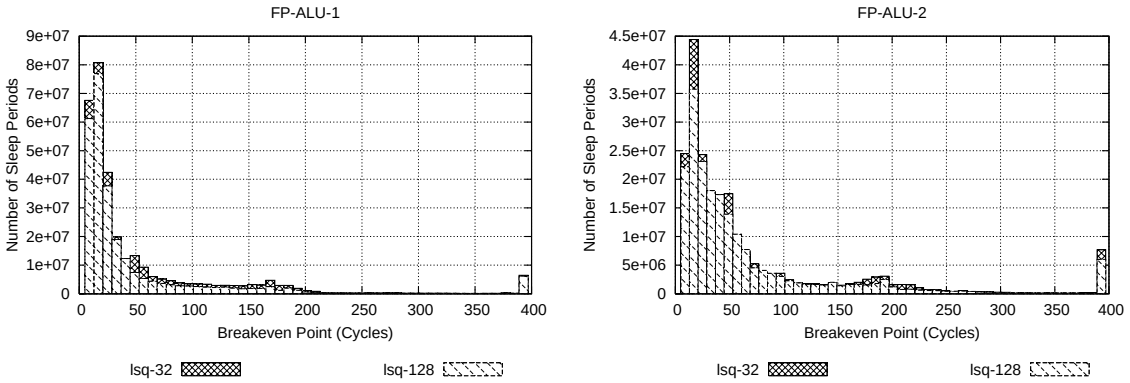


Figure 5.16: Histograms of the execution traces on the floating point ALU.

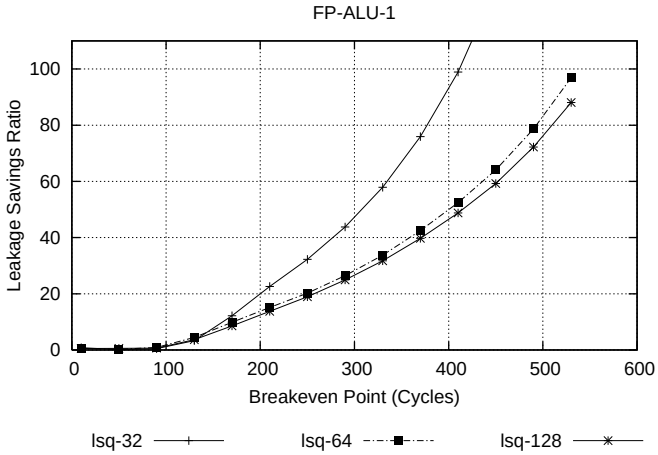


Figure 5.17: Leakage Savings for FP-MULT.

The variation of the sleep length between the LSQ-32 and LSQ-128 resulted in the variation in the DSSG performance shown in Fig. 5.17. The the higher number of smaller sleep periods tends to trick the SSSG in triggering the sleep signal. However, these periods never reach the breakeven point thus favoring a dynamic approach to handle this changes.

5.4 Embedded Processors

Since embedded processors face a more stringent power budget compared to general purpose processors they are severely affected by the increase of leakage power. Accordingly, the importance of sleep signal generation for embedded processors is significant.

Although embedded processor designers have tighter control on the software running on the processor, the sleep profile is not expected to be uniform. Accordingly, based on understanding the performance of both the DSSG and the SSSG on general purpose processors it is expected that embedded processors will also benefit from the use of the DSSG since it is capable of tracking non-uniform sleep profiles.

Finally, it is important to note that embedded processors are usually in-order scalar processors. This in turn will introduce some challenges to the task of power gating since it will require the complete shutdown of the processor front end. Alternatively, the DSSG can be used with DVS techniques to slow down the core instead of a complete shutdown.

5.5 Summary

Reviewing the results of applying the DSSG and the SSSG on several state-of-the-art processors draws attention to the viability of aggressive leakage management of processor functional units. The SSSG is generally superior for small, low frequency functional units albeit at a higher miss rate potentially compromising the processor performance due to unnecessary shutdowns. The DSSG on the contrary maintains a comfortable lead on both the superscalar and SMT architectures. The DSSG has higher accuracies at higher breakeven points which correspond to high frequency operations. However, results show that neither techniques is overly successful in targeting the integer ALUs. The integer ALUs are very heavily utilized preventing successful shutdowns during code execution.

Chapter 6

Multi-Pin Interconnect Power Optimization

Since interconnect wires are slowly dominating the overall chip performance, interconnect delays are becoming more critical to design than gate delays, and the interconnect-related power consumption is determining the total power that is consumed by the chip. Therefore, power awareness is crucial to optimize the interconnects.

In this chapter, the formulation for a power-aware multi-objective interconnect optimization methodology is discussed. The focus is on solving a Power-Efficient multi-pin Integer linear programming based global Routing Technique (*PIRT*).

6.1 Power Driven Routing

Timing driven routing is the focus of most current research on global interconnect design [88–90]. However, due to the tremendous increase in dynamic power consumption in global interconnects, research has also focused on minimizing their dynamic power while striving to minimize the impact on the chip delay.

Research in [12–19] tackled the low power interconnect problem through optimum buffer insertion within the framework of sequential global routers. Research in [14, 15, 91] focuses on only single net optimization. In the meantime, the research in [16–19, 92, 93] focuses on the optimum buffer insertion assuming prerouted nets.

The main limitation facing power optimal sequential global routers is the lack of simultaneous optimization of the various interconnect performance metrics. Sequential routers are based on single net optimizations [14, 15, 91] which inherently implies a net

ordering effect. Net ordering effect means that for successful chip timing closure the nets has to be ordered according to their importance. This ordering limits the ability of the technique to find globally optimum solution. Furthermore, several sequential routers postpone the task of buffer insertion after net routing [16–19, 92, 93] limiting the ability of the technique to prevent congestion in general and especially buffer related congestion where the availability of buffer insertion locations is contested by several nets.

PIRT on the other hand introduces a simultaneous power optimal routing technique that eliminates the dependence on the net ordering while considering buffer insertion in conjunction with routing. The technical contributions for the newly developed technique were presented in Chapter 1 and are summarized as follows:

1. Unlike previous approaches, the newly developed approach is capable of timing optimization, buffer insertion and power reduction simultaneously with routability consideration.

2. The optimization of power consumption and simultaneously accounting for the buffer blockage, which has not been considered in previous analytical formulations of the power optimization problem, is formulated.

3. The optimization of the power consumption without affecting the chip's maximum frequency.

4. The problem is formulated so that it is independent of the delay and the power models used, allowing for more flexibility in applying the new technique to scaled technologies.

5. PIRT is capable of simultaneously routing and power optimizing the chip with runtime less than 0.1 second per net.

The objective of PIRT is to find a globally optimal routing solution. The routing solution involves the simultaneous power optimization and routing of all the nets. The PIRT solution strives to trade-off the slack between the maximum required frequency and the delay of the nets that are faster than the required frequency with the optimization of the interconnect power consumption. In order to present the proposed power optimal multi-objective optimization for global interconnects, the various efforts pertaining to interconnect optimization should be revisited.

In Section 6.2 general background information pertaining to the global routing and buffer insertion problems are presented. This is followed by Section 6.3 discussing preliminary routing techniques and the power and delay models used in the formulation.

Section 6.4, introduces the ILP based routing formulation (PIRT). Lastly, Section 6.5 provides the results of applying the new formulation on industrial benchmarks.

6.2 Introduction

6.2.1 Global Routing: Unified Timing and Congestion Minimization

One of the fundamental goals of global routing is to route all the nets within the circuit without overflow (i.e., congestion minimization). Several works have been proposed in this area such as sequential approaches [94], rip-up-and-reroute techniques [95, 96], multicommodity flow-based techniques [97,98], and hierarchical methods [99]. As technology scales in terms of the device dimensions, the interconnect delay becomes a performance bottleneck. Therefore, minimizing congestion alone is not sufficient. To deal with this trend, several research efforts have been performed on timing-driven global routing [100–105] (i.e., interconnect delays are considered explicitly during global routing).

In [100], a set of routing trees, satisfying timing constraints for each net, have been initially chosen, after which a multicommodity flow method was applied to choose a single routing tree for each net such that the congestion was minimized. In [101], the nets have been individually routed after which the congested area was ripped up and rerouted by a multicommodity flow algorithm. The work in [102] has incorporated timing issues with an iterative deletion technique for standard cell designs. A top-down hierarchical and assignment method that is combined with timing constraints has also been used for FPGA routing [103]. In almost all the aforementioned methods, several wires have required detours (to avoid congestion); therefore, the signal delay might be detrimentally affected (i.e., congestion and delay are often competing objectives). In [104] an approach has been suggested to simultaneously optimize congestion and delay according to a network flow formulation so that the timing slack consumptions were adaptive for the congestion distributions. In [105] the authors have formulated global routing as a multicommodity flow problem, and adopted a shadow price mechanism to incorporate the timing performance and routability into a unified objective function. Although most of these researchers have proposed solutions to offer two important and competing objectives, congestion and delay, none has considered the power minimization of realistic interconnect trees under given timing budgets.

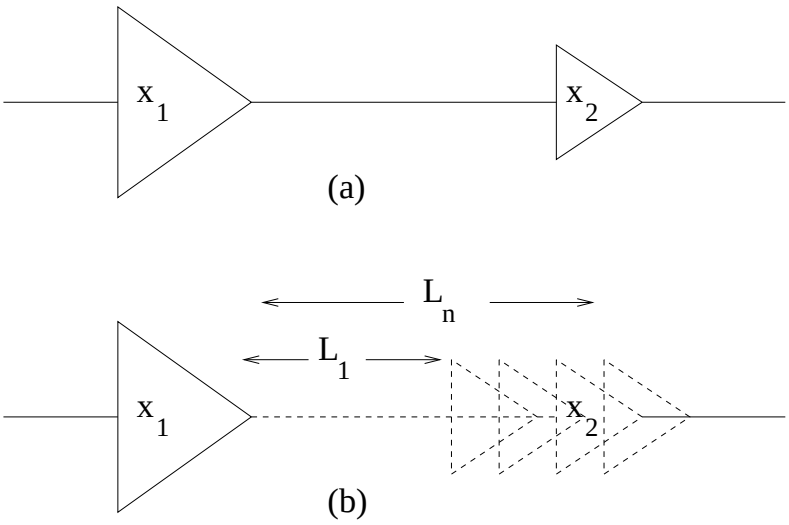


Figure 6.1: Buffer insertion and sizing: (a) the buffer sizing problem finds values of buffer sizes that satisfy delay constraint and (b) the buffer insertion problem finds optimum buffer positions to satisfy delay constraint.

6.2.2 Buffer Insertion-Based Methods

Buffer insertion-based techniques are effective for reducing interconnect delay. Several works have examined delay driven buffer insertion for 2-pin nets [12, 106–109]. Broadly characterizing the interconnect optimization efforts have indicated two principal techniques: analytical and dynamic programming.

Analytical optimization techniques have been applied to find a closed form expression that can minimize one of the major objectives: delay, power and routing topology, with the other objectives kept within bounds. Simple optimization techniques have been employed to minimize the delay of the interconnect by buffer insertion, disregarding all other aspects of the interconnect [11, 110]. However, an important limitation to these formulations of the problem is that it has not taken into account the case when the optimum buffer positions lie on top of the pre-placed functional blocks. This limitation has introduced another set of problems to find the independent buffer sizes ($x_1 \neq x_2 \ldots \neq x_n$) that minimize the delay, as shown in Fig. 6.1(a), and the exact sizes of the wire segments independently ($L_1 \neq L_2 \ldots \neq L_n$) that minimize this delay, as seen in Fig. 6.1(b).

It is also interesting that these efforts have been extended to include the more elaborate problem of wire sizing, where the wire lines are divided into a discrete set of segments, and each segment is sized independently ($W_1 \neq W_2 \ldots \neq W_n$), as represented in Fig. 6.2. In fact, the proper wire sizing has been proven to minimize the signal delay in an RC interconnect line [111].

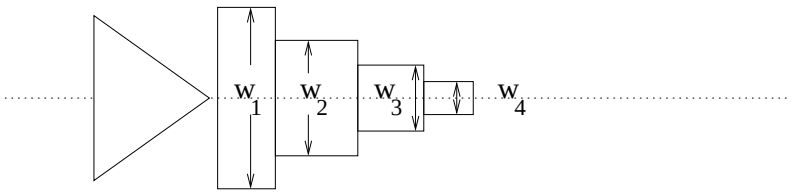


Figure 6.2: Wire sizing.

More advanced techniques have involved the combined buffer insertion/wire sizing problem while minimizing the power consumption of the buffers [112, 113]. These techniques have employed a delay relaxed formula ($D = \kappa t_{crit}$), where t_{crit} was the minimum achievable delay on a global net, and $\kappa > 1$. Therefore, κ was used as a control knob trading off the delay of the interconnect for its power consumption. The work in [112] has demonstrated that a 15% relaxation in delay can save up to 33% of the power consumed by the interconnect buffers. These efforts has been adapted to plot the delay-power trade off curve in order to determine the optimal number of buffers and their respective positions [8, 114–117].

Analytical approaches lack the capacity to accommodate buffer blockage, which is defined as the presence of macro blocks that prevent buffer insertion. The fact that the closed form solutions can provide the optimal number of buffers and their separation as a fixed wire length, limits their capability to deal with the blockage. Moreover, analytical solutions can even prevent an optimal solution, if the blockage area is too large. This is not the case if a dynamic programming or graph-based algorithm is employed since they generally allow for unequal buffer spacing.

Typically, *dynamic programming* techniques depend on finding a solution within the framework of a graph theoretic representation of the problem. Usually, such techniques are derived from the extension of van Ginneken's algorithm [118] to find the optimal delay within the framework of the optimal power. In [119], fixed buffer locations have been used, and in [120], an $O(n^5)$ algorithm has been employed to find the optimal setup. In addition, Sapatnekar et al. [121] have formulated this problem with the framework of fixed buffer locations. Although, in most of these efforts the blockage has not been explicitly considered, it can be easily incorporated into these methodologies. To combine the efforts to find a power optimal solution, a power optimal maze routing methodology has been devised in [12]. This work extends the efforts of delay optimal maze routing and buffer insertion in [122] by considering the shortest paths algorithm to find the power optimal path. The drawback in [122] is the inability to find the routing tree for the multi-pin nets. These nets represent 30% to 40% of the interconnects on a modern chip [123]. This issue has been addressed in [13] at the expense of a significant increase in the total

runtime.

6.3 Preliminaries

In this section, some of the fundamental concepts related to the new formulation are discussed.

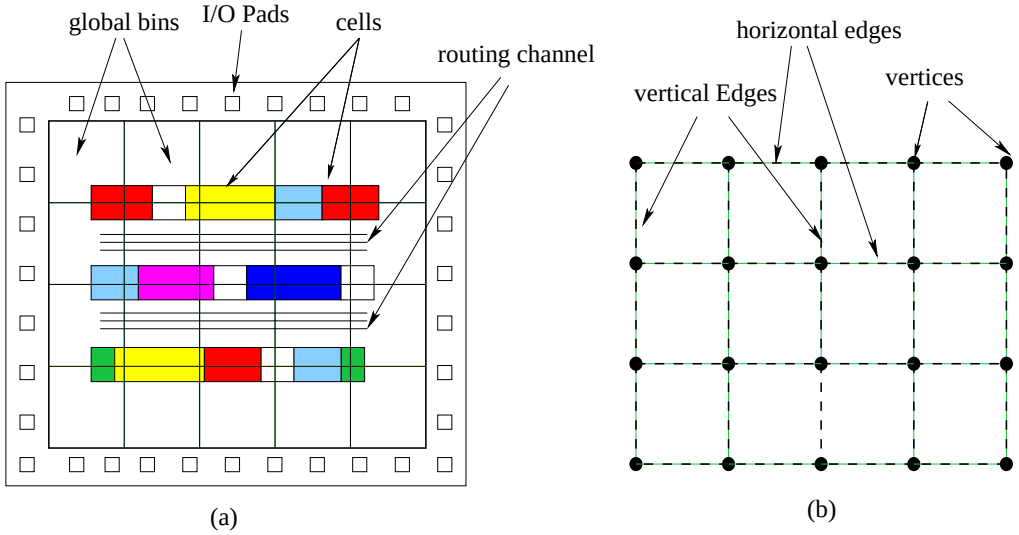


Figure 6.3: Grid graph for standard-cell based designs. (a) Global bin graph. (b) The corresponding grid graph.

6.3.1 Global Routing Problem

In global routing, the connection pattern for each net that satisfies the different objectives must be decided. The input to the global routing problem consists of a net-list that indicates the interconnections between the terminals and placement information, including the terminal positions and the location of the routing channels between them. Typically, the global routing problem is presented as a graph problem, where the routing regions and the module connections are modeled by using a grid graph. Initially, a given circuit is partitioned into a set of rectangular regions, called global bins. After the cells are placed in these bins, each cell is assumed to be placed at the center of the global bin, as depicted in Figure 6.3(a).

From Fig. 6.3(b), it is clear that the global bins and edges can be transformed into a grid graph. The vertices of the graph represent the possible positions of the interconnect

terminals, and the horizontal and vertical edges (called the grid edge) that lie between two adjacent vertices represent channels that can be used for wire routing. A net is an unordered set of points on the grid graph. A route (or tree) of a net is a set of grid edges for connecting all the terminals of the net. Since the routing resources are finite, each grid edge has a capacity. With such a graph representation, the graph version of the global routing problem, instead of the original problem can be solved.

6.3.2 Global Routing Techniques

Since all versions of the global routing problem are NP-hard [124], a variety of heuristic algorithms have been developed for it. They are classified as *sequential global routing* and *concurrent global routing* algorithms. The most common approach to global routing is sequential routing [1]. In such an approach, the nets are first ordered according to their importance, and then based on the ordering, the nets are routed sequentially. The quality of a sequential global router largely depends on the ordering of nets. Due to the sequential nature of these techniques, they fail to give adequate results. Besides, the sequential heuristic techniques cannot provide a key answer as to whether or not a feasible solution exists. In other words, if they fail to yield a feasible solution, it is not clear whether this is attributable to the non-existence of a feasible solution or due to the shortcomings of the heuristic. Moreover, when a heuristic does find a feasible solution, it is not known whether or not this solution is optimal, or how far it is from optimality.

To avoid the net ordering problem and to make the solution more predictable, concurrent-based global routing algorithms, in the form of Integer Programming models, have been developed to route all the nets simultaneously. In the mathematical programming-based approach, global routing is formulated as a 0/1 integer programming problem. Given a set of Steiner trees for each net and a routing graph, the objective of the Integer Programming technique is to select a Steiner tree for each net from its set of Steiner trees without violating the channel capacities while minimizing the total wire-length [125]. This approach tends to result in a more global solution and no initial ordering of the nets is required.

A general formulation of the ILP based global routing is as follows:

$$\text{Minimize } \sum_{j=1}^{t} b_j x_j, \tag{6.1}$$

subject to

[1] Most industrial tools utilize Maze Routers as a solver.

$$\sum_{x_j \in N_k} x_j = 1, \quad k \in \{1, ..., n\},$$
$$\sum_{j=1}^{t} a_{ij} x_j \leq Cap_j, \quad i \in \{1, ..., p\},$$
$$x_j \in \{0, 1\} \quad j \in \{1, ..., t\} \tag{6.2}$$

In this model, The global routing problem is formulated as a 0/1 ILP problem by associating a variable x_j with each tree which connects a net. The variable x_j equals "1" if that particular tree is selected and "0" otherwise. The constant b_j is the cost of connecting a net using jth tree. Cap_j is the edge capacity representing the routing supply of each edge. For the second constraint, all the possible tree combinations created for each net in the two routing layers are represented by a (0,1) matrix $[A_{ij}]$, with the i^{th} row corresponding to the i^{th} edge in the grid graph and each column corresponding to the possible tree combinations for each net. The element a_{ij} is expressed as:

$$a_{ij} = \begin{cases} 1 & \text{if tree j passes through edge i;} \\ 0 & \text{otherwise.} \end{cases}$$

There are two types of constraints in this formulation. The selection of one and only one routing for each net is forced by the first set of constraints. The edge capacity requirements of each edge are represented by the second set of constraints. The time to solve the Integer Programming problem increases exponentially with the number of Steiner trees generated in the formulation of the program. Therefore, techniques to solve the ILP-based global routing problem efficiently becomes a significant issue.

6.3.3 Interconnect Modeling

In this research, the Elmore delay model is chosen for its simplicity [126] for modeling the delay of the interconnects. The model states that the delay D_i from the source node i (as shown in Fig. 6.4) is calculated as follows:

$$D_i = \sum_{k=1}^{N} C_k R_{ik}, \tag{6.3}$$

where R_{ik} represents the resistance, shared among the paths from the source node s to nodes i and k, C_k is the capacitance of each wire segment between the nodes (Fig. 6.4), and N is the number of nodes.

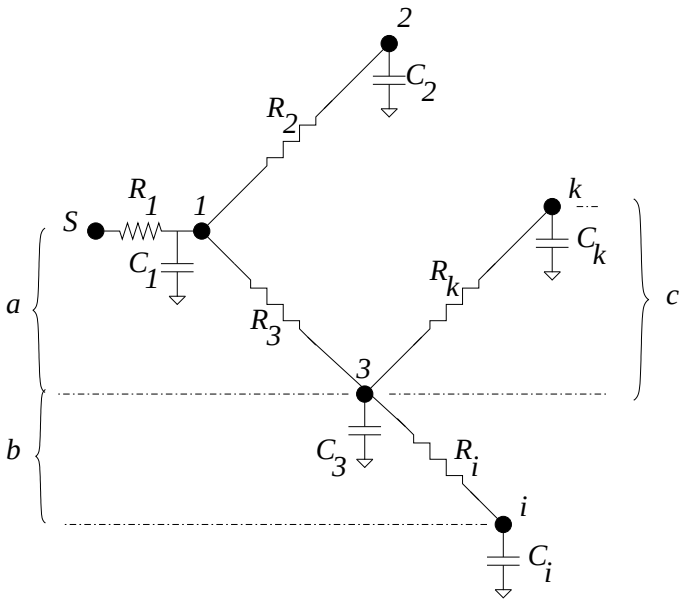


Figure 6.4: RC tree [11].

To estimate the interconnect wire capacitance, the model introduced in [127] is used. It accounts for the fringing, coupling, and plate capacitances for an interconnect according to the structure in Fig. 6.5. The unit length capacitance (c_{wire}) of the wire is[2]

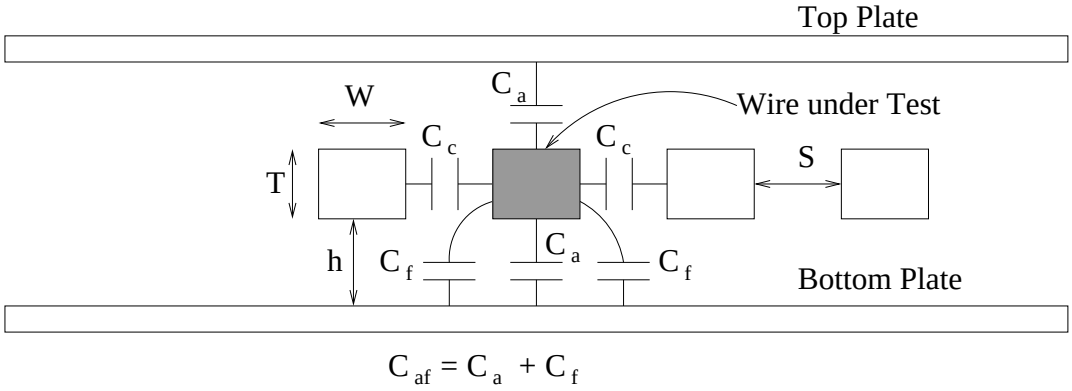


Figure 6.5: Wire structure for capacitance extraction.

$$c_{wire} = 2C_{af} + 2C_c, \tag{6.4}$$

where C_{af} is the total wire fringing and plate capacitance, and C_c is the coupling capacitance between the metal line and the adjacent metal lines. C_{af} is given by computing [127]

[2]The equations were used and evaluated by the Berkeley Predictive Technology Model (BPTM) which is provided by the Device Group at UC Berkeley [128].

$$\frac{C_{af}}{\epsilon_{ILD}} = \frac{W}{h} + 2.04\Big(\frac{S}{S+0.5355h}\Big)^{1.773} .\Big(\frac{T}{T+4.532h}\Big)^{0.071} \quad (6.5)$$

where ϵ_{ILD} is the interlayer-dielectric primitivity, W is the wire width, T is the wire thickness, S is the spacing between the wires, and h is the spacing between the wire and the ground plans. C_c is given by the following:

$$\frac{C_c}{\epsilon_{ILD}} = 1.411\frac{T}{S}exp\Big(\frac{-4S}{S+8.014h}\Big) +2.3074\Big(\frac{W}{W+0.3078S}\Big)^{0.25724} .\Big(\frac{h}{h+8.691S}\Big).exp\Big(\frac{-2S}{S+6h}\Big). \quad (6.6)$$

The ITRS [2] predictions are employed to estimate the interconnect wire resistance. Note that although an analytical model is chosen for the capacitance and resistance calculations, without the loss of generality, more exact extraction techniques can be used with no modification to the problem formulation in the next section.

In practice, the number of sinks, connected to the same driver without a buffer between the sinks, is small [129]. In fact, connecting too many sinks to a single source results in excessive delays that cannot be recovered by buffer insertion. Accordingly, if the degree of this net configuration is limited, the wire lengths of $a, b,$ and c in Fig. 6.4 can be computed by generating a limited set of Steiner trees between the driver and the set of sinks connected to it.

On the other hand, to estimate the total power, consumed by the driver and sink gates and the driven interconnect, the power consumption model employed in [8] is used. The switching power (P_{active}) is hence calculated as follows:

$$P_{active} = \alpha V_{DD}^2.f_{clk} \times (W_{driver}.C_o + c_{wire}.l + W_1.C_{L_{sink\ 1}} + \ldots + W_p.C_{L_{sink\ p}}), \quad (6.7)$$

where V_{DD} is the supply voltage, f_{clk} is the clock frequency, W_{driver} and $W_{1\ \ldots\ p}$ are the widths of the driver and all the sink gates connected to it, respectively, α is the switching factor which is assumed to be 0.15 [110], c_{wire} is the wire capacitance per unit length, C_o is the output capacitance of the driving gate and C_{L_i} is the loading capacitance of sink gate i. In addition, l is the total Steiner tree wire length.

The following section presents the performance and power driven ILP-based global routing technique followed by the experimental results.

6.4 Power-Efficient Multi-pin ILP Based Global Routing

In order to optimize the routing and interconnect delay, as well as to reduce the power consumed by the interconnect buffers, a **P**ower-efficient multi-pin **I**LP based global **R**outing **T**echnique (PIRT) is proposed in this section. This performance and power aware routing algorithm is based on the Integer Linear Programming (ILP)-based global routing model in [130].

The goal of the proposed PIRT is to find an efficiently powered buffer path for each net without violating the delay constraint. The routing area is modeled by using a routing grid that is similar to the one in Fig. 6.3(b). In this grid, $G = (V, E)$, each vertex u represents a buffer possible location, and each edge (u, v) represents a possible route for interconnect wires. In this case, the problem can be formally described as follows:

PIRT: *Given a predefined routing grid* $(G = (V, E))$*, a buffer library (B), a buffer function* $(f(u) = 1)$ *if a buffer is allowed at vertex* u*, and* $(f(u) = -1)$ *otherwise, and a set of nets* $(net_1, net_2, ..., net_i)$*, find the minimum power path for each net, subject to a delay constraint*

6.4.1 PIRT Phases

The PIRT flow can be illustrated by dividing it into two distinct phases, as shown in Fig. 6.6. First, an initialization phase (phase I) where the initial minimum Steiner trees are constructed for each net. To reduce the global routing congestion, additional detoured trees are also built for each net. In order to create enough routing alternatives considering the delay requirements several buffered trees (i.e., trees with buffers inserted) are built for each net, if possible. Next, a power optimization phase (phase II) is invoked. This phase attempts to find a low power route (i.e., tree) for each net so that the total power of all the nets is minimized while satisfying the delay requirement of the chip.

6.4.2 Phase I (Initialization)

In order to perform the interconnect power optimization in phase II, several routing alternatives are needed for each net. Creating routing alternatives allows the ILP model in phase II to pick a globally optimum solution that covers all the nets on the chip simultaneously. The routing alternatives are either unbuffered or buffered routes that connect the interconnect source to its sinks.

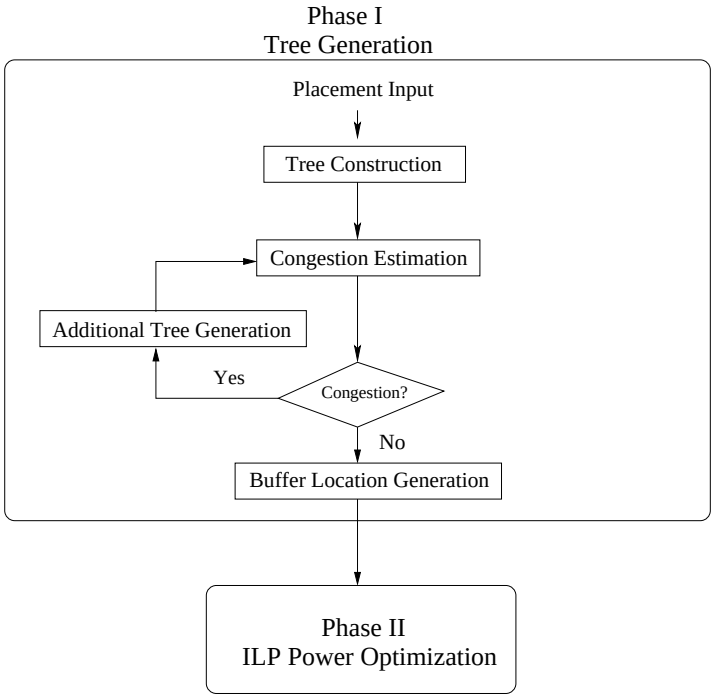


Figure 6.6: PIRT flow chart.

Unbuffered Tree Construction

The first step in this phase is to produce a set of admissible routes for each net. In a practical circuit, the terminals in a net are connected by horizontal and vertical wires. Therefore, only the rectilinear spanning trees or rectilinear Steiner trees are considered in the tree construction process. These trees become the unknown variables of the ILP problem. In the experimental section, FLUTE presented in [131] is used to construct the trees. Obviously, the number of trees for each net should not be too large, since the time complexity of the ILP problem is a function of the number of trees. However, a number of trees should be built for each net to guarantee the feasibility of the problem. To remedy this problem, an additional tree generation step is proposed in [130] to reduce the number of trees created for each net, while ensuring that the constructed trees will likely result in a promising (feasible and optimal) solution. Initially, the potentially congested areas in the routing graph are predicted by a heuristic technique [130, 132] in the congestion estimation stage. This a priori congestion information is then used in the additional tree generation stage to eliminate the congested areas by adding trees to the nets passing through these areas iteratively. The congestion of the circuit is re-estimated after each additional tree generation step.

Complexity Analysis of Tree Generation

The complexity of the initial tree construction for each net is $O(m\log m)$, where m is the degree of the routed net [131]. The complexity of the tree construction for the whole chip is $O(N.m\log m)$ where N is the total number of routed nets. The complexity of congestion estimation is $O(E)$, where E is the total number of edges in the graph. The complexity of the additional tree construction is $O(C)$, where C is the total number of nets that pass through the congested edges [130]. Since the maximum value of C is less than N, $E \ll N$ and $m \ll N$, the complexity of the initial tree construction, congestion estimation and the additional tree construction is $O(N)$.

Buffered Tree Construction

Since unbuffered nets are not guaranteed to achieve the timing requirements of the chip, a set of buffered routes has to be added to the unbuffered trees generated earlier to extend the alternatives presented to the optimization step at phase II.

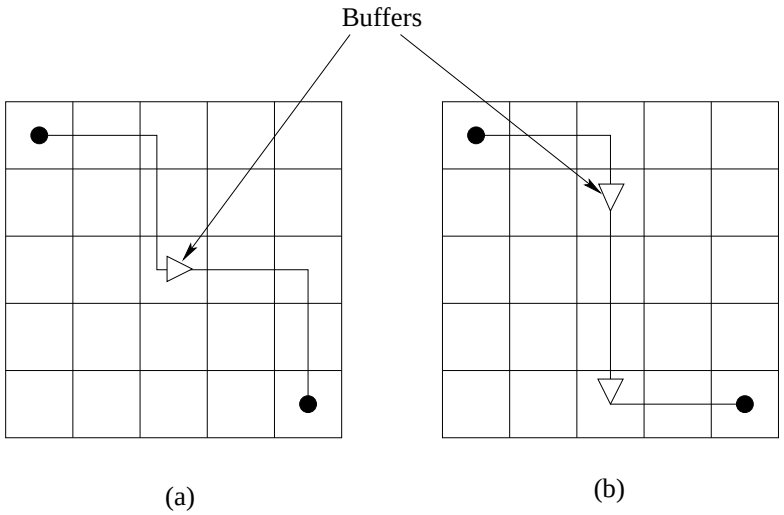


Figure 6.7: Buffer insertion for two-terminal nets (a) buffered-tree with one buffer. (b) buffered-tree with two buffers.

However, due to the preplaced modules there is usually a limited number of available buffer locations. In this formulation, each location is expected to allow for a limited number of buffers to be inserted. Accordingly, in our current implementation, for each tree of a two-terminal net, at most two buffered trees are produced: one with one buffer inserted and the other with two buffers inserted, as denoted in Fig. 6.7. The buffer insertion process of three-terminal nets begins by identifying the Steiner point or middle

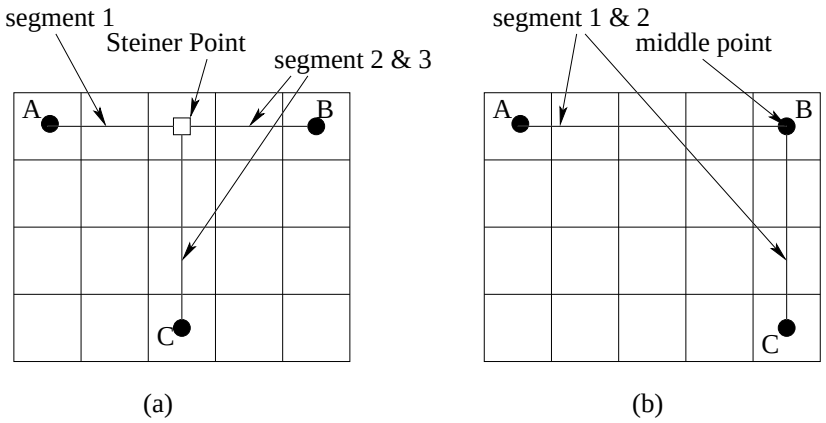


Figure 6.8: Buffer insertion for three terminal nets (a) three terminal net with Steiner point. (b) three terminal net with middle point.

point of the net in Fig. 6.8. Following this, each net is divided into two or three segments, based on the presence of either a Steiner point or a middle point. If one segment is much longer than the other segments ($> 90\%$), the buffer insertion is considered for only the longest segment. Otherwise, all the segments are sorted in descending order, based on their length, and one buffer is inserted in each segment. For the generated buffered two and three terminal nets, only the trees, where the addition of a buffer results in the reduction of the delay are added to routing alternatives for Phase II.

Limiting the alternatives to two buffered trees per unbuffered tree assists in limiting the runtime of the algorithm. However, this limit can be removed at the expense of longer runtime. Also, limiting the number of buffers to at-most two is a runtime trade-off that can be tuned.

In addition, due to the limited availability of buffer locations, all the nets are first sorted in descending order according to their wirelength such that the long nets have a higher priority for buffered routes generation. However, the priority of short critical or highly loaded nets can be easily elevated to allow for buffered routes to be created for these nets. It is important to note that the buffer route generation priority does not affect the non-ordering nature of the formulation specially for the congestion consideration in phase II.

Complexity Analysis of Buffered Tree Generation

Fig. 6.9 illustrates the pseudocode of the buffer generation algorithm. The time complexity of step 1 (i.e., sorting all the nets) is $O(N\log N)$, where N is the total number of routed nets. For step 2, the complexity is $O(NT)$, where T is the number of trees built

Figure 6.9: Buffered tree generation algorithm.

```
1. Sort all the nets based on the wirelength.
   QuickSort(Nets)
2. Insert buffer for each sorted net.
   For each net i
      For each tree j of net i
         find buffer location and insert buffer;
         calculate the power and delay;
         if(inserted buffer reduces delay)
            connect the buffered tree to
            the tree list of net i ;
      End For
   End For
3. Stop.
```

for each net. Since T is a constant the complexity is $O(N)$. Accordingly, the complexity of the whole algorithm is $O(N\log N)$.

6.4.3 Phase II (Power Minimization)

To achieve PIRT's objective to minimize the power consumption of the interconnect while satisfying the chips delay constraints a constant Max_Delay is needed. Max_Delay is the maximum acceptable delay for all the nets which is derived from the clock frequency set by the product specifications. Since, any net having shorter delay than Max_Delay is unnecessarily fast, PIRT strives to trade this delay slack with the power consumption.

Accordingly, PIRT power minimization under Max_Delay constraint is presented as follows:

$$\text{Minimize } \sum_{j=1}^{t} w_{pj} x_j, \tag{6.8}$$

subject to

$$\sum_{x_j \in N_k} x_j = 1, \quad k \in \{1, ..., n\}, \tag{6.9}$$

$$\sum_{x_j \in N_k} D_{x_j} x_j \leq Max\_Delay, \quad k \in \{1, ..., n\}, \tag{6.10}$$

$$\sum_{j=1}^{t} a_{ij} x_j - Cap_i \leq Z_i, \quad i \in \{1, ..., p\}, \tag{6.11}$$

$$x_j \in \{0, 1\} \quad j \in \{1, ..., t\},$$

$$Z_i \in \{0, C\},$$

Similar to (6.1) introduced in Section 6.3, The global routing problem is formulated as a 0/1 ILP problem by associating a variable x_j with each tree which connects a net. The variable x_j equals "1" if that particular tree is selected and "0" otherwise. The first constraint ensures that only a single tree among all the possible trees generated for net k is selected. w_{pj} is the weight associated with the power of tree j and is calculated as:

$$w_{pj} = \frac{power\ of\ tree\ j}{max\ power\ of\ trees\ constructed\ for\ net\ k} \tag{6.12}$$

The power of tree j[3] is modeled and calculated by equation (6.7).

For the second constraint, D_{xj} is the delay of tree j calculated using equation (6.3). This constraint ensures that for all the selected nets, the delay of each net does not exceed the Max_Delay.

For the third constraint, all the possible tree combinations created for each net in the two routing layers are represented by a (0,1) matrix $[A_{ij}]$, with the i^{th} row corresponding to the i^{th} edge in the grid graph and each column corresponding to the possible tree combinations for each net. The element a_{ij} is expressed as:

$$a_{ij} = \begin{cases} 1 & \text{if tree j passes through edge i;} \\ 0 & \text{otherwise.} \end{cases}$$

Since the routing resources are finite, each grid edge has a capacity. The capacity of each edge is represented by constraint (6.11), where p is the number of edges on the grid graph, t is the total number of trees produced for all the nets, and Cap_i is the edge capacity of the i^{th}edge. In order to achieve a global routing solution a slight allowance

[3]Low activity nets can be pruned quickly from the search space to eliminate unnecessary runtime overhead.

on the overflow is set by Z_i which is a variable associated with the routing overflow of each edge. C is the upper bound on Z_i. It is a positive value obtained from experimental results. The value of C represents the minimum number of extra tracks needed by the detailed router to achieve a fully routed chip.

Solving the minimization problem set by equation (6.8) ensures that the final net selection simultaneously achieves the delay constraints and minimize the power consumption.

6.5 Experimental Results

The ILP-based router is implemented in C++ on a 900 MHz Sun Blade 2000 workstation with a 1 GB memory. FLUTE [131] is used for the Steiner tree construction, and iLog CPLEX10.0 package is used as the ILP solver. Table 6.1 shows the statistics of the ISPD98 IBM benchmarks [133] and ISPD2007 benchmarks [134], used in this work.

Table 6.1: ISPD98, IBM, and ISPD2007 benchmark statistics.

Benchmarks	Total Nets	H/V Cap	Grid Size	Chip Size (mm)
ibm08	35195	32/21	64×192	1.2×3.6
ibm09	39592	28/14	64×256	1.2×4.8
ibm10	49491	40/27	64×256	1.2×4.8
ispd01	176715	70/70	324×324	5×5
ispd02	207972	80/80	424×424	5×5
ispd03	323887	92/92	474×479	5×5
ispd04	357104	92/92	474×479	5×5

It is important to note that for these benchmarks, two-terminal and three-terminal nets constitute the majority of the nets in all the test benchmarks. The column, "Total Nets", indicates the total number of nets in each benchmark. The column, "H/V Cap", lists the horizontal and vertical edge capacity. Additionally, Table 6.1 lists the grid size and the chip size to demonstrate the size of the routing problem. Based on the work in [12], the chosen 130nm technology requires a buffer library that spans the range between 5 and 15 times the minimum sized buffer of Wp/Wn = 260/130nm to efficiently buffer the nets.

Accordingly, the buffer library for these tests represents three types of buffers; weak, medium and strong. They are chosen to correspond to three different buffer sizes:

$5, 10, 15\times$, the minimum buffer size of the $130nm$ technology. This allows the verification of the technique for different buffer driving capabilities. Table 6.2 lists the parameters of the $130nm$ technology used for the power and delay calculation.

Table 6.2: Technology and equivalent circuit model parameters for global interconnects.

Parameters for 130nm Technology	
Oxide thickness $T_{ox}(nm)$	2.3
Gate relative dielectric constant ϵ_r	3.9
ILD relative dielectric constant ϵ_r	3.0
Transistor length $L_{min}(nm)$	130
Supply voltage $V_{DD}(V)$	1.2
Clock frequency $f_{clk}(GH_z)$	1.6
Saturation current $I_{DD}(\mu A/\mu m)$	900
Subthreshold leakage $I_{off}(\mu A/\mu m)$	0.01
Parasitic capacitance percent of ideal gate capacitance	19%
Wire width $W_{wire}(nm)$	670
Wire width / thickness A/R	2.0
Wire height $h_{wire}(nm)$	670
Wire spacing $s_{wire}(nm)$	670
Wire resistance $r_{wire}(k\Omega/m)$	24.5
Wire capacitance $c_{wire}(pF/m)$	211.4

To fully quantify the performance of the proposed technique, the power savings, routing quality, and runtime need to be discussed. Consequently, the following sections introduce the results for these parameters when the PIRT is applied to the multi-pin netlists in different benchmarks.

6.5.1 Experimental Results for Multi-Pin Nets

6.5.1.1 Power Savings

To calculate the power savings achieved by applying PIRT the value of Max_Delay needs to be identified. In addition, a baseline that does not perform power optimization has to be established for comparison purposes. In the absence of clock frequency constraints in the IBM and ISPD benchmarks the Max_Delay value has to be identified

using the available routing information. Since Max_Delay is the longest net delay when the chip is properly routed, i.e. all nets are optimized for their minimum delay, a delay minimization model that uses the data from PIRT's phase I and solves for the minimum delay instead of minimum power will properly route the chip and the longest net delay can be easily extracted. In addition, the power consumed by the final routing solution of the delay minimization model is the perfect baseline to identify the power savings achieved by PIRT in comparison to this delay minimization model.

6.5.1.2 Delay Minimization

In order to define the Max_Delay and the power consumption of delay optimal routed chip the following delay minimization model is solved.

$$\text{Minimize } \sum_{j=1}^{t} w_{dj} x_j, \tag{6.13}$$

subject to

$$\begin{aligned}
&\sum_{x_j \in N_k} x_j = 1, \quad k \in \{1, ..., n\}, \\
&\sum_{j=1}^{t} a_{ij} x_j - Cap_i \leq Z_i, \quad i \in \{1, ..., p\}, \\
&x_j \in \{0, 1\} \quad j \in \{1, ..., t\}, \\
&Z_i \in \{0, C\},
\end{aligned} \tag{6.14}$$

where x_j represents tree j built for net k, Cap_i is the routing supply of each edge, and Z_i is a variable associated with the routing overflow of each edge. w_{dj} represents the weight, associated with the delay of tree j and is calculated by the following:

$$w_{dj} = \frac{delay\ of\ tree\ j}{max\ delay\ of\ trees\ constructed\ for\ net\ k} \tag{6.15}$$

The delay of tree j is modeled and calculated by (6.3).

In order solve the delay minimization model a number of allowable buffer locations (about 10% of the total vertices) are generated(Fig. 6.10).

After the delay minimization problem is solved, one tree is selected for each net k such that the total delay of all the nets is minimized under the routing overflow constraint. The delay of net k equals the delay of the selected tree. The maximum delay of a circuit

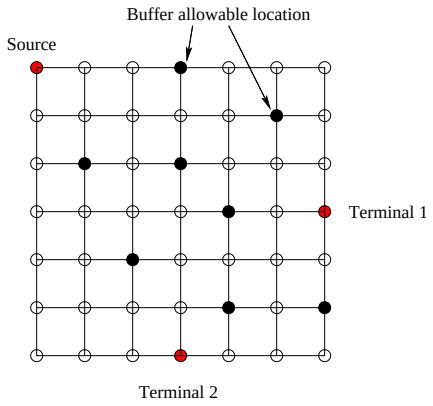


Figure 6.10: Buffer location generation.

is defined as $Max\_Delay = max_{k \in N}\{d_k\}$, where d_k is the delay of net K after the delay minimization. In addition, the power consumed by the final solution represents the baseline power consumption for the calculation of the power savings.

6.5.1.3 Power Savings Comparison

Fig. 6.11 depicts the power savings achieved when the PIRT is applied to the various benchmarks in Table 6.1 in comparison to the power consumption baseline established in Section 6.5.1.2. In addition to the average power savings for the IBM and ISPD benchmarks, respectively. This figure also compares the power savings when considering only the two and three terminal nets that were buffered in phase I to the total power savings when all the nets are included.

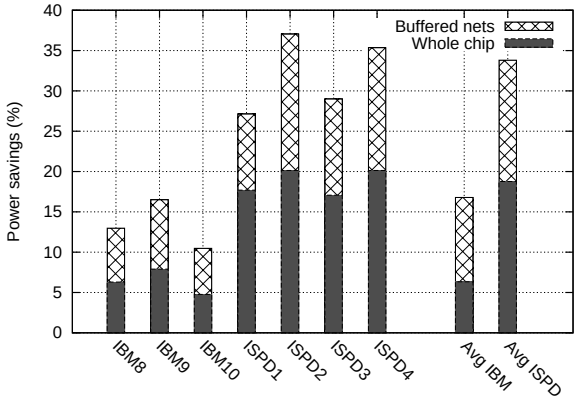


Figure 6.11: Power savings by PIRT and the average for IBM and ISPD benchmarks (strong buffer size.)

For the two and three terminal nets, the average savings for the IBM benchmarks and

ISPD2007 benchmarks are 16% and 32%, respectively. These values drop to 6.5% and 19%, when the remaining unbuffered nets are factored in. Since the global interconnects consume up to 21% of the total dynamic power [9], PIRT is capable of saving up to 4% of the total dynamic power. A comparison of the results for the IBM and ISPD benchmarks indicates a significant increase in savings for the ISPD benchmarks. This can be attributed to the fact that the larger ISPD benchmarks have a higher number of long buffered nets which are the primary target of the PIRT. Since the trend for future chips is an increase in size,[4] this enhanced performance of the PIRT for the ISPD benchmarks emphasizes the importance of the PIRT as a power management technique.

Since PIRT specifies the delay constraint on the chip to be the delay of the worst net, it is interesting to see how the PIRT affects the delay of the second worst net on the chip. Fig. 6.12 shows the percentage difference between the delay of the worst and the second worst nets. It is evident that the effect of the PIRT is to reduce the slack between these two nets and translate it into power savings. Consequently, the average reduction of the slack from 29% to 7% explains the savings in Fig. 6.11.

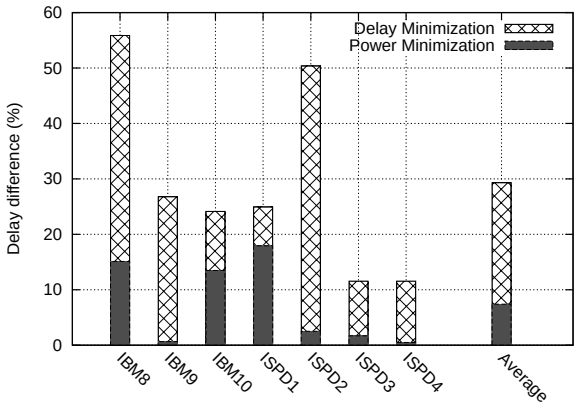


Figure 6.12: Delay difference for the second longest net between delay minimization and power minimization.

Finally, a comparison of the PIRT average power savings with the different buffer sizes in Fig. 6.13 shows that PIRT has more potential for larger buffer libraries. This is explained by noting that the removal of a large power hungry-buffer from the strong library results in higher savings, compared to the weak buffers, less power hungry, from the same library. Although it is tempting to try to route the nets by using the weakest buffer from the beginning, a significant hit to the signal integrity of the routed nets occurs. Accordingly, the use of a mixed buffer library ensures that the nets are optimally performing over a wide range of chip sizes.

[4] Although the area might not change, the number of nets and buffers are exponentially growing.

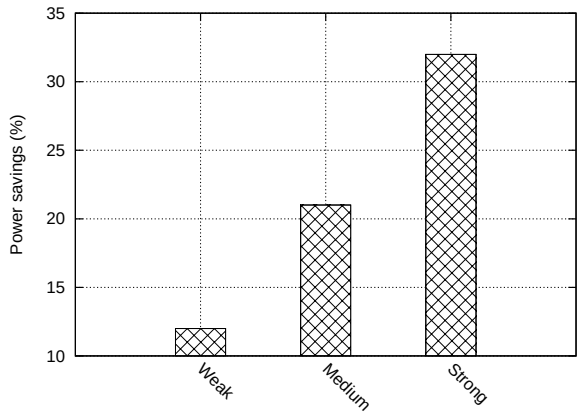


Figure 6.13: Average power reduction over different buffer sizes (5,10 and 15 times the minimum sized buffer).

6.5.1.4 Routing Quality

Table 6.3 provides the total wirelength, number of bends, and overflow of the delay minimization model, and power minimization model for the buffer size 15 x 130nm. As shown, the power minimization model reduces the total overflow significantly without increasing the total wirelength and total number of bends for both the IBM and ISPD benchmarks. The overflow is reduced since PIRT allows for a more relaxed constraint on the non-critical nets allowing for more detours. Although the wirelength of some nets might slightly increase, the freed up tracks usually allows for an overall reduction of power consumption by properly routing heavily loaded nets.

Table 6.3: Comparison of delay minimization and power minimization models.

Routing Results for Multi-pin Nets						
Circuit	Delay Minimization (15x130nm)			Power Minimization (15x130nm)		
	Wirelength	Bends	OverFlow	Wirelength	Bends	OverFlow
ibm08	1147507	9761	17	1147495	9592	6
ibm09	1043231	10264	634	1042778	10119	436
ibm10	1988162	13531	1180	1985537	13246	946
ispd01	1845740	61029	5	1845718	60962	0
ispd02	1698413	87202	53	1698048	86560	15
ispd03	2887420	106369	13	2887350	105656	3
ispd04	2950644	139298	4	2950627	138948	0

6.5.1.5 Computation Time

Table 6.4: Computation time comparison of PIRT for buffer size 15 x 130nm.

Computation Time of the PIRT (Routing Multi-pin Nets)					
Circuit	PIRT Method (buffer size:15 x 130nm)				
	T-Time(s)	B-Time(s)	S-Time(s)	Tot-Time(s)	T/PerNet(s)
ibm08	95	31	1	127	0.003
ibm09	167	46	9	222	0.005
ibm10	182	57	15	254	0.005
ispd01	5412	1850	28	7290	0.04
ispd02	12713	3290	50	16053	0.07
ispd03	23482	8197	880	32559	0.1
ispd04	24071	11007	980	36058	0.1

Table 6.4 displays a comparison of the computation time of the different phases of the PIRT. Columns “T-Time”, “B-Time”, “S-Time”, and “Tot-Time” reveal the computation times of the tree construction phase, buffer insertion phase, power or delay model solving time, and the total time, respectively. It is clear that most of the computation time is consumed by the tree construction phase which is common to most other routing algorithms. Buffer insertion and power minimization phases consume a relatively small part of the total computation time. Due to this small overhead, the PIRT manages to achieve its goal of power reduction without affecting the total runtime. This enables many existing routing techniques to benefit from the inclusion of the PIRT for power minimization.

Finally, Table 6.5 compares the total runtime of PIRT versus state of the art power minimization techniques. Since all existing techniques target the power minimization of single nets and expect a sequential router to finalize the chip routing, they do not account for congestion. On the other hand PIRT is capable of minimizing power, while managing congestion with considerably lower average runtime.

6.6 Summary

Timing optimization and low power are important goals in global routing, especially in deep submicron designs. Previous efforts that focused on power optimization for

Table 6.5: Computation time comparison with power driven routers.

Router	Processor and Memory Specifications	Overflow avg/max	Runtime(s)/net avg/max
PB [93]	1.9GHz/2GB	NA	472/1859
FREEZE [135]	2.8GHz/1GB	NA	4.09/12.7
GP [136]	3.2GHz/0.5GB	NA	<1
PRI [137]	2.2GHz/1GB	NA	<0.12
PIRT	0.9GHz/1GB SparcV9	200/946 ($<2\%$)	0.03/0.1

global routing are hindered by excessively long run times or the routing of a subset of the nets. Accordingly, the development of a multi-objective power efficient multi-pin global routing technique (PIRT) is a critical component towards balancing chip's power budget. The integer linear programming based technique strives to find a power efficient global routing solution. The results indicate that an average power savings as high as 32% for the 130-nm technology can be achieved with no impact on the maximum chip frequency.

Chapter 7

Conclusions and Future Work

As transistor sizes shrink, the total power consumption of chips is becoming a dominant factor in determining the chip performance. The power consumption of microprocessor cores and interconnects constitutes a significant portion of the total power consumption of modern microprocessors.

In this thesis, two power management techniques that tackle the power consumption of the processor core and interconnects are developed. Both techniques represent a cornerstone in the quest to achieve power budget closure for modern processors.

The concept of predictive sleep signal generation is developed through the design of a Dynamic Sleep Signal Generator (DSSG) that is capable of tracking the processor standby profile. The DSSG is responsible for the accurate generation of the sleep signal for microprocessor functional units through the use of a simple finite state machine. The DSSG is evaluated in comparison to a Static Sleep Signal Generator that triggers the sleep signal based on a simple counter that is shown to be incapable of tracking the processor behavior. A DSSG model is implemented in C++ to verify the accuracy. In addition, a custom built 90nm DSSG circuit is designed to verify the power saving capabilities.

The DSSG exhibits accuracies up to 80% with leakage savings as high as 146% compared to the SSSG. The newly proposed DSSG allows the processor to maximize its leakage savings while minimizing the impact on the processor performance through the reduction of the number of unnecessary assertions of the sleep signal.

In addition, a Power-efficient multi-pin ILP based global Routing Technique (PIRT), that is able to accommodate simultaneous routing, congestion minimization and buffer insertion is formulated. The methodology is able to account for the buffer and wire blockage, and performs the optimization without affecting the chips maximum frequency.

The PIRT is formulated in two phases comprising buffered and unbuffered tree generation in addition to solving a power optimal ILP based model for minimizing power while meeting the chips delay constraints. The tree generation steps are $O(N\log N)$ where N is the total number of routed nets. While the power minimization model was solvable in less than 0.1 second per net for several industrial benchmarks. Finally, PIRT provides up to 32% power savings for the 130nm technology used in the formulation.

7.1 Contributions

The contributions of the research for this thesis can be summarized as follows:

Leakage power optimization for microprocessor cores

1. Contrary to compiler-based low leakage techniques that are bound to a specific Instruction Set Architecture (ISA), the DSSG depends on only the information about current and previous standby periods.

2. Phase extraction techniques that are concerned with the coarse granularity prediction of program phases are not adequate for the task of low leakage management. These techniques are generally slow and perform complex tasks, prohibiting the techniques' application in small-scale leakage management. In contrast, the DSSG focuses on a fine granularity analysis (a few hundred cycles) of the profile information.

3. Current circuit techniques for leakage power reduction depend on detecting when the circuit is actually in standby, which leaves the techniques prone to erroneous decisions regarding very short standby periods, since the techniques lack the ability to predict the length of the period ahead of time [3–5]. Accordingly, the proposed DSSG should eliminate these short periods from the decision tree of these techniques

4. The proposed DSSG and the associated finite state machine are capable of tracking the executed program behavior across different time segments and predict the length of the standby periods accordingly. This leads to a high accuracy in asserting the sleep signal when it is most likely to achieve a net increase in the total power savings. This is accomplished with minimal power overhead.

5. The DSSG is a simple hardware based approach. This allows the incorporation of the DSSG in existing microprocessors (General purpose and embedded) with minimal design overheads.

6. The DSSG finite state machine does not rely on any memory like structure reducing the layout footprint and the overall power consumed to achieve the higher prediction accuracy targeted by the DSSG.

7. The DSSG exhibits low power consumption, in the order of 300 μW to achieve accuracies up to 80% in predicting the length of the standby period.

Dynamic power optimization for global interconnects

8. Unlike previous approaches, the newly developed approach is capable of timing optimization, buffer insertion and power reduction simultaneously with routability consideration.

9. The optimization of power consumption and simultaneously accounting for the buffer blockage, which has not been considered in previous analytical formulations of the power optimization problem, is formulated.

10. The optimization of the power consumption without affecting the chip maximum frequency.

11. The problem is formulated so that it is independent of the delay and the power models used, allowing for more flexibility in applying the new technique to scaled technologies.

12. PIRT is capable of simultaneously routing and power optimizing the chip with runtime less than 0.1 second per net.

7.2 Future Work

Since the DSSG has shown tremendous potential for managing the leakage in processor functional units, it is expected that an extension of the DSSG to handle processor caches will be beneficial as well. A cache targeting DSSG will require careful attention to the shutdown procedure of the caches since a single cycle wake-up will inevitably strain the power delivery grid.

In addition to cache targeted DSSG, state preserving low leakage techniques can be employed to reduce the overhead of repopulating the caches on power up. However, it is expected that predictively managing leakage in processor caches will require significant changes the cache design.

Although the DSSG was tested on the RISC architecture the analysis of the DSSG on CISC and multicore architectures can further extend the application domain of the DSSG.

Finally, PIRT can be extended by further enhancement of the tree generation phase. Priority tree generation where non critical nets can be tagged for elimination. In addition, the ability to elevate the priority of short critical nets can be easily incorporated. PIRT can also be extended using enhanced buffer insertion techniques. These techniques are expected to yield higher performance by increasing the quality of the buffered trees constructed.

Publications

Publications resulting from this research

[1] Ahmed Youssef, Mohab Anis and Mohamed Elmasry, "A Comparative Study between Static and Dynamic Sleep Signal Generation Techniques for Leakage Tolerant Designs," *IEEE Transactions on VLSI Systems*, accepted (in press)

[2] Ahmed Youssef, Mohamed Zahran, Mohab Anis and Mohamed Elmasry "On the Power Management of Simultaneous Multithreading Processors", *IEEE Transactions on VLSI systems* (under review)

[3] Ahmed Youssef, Zhen Yang, Mohab Anis, Shawki Areibi, Anthony Vannelli, and Mohamed Elmasry, "A Power-Efficient Multi-Pin ILP Based Routing Technique", to *IEEE Transactions on Circuits and Systems-I* (under review)

[4] Ahmed Youssef, Mohab Anis and Mohamed Elmasry, "Dynamic Standby Prediction for Leakage Tolerant Microprocessor Functional Units," *Proceedings of the IEEE/ACM conference on Microarchitecture (MICRO)*, 2006, pp. 371-381.

[5] Ahmed Youssef, Tor Myklebust, Mohab Anis and Mohamed Elmasry, "A Low-Power Multi-Pin Maze Routing Methodology," *Proceedings of the IEEE International Symposium on Quality Electronic Design (ISQED)*, 2007, pp. 153-158. (Best Paper Award Nomination)

References

[1] J. Shen and M. Lipasti, *Modern Processor Design: Fundamentals of Superscalar Processors*. Boston: McGraw-Hill, 2005.

[2] "International Technology Roadmap for Semiconductors," 2002. [Online]. Available: http://public.itrs.net/Files/2002Update/2002Update.htm.

[3] J. Tschanz, S. Narendra, Y. Ye, B. Bloechel, S. Borkar, and V. De, "Dynamic Sleep Transistor and Body Bias for Active Leakage Power Control of Microprocessors," *IEEE Journal of Solid-State Circuits*, vol. 38, no. 11, pp. 1838–1845, Nov. 2003.

[4] Z. Hu, A. Buyuktosunoglu, V. Srinivasan, V. Zyuban, H. Jacobson, and P. Bose, "Microarchitectural Techniques for Power Gating of Execution Units," *Proceedings of the International Symposium on Low Power Electronics and Design*, pp. 32–37, 2004.

[5] S. Dropsho, V. Kursun, D. H. Albonesi, S. Dwarkadas, and E. G. Friedman, "Managing Static Leakage Energy in Microprocessor Functional Units," *IEEE/ACM International Symposium on Microarchitecture*, pp. 321–332, 2002.

[6] A. Bellaouar and M. I. Elmasry, *Low-Power Digital VLSI Design*. Boston: Kluwer Academic Publishers, 1995.

[7] J. Cong, L. He, C. Koh, and P. H. Madden, "Performance Optimization of VLSI Interconnect Layout," *Integration, the VLSI Journal*, vol. 21, no. 1-2, pp. 1–94, Nov. 1996.

[8] K. Banerjee and A. Mehrotra, "A Power-Optimal Repeater Insertion Methodology for Global Interconnects in Nanometer Designs," *IEEE Transactions on Electron Devices*, vol. 49, no. 11, pp. 2001–2007, Nov. 2002.

[9] N. Magen, A. Kolodny, U. Weiser, and N. Shamir, "Interconnect-Power Dissipation in a Microprocessor," *Proceedings of International Workshop on System Level Interconnect Prediction*, pp. 7–13, Feb. 2004.

[10] D. Sylvester, H. Kaul, K. Agarwal, R. Rao, S. Nassif, and R. B. Brown, "Power-Aware Global Signaling Strategies," *IEEE International Symposium on Circuits and Systems*, vol. 1, pp. 604–607, May 2005.

[11] J. Rabaey, A. Chandrakasan, and B. Nikolic', *Digital Integrated Circuits*. NJ: Prentice Hall, 2003.

[12] A. Youssef, M. Anis, and M. Elmasry, "POMR: A Power-Aware Interconnect Optimization Methodology," *IEEE Transactions on Very Large Scale Integration Systems*, vol. 13, no. 3, pp. 297–307, March 2005.

[13] A. Youssef, T. Myklebust, M. Anis, and M. Elmasry, "A Low-Power Multi-Pin Maze Routing Methodology," *Proceedings of the IEEE International Symposium on Quality Electronic Design*, pp. 153–158, 2007.

[14] Y. Peng and X. Liu, "Low-Power Repeater Insertion with both Delay and Slew Rate Constraints," *Proceedings of the Design Automation Conference*, pp. 302–307, 2006.

[15] K. H. Tam and L. He, "Power Optimal Dual-Vdd Buffered Tree Considering Buffer Stations and Blockages," *Proceedings of the Design Automation Conference*, pp. 497–502, 2005.

[16] X. Liu, Y. Peng, and M. C. Papaefthymiou, "Practical repeater insertion for low power: What repeater library do we need?" *Proceedings of the Design Automation Conference*, pp. 30–35, 2004.

[17] Y. Peng and X. Liu, "FREEZE: Engineering A Fast Repeater Insertion Solver for Power Minimization Using the Ellipsoid Method," *Proceedings of the Design Automation Conference*, pp. 813–818, 2005.

[18] V. Wason and K. Banerjee, "A Probabilistic Framework for Power-Optimal Repeater Insertion in Global Interconnects Under Parameter Variations," *Proceedings of the International Symposium on Low Power Electronics and Design*, pp. 131–136, 2005.

[19] W. T. Cheung and N. Wong, "Power optimization in a repeater-inserted interconnect via geometric programming," *Proceedings of the International Symposium on Low Power Electronics and Design*, pp. 226–231, 2006.

[20] P. Kapur, G. Chandra, and K. C. Saraswat, "Power Estimation in Global Interconnects and its Reduction Using a Novel Repeater Optimization Methodology," *Proceedings of the Design Automation Conference*, pp. 461–466, 2002.

[21] H. Fatemi, B. Amelifar, and M. Pedram, "Power Optimal MTCMOS Repeater Insertion for Global Buses," *Proceedings of the International Symposium on Low Power Electronics and Design*, pp. 98–103, 2007.

[22] K. Nose and T. Sakurai, "Power-Conscious Interconnect Buffer Optimization with Improved Modeling of Driver MOSFET and its Implications to Bulk And SOI CMOS Technology," *Proceedings of the International Symposium on Low Power Electronics and Design*, pp. 24–29, 2002.

[23] "International Technology Roadmap for Semiconductors," 2007. [Online]. Available: http://public.itrs.net/Files/2007ITRS/Home2007.htm.

[24] R. H. Dennard, F. H. Gaensslen, W.-N. Yu, V. L. Rideout, E. Bassous, and A. R. Leblance, "Design of Ion-Implanted MOSFET'S with Very Small Physical Dimensions," *IEEE Journal of Solid-State Circuits*, vol. 9, no. 5, pp. 256– 268, Sept. 1974.

[25] A. Chandrakasan, W. J. Bowhill, and F. Fox, *Design of High-Performance Microprocessor Circuits*. IEEE Press, 2000.

[26] S. Thompson, P. Packan, and M. Bohr, "MOS Scaling: Transistor Challenges for the 21st Century," *Intel Technology Journal*, p. 19, 1998.

[27] M. Ellsworth, "Chip Power Density and Module Cooling Technology Projections for the Current Decade," *Proceedings of the Intersociety Conference on Thermal and Thermomechanical Phenomena in Electronic Systems*, pp. 707–708, 2004.

[28] L. Wei, K. Roy, and K. D. Vivek, "Low Voltage Low Power CMOS Design Techniques for Deep Submicron ICs," *Proceedings of the International Conference on VLSI Design*, pp. 24–29, 2000.

[29] J. Kao, S. Narendra, and A. Chandrakasan, "Subthreshold Leakage Modeling and Reduction Techniques," *Proceedings of the International Conference on Computer-Aided Design*, pp. 141–148, 2002.

[30] K. Roy, S. Mukhopadhyay, and H. Mahmoodi-Meimand, "Leakage Current Mechanisms and Leakage Reduction Techniques in Deep-Submicrometer CMOS Circuits," *Proceedings of the IEEE*, vol. 91, no. 2, pp. 305–327, Feb. 2003.

[31] Y. Taur and T. H. Ning, *Fundamentals of Modern VLSI Devices*. NY: Cambridge University Press, 1998.

[32] D. Sima, "Decisive Aspects in the Evolution of Microprocessors," *Proceedings of the IEEE*, vol. 92, no. 12, pp. 1896–1926, Dec. 2004.

[33] J. Smith and G. Sohi, "The Microarchitecture of Superscalar Processors," *in Proceedings of the IEEE*, vol. 83, no. 12, pp. 1609–1624, Dec. 1995.

[34] K. C. Yeager, "The MIPS R10000 Superscalar Microprocessor," *IEEE Micro*, vol. 16, no. 2, pp. 28–41, April 1996.

[35] D. M. Tullsen, S. Eggers, and H. M. Levy, "Simultaneous Multithreading: Maximizing On-Chip Parallelism," *Proceedings 22th International Symposium on Computer Architecture*, pp. 392–403, 1995.

[36] P. Kongetira, K. Aingaran, and K. Olukotun, " Niagara: A 32-Way Multithreaded SPARC Processor," *IEEE Micro*, vol. 25, no. 2, pp. 21–29, April 2005.

[37] P. Babighian, L. Benini, A. Macii, and E. Macii, "Post-Layout Leakage Power Minimization Based on Distributed Sleep Transistor Insertion," *Proceedings of the 2004 International Symposium on Low Power Electronics and Design*, pp. 138–143, 2004.

[38] Y. Ye, S. Borkar, and V. De, "A New Technique for Standby Leakage Reduction in High-Performance Circuits," *Digest of Technical Papers of the Symposium on VLSI Circuits*, pp. 11–13, June 1998.

[39] J. Kao, M. Miyazaki, and A. Chandrakasan, "A 175-mV Multiply-Accumulate Unit Using an Adaptive Supply Voltage and Body Bias Architecture," *IEEE Journal of Solid-State Circuits*, vol. 37, no. 11, pp. 1545–1554, Nov. 2002.

[40] H. Ananthan, C. H. Kim, and K. Roy, "Larger-than-Vdd Forward Body Bias in Sub-0.5V Nanoscale CMOS," *Proceedings of the 2004 International Symposium on Low Power Electronics and Design*, pp. 8–13, 2004.

[41] M. Johnson, D. Somasekhar, and K. Roy, "Models and Algorithms for Bounds on Leakage in CMOS Circuits," *Transactions on Computer-Aided Design of Integrated Circuits and Systems*, vol. 18, no. 6, pp. 714–725, June 1999.

[42] M. Johnson, D. Somasekhar, L.-Y. Chiou, and K. Roy, "Leakage Control with Efficient Use of Transistor Stacks in Single Threshold CMOS," *Transactions on VLSI Systems*, vol. 10, no. 1, pp. 1–5, Feb. 2002.

[43] D. Duarte, Y.-F. Tsai, N. Vijaykrishnan, and M. J. Irwin, "Evaluating Run-Time Techniques for Leakage Power Reduction," *Proceedings of the Design Automation Conference*, pp. 31–38, Jan. 2002.

[44] A. Bhavnagarwala, B. Austin, K. Bowman, and J. Meindl, "A Minimum Total Power Methodology for Projecting Limits on CMOS GSI," *IEEE Transactions on Very Large Scale Integration Systems*, vol. 8, no. 3, pp. 235–251, June. 2000.

[45] S. Tyagi, M. Alavi, R. Bigwood, T. Bramblett, J. Brandenburg, W. Chen, M. H. B. Crew, P. Jacob, C. Kenyon, C. Lo, B. Mcintyre, Z. Ma, P. Moon, P. Nguyen, L. Rumaner, R. Schweinfurth, S. Sivakumar, M. Stettler, S. Thompson, B. Tufts, J. Xu, S. Yang, and M. Bohr, "A 130nm Generation Logic Technology Featuring 70nm Transistors, Dual Vt Transistors and 6 Layers Cu Interconnects," *International Electron Devices Meeting Technical Digest.*, pp. 567–570, 2000.

[46] M. Meijer, F. Pessolano, and J. P. de Gyvez, "Technology Exploration for Adaptive Power and Frequency Scaling in 90nm CMOS," *Proceedings of the 2004 International Symposium on Low Power Electronics and Design*, pp. 14–19, 2004.

[47] V. Gutnik and A. Chandrakasan, "Embedded Power Supply for Low-Power DSP," *Transactions on VLSI Systems*, vol. 5, no. 4, pp. 425–435, Dec. 1997.

[48] A. Keshavarzi, S. Narendra, S. Borkar, C. Hawkind, K. Roy, and V. De, "Technology Scaling Behavior of Optimum Reverse Body Bias for Standby Leakage Power Reduction in CMOS IC's," *International Symposium on Low Power Electronics and Design*, pp. 252–254, 1999.

[49] M. Anis and M. Elmasry, *Multi-Threshold CMOS Digital Circuits - Managing Leakage Power*. Norwell: Kluwer Academic Publishers, 2003.

[50] Z. Chen, L. Wei, and K. Roy, "Estimation of Standby Leakage Power in CMOS Circuits Considering Accurate Modeling of Transistor Stacks," *Proceedings of the International Symposium on Low-Power Electronics and Design*, pp. 239–244, 1998.

[51] J. Al-Eryani, "FPU100 at Opencores.org," 2007. [Online]. Available: http://www.opencores.org/projects/fpu100

[52] S. Shigematsu, S. Mutoh, Y. Matsuya, Y. Tanabe, and J. Yamada, "A 1-V High-Speed MTCMOS Circuit Scheme for Power-Down Application Circuits," *IEEE Journal of Solid-State Circuits*, vol. 32, no. 6, pp. 861–869, June 1997.

[53] S. Kim, S. Kosonocky, D. Knebel, and K. Stawiasz, "Experimental Measurement of a Novel Power Gating Structure with Intermediate Power Saving Mode," *Proceedings of the 2004 International Symposium on Low Power Electronics and Design*, pp. 20–25, 2004.

[54] Z. Zhu and X. Zhang, "Look-Ahead Architecture Adaptation to Reduce Processor Power Consumption," *IEEE Micro*, vol. 25, no. 4, pp. 10–19, July 2005.

[55] H. Li, S. Bhunia, Y. Chen, T. N. Vijaykumar, and K. Roy, "Deterministic Clock Gating for Microprocessor Power Reduction," *The International Symposium on High-Performance Computer Architecture*, pp. 113–122, 2003.

[56] H. Li, C. Cher, K. Roy, and T. N. Vijaykumar, "Combined Circuit and Architectural Level Variable Supply-Voltage Scaling for Low Power," *IEEE Transactions on Very Large Scale Integration Systems*, vol. 13, no. 5, pp. 564–576, May 2005.

[57] A. Baniasadi and A. Moshovos, "Branch Predictor Prediction: a Power-Aware Branch Predictor for High-Performance Processors," *Proceedings of the IEEE International Conference on Computer Design: VLSI in Computers and Processors*, pp. 458–461, 2002.

[58] D. Parikh, K. Skadron, Y. Zhang, and M. Stan, "Power-Aware Branch Prediction: Characterization and Design," *IEEE Transactions on Computers*, vol. 53, no. 2, pp. 168–186, 2004.

[59] K. Roy and S. Prasad, *Low-Power CMOS VLSI Circuit Design*. NY: Wiley-Interscience, 2000.

[60] T. Takayanagi, J. L. Shin, B. Petrick, J. Y. Su, H. Levy, H. Pham, J. Son, N. Moon, D. Bistry, U. Nair, M. Singh, V. Mathur, and A. S. Leon, "A Dual-Core 64-bit UltraSPARC Microprocessor for Dense Server Applications," *IEEE Journal of Solid-State Circuits*, vol. 40, no. 1, pp. 7–18, Jan 2005.

[61] C.-L. Su, C.-Y. Tsui, and A. M. Despain, "Low Power Architecture Design and Compilation Techniques for High-Performance Processors," in *Proceedings of the IEEE Computer Conference*, 1994, pp. 489–498.

[62] V. Tiwari, S. Malik, A. Wolfe, and M. T.-C. Lee, "Instruction Level Power Analysis and Optimization of Software," *Journal of VLSI Signal Processing*, vol. 13, no. 2-3, pp. 223–238, Aug. 1996.

[63] T. Sherwood, E. Perelman, and B. Calder, "Basic Block Distribution Analysis to Find Periodic Behavior and Simulation Points in Applications," *International Conference on Parallel Architectures and Compilation Techniques*, pp. 3–14, 2001.

[64] N. Pettis, L. Cai, and Y.-H. Lu, "Dynamic Power Management for Streaming Data," *Proceedings of the International Symposium on Low Power Electronics and Design*, pp. 62–65, 2004.

[65] S. Rele, S. Pande, S. Onder, and R. Gupta, "Optimizing Static Power Dissipation by Functional Units in Superscalar Processors," *Lecture Notes in Computer Science*, vol. 2304, pp. 261–276, 2002.

[66] H. S. Kim, N. Vijaykrishnan, M. Kandemir, and M. J. Irwin, "Adapting Instruction Level Parallelism for Optimizing Leakage in VLIW Architectures," *Proceedings of the ACM SIGPLAN conference on Languages, Compilers, and Tool support for Embedded Systems*, vol. 38, no. 7, pp. 275–283, 2003.

[67] S. Narayanasamy, T. Sherwood, S. Sair, B. Calder, and G. Varghese, "Catching Accurate Profiles in Hardware," *Proceedings of the International Symposium on High-Performance Computer Architecture*, pp. 269–280, 2003.

[68] J. Lau, S. Schoenmackers, and B. Calder, "Transition Phase Classification and Prediction," *Proceedings of the International Symposium on High-Performance Computer Architecture*, pp. 278–289, 2005.

[69] T. Sherwood, E. Perelman, G. Hamerly, S. Sair, and B. Calder, "Discovering and Exploiting Program Phases," *IEEE Micro*, vol. 23, no. 6, pp. 84–93, Nov.-Dec. 2003.

[70] T. Austin, E. Larson, and D. Ernst, "Simplescalar: An Infrastructure for Computer System Modeling," *IEEE Computer Magazine*, vol. 35, no. 2, pp. 59–67, Feb. 2002.

[71] "SPEC Benchmark Suite Release 1.3," *SPEC 2000*, 2000. [Online]. Available: http://www.spec.org

[72] M. R. Guthaus, J. S. Ringenberg, D. Ernst, T. M. Austin, T. Mudge, and R. B. Brown, "MiBench: A Free, Commercially Representative Embedded Benchmark Suite," *IEEE International Workshop on Workload Characteristics*, pp. 3–14, 2001.

[73] D. Brooks, P. Bose, V. Srinivasan, M. K. Gschwind, P. G. Emma, and M. G. Rosenfield, "New Methodology for Early-Stage, Microarchitecture-Level Power Performance Analysis of Microprocessors," *IBM Journal of Research and Development*, vol. 47, no. 5/6, pp. 653–670, Sept./Nov. 2003.

[74] C. Price, "MIPS IV Instruction Set, revision 3.1," *MIPS Technologies, Inc., Mountain View, CA*, January 1995.

[75] G. S. Sohi, "Instruction Issue Logic for High-Performance, Interruptible, Multiple Functional Unit, Pipelined Computers," *IEEE Transactions on Computers*, vol. 39, no. 3, pp. 349–359, March 1990.

[76] D. Brooks, V. Tiwari, and M. Martonosi, "Wattch: A Framework for Architectural-Level Power Analysis and Optimizations," *Proceedings of International Symposium on Computer Architecture*, pp. 83–94, 2000.

[77] M. Elgamel, S. Goel, and M. Bayoumi, "Noise Tolerant Low Voltage XOR-XNOR for Fast Arithmetic," *Proceedings of the ACM Great Lakes symposium on VLSI*, pp. 285–288, 2003.

[78] N. Azizi, M. Khellah, V. De, and F. N. Najm, "Variations-Aware Low-Power Design With Voltage Scaling," *Proceedings of the Design Automation Conference*, pp. 529–534, 2005.

[79] R. Ho, K. W. Mai, and M. A. Horowitz, "The Future of Wires," *in Proceedings of the IEEE*, vol. 89, no. 4, pp. 490–504, April 2001.

[80] M. Hrishikesh, N. Jouppi, K. Farkas, D. Burger, S. Keckler, and P. Shivakumar, "The Optimal Logic Depth Per Pipeline Stage is 6 to 8 FO4 Inverter Delays," *Proceedings of the International Symposium on Computer Architecture*, pp. 14–24, 2002.

[81] S. Heo and K. AsanoviC, "Power-Optimal Pipelining in Deep Submicron Technology," *Proceedings of the International Symposium On Low Power Electronics And Design*, pp. 218–223, 2004.

[82] M.Anis, S.Areibi, and M.Elmasry, "Design and Optimization of Multi-Threshold CMOS (MTCMOS) Circuits," *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, vol. 22, no. 10, pp. 1324–1242, Oct. 2003.

[83] A. K. Osowski and D. J. Lilja, "MinneSPEC: A New SPEC Benchmark Workload for Simulation-Based Computer Architecture Research," *Computer Architecture Letters*, vol. 1, pp. 7–10, 2002.

[84] D. Tullsen, "Simulation and Modeling of a Simultaneous Multithreading Processor," *22nd Annual Computer Measurement Group Conference*, pp. 819–828, 1996.

[85] S. Choi and D. Yeung, "Learning-Based SMT Processor Resource Distribution via Hill-Climbing," *Proceedings of the International Symposium on Computer Architecture*, pp. 239–251, 2006.

[86] J. Chang and G. Sohi, "Cooperative Caching for Chip Multiprocessors," *Proceedings of the International Symposium on Computer Architecture*, pp. 264 – 276, 2006.

[87] Z. Chishti, M. D. Powell, and T. N. Vijaykumar, "Optimizing Replication, Communication, and Capacity Allocation in CMPs," *Proceedings of the International Symposium on Computer Architecture*, pp. 357–368, 2005.

[88] I. M. Liu, A. Aziz, and D. E. Wong, "Meeting Delay Constraints in DSM by Minimal Repeater Insertion," *Proceedings of the Conference on Design, Automation and Test in Europe*, pp. 436–440, 2000.

[89] M. Cho, K. Lu, K. Yuan, and D. Z. Pan, "BoxRouter 2.0: Architecture and Implementation of a Hybrid and Robust Global Router," *Proceedings of the International Conference on Computer-Aided Design*, pp. 503–508, 2007.

[90] R. T. Hadsell and P. H. Madden, "Improved Global Routing through Congestion Estimation," *Proceedings of the Design Automation Conference*, pp. 28 – 31, 2003.

[91] M. Hrkic and J. Lillis, "S-Tree: a technique for buffered routing tree synthesis," *Proceedings of the Design Automation Conference*, pp. 578–583, 2002.

[92] Y. Peng and X. Liu, "Low-Power Repeater Insertion with both Delay and Slew Rate Constraints," *Proceedings of the Design Automation Conference*, pp. 302–307, 2006.

[93] K. H. Tam and L. He, "Power Optimal Dual-Vdd Buffered Tree Considering Buffer Stations and Blockages," *Proceedings of the Design Automation Conference*, pp. 497–502, 2005.

[94] C. Chiang, C. K. Wong, and M. Sarrafzadeh, "A Weighted Steiner Tree-based Global Router with Simultaneous Length and Density Minimization," *IEEE*

Transactions on Computer-Aided Design, vol. 13, no. 12, pp. 1461–1469, Dec. 1994.

[95] W. A. Dees and P. G. Karger, "Automated rip-up and reroute techniques," *Proceedings of ACM/IEEE Design Automation Conference*, pp. 432–439, 1982.

[96] B. S. Ting and B. N. Tien, "Routing Technique for Gate Array," *IEEE Transactions on Computer-Aided Design*, vol. 2, no. 4, pp. 301–312, Oct. 1983.

[97] C. Albercht, "Global Routing by New Approximation Algorithms for Multicommodity Flow," *IEEE Transactions on Computer-Aided Design*, vol. 20, no. 5, pp. 622–632, May 2001.

[98] R. Carden, J. Li, and C. Cheng, "A Global Router with a Theoretical Bound on the Optimal Solution," *IEEE Transactions on Computer-Aided Design*, vol. 15, no. 2, pp. 208–216, February 1996.

[99] J. Cong, J. Fang, and Y. Zhang, "MARS-A Multilevel Full-Chip Gridless Routing System," *IEEE Transactions Computer-Aided Design*, vol. 24, no. 3, pp. 382–394, March 2005.

[100] J. Huang, E. S. Kuh, C.Cheng, and X. Hong, "An Efficient Timing-Driven Global Routing Algorithm," *Proceedings of the Design Automation Conference*, pp. 596–600, 1993.

[101] D. Wang and E. Kuh, "Performance-Driven Interconnect Global Routing," *Proceedings of the Great Lakes Symposium on VLSI*, pp. 132–136, 1996.

[102] J. Cong and P. Madden, "Performance Driven Global Routing for Standard Cell Design," *Proceedings of the ACM International Symposium on Physical Design*, pp. 73–80, 1997.

[103] K. Zhu, Y. W. Chang, and D. F. Wong, "Timing-Driven Routing for Symmetrical-Array-Based FPGAs," *Proceedings of the International Conference on Computer Design*, pp. 628–633, 1998.

[104] J. Hu and S. Sapatnekar, "A Timing-Constrained Algorithm for Simultaneous Global Routing of Multiple Nets," *Proceedings of the International Conference on Computer-Aided Design*, pp. 99 –103, 2000.

[105] T. Jing, X. L. Hong, J. Y. Xu, , C. K. C. heng, and J. Gu, "UTACO: A Unified Timing and Congestion Optimization Algorithm for Standard Cell Global Routing," *IEEE Transactions Computer-Aided Design*, vol. 23, pp. 358–365, 2004.

[106] I.-M. Liu, A. Aziz, D. F. Wong, and H. Zhou, "An Efficient Buffer Insertion Algorithm for Large Networks Based on Lagrangian Relaxation," *Proceedings of the International Conference on Computer Design*, pp. 210–215, 1999.

[107] R. Chen and H. Zhou, "Efficient Algorithms for Buffer Insertion in General Circuits Based on Network Flow," *Proceedings of the International Conference on Computer-Aided Design*, pp. 509–514, 2005.

[108] M. Waghmode, Z. Li, and W. Shi, "Buffer insertion in large circuits with constructive solution search techniques," *Proceedings of the Design Automation Conference*, pp. 296–301, 2006.

[109] C. N. Sze, C. J. Alpert, J. Hu, and W. Shi, "Path based buffer insertion," *Proceedings of the Design Automation Conference*, pp. 509–514, 2005.

[110] A. P. Chandrakasan and R. W. Brodersen, *Low Power Digital CMOS Design*. Boston: Kluwer Academic Publishers, 1995.

[111] C. Chu and D. Wong, "Closed Form Solution to Simultaneous Buffer Insertion/Sizing and Wire Sizing," *ACM Transactions on Design Automation of Electronic Systems*, vol. 6, no. 3, pp. 343–371, July 2001.

[112] R. Otten and G. S. Garcea, "Simultaneous Analytic Area and Power Optimization for Repeater Insertion," *Proceedings of the International Conference on Computer-Aided Design*, pp. 568–573, 2003.

[113] R. Li, D. Zhou, J. Liu, and X. Zeng, "Power-Optimal Simultaneous Buffer Insertion/Sizing and Wire Sizing," *Proceedings of the International Conference on Computer-Aided Design*, pp. 581–586, 2003.

[114] A. Nalamalpu and W. Burleson, "A Practical Approach to DSM Repeater Insertion: Satisfying Delay Constraints while Minimizing Area and Power," *Proceedings of the IEEE International ASIC/SOC Conference*, pp. 152–156, Sept. 2001.

[115] S. Turgis, N. Azemard, and D. Auvergne, "Design and Selection of Buffers for Minimum Power-Delay Product," *Proceedings of the The European Design and Test Conference*, pp. 224–228, 1996.

[116] J. C. Eble, V. K. De, D. S. Wills, and J. D. Meindl, "Minimum Repeater Count, Size, and Energy Dissipation for Gigascale Integration (GSI) Interconnects," *Proceedings of the International Interconnect Technology Conference*, pp. 56–58, 1998.

[117] C. Chu and D. Wong, "An Efficient and Optimal Algorithm for Simultaneous Buffer and Wire Sizing," *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, vol. 18, no. 9, pp. 1297–1304, Sept. 1999.

[118] L. P. P. P. van Ginneken, "Buffer Placement in Distributed RC-Tree Networks for Minimal Elmore Delay," *Proceedings of the International Symposium on Circuits and Systems*, pp. 865–868, 1990.

[119] J. Cong, C. Koh, and K. Leung, "Simultaneous Buffer and Wire Sizing for Performance and Power Optimization," *Proceedings of the International Symposium on Low Power Electronics and Design*, pp. 271–276, 1996.

[120] J. Lillis, C. Cheng, and T. Lin, "Simultaneous routing and buffer insertion for high performance interconnect," *Proceedings of Great Lakes Symposium on VLSI*, pp. 148–153, 1996.

[121] J. C. Shah and S. S. Sapatnekar, "Wiresizing with buffer placement and sizing for power-delay tradeoffs," *Proceedings of VLSI Design*, pp. 346–351, 1996.

[122] M. Lai and D. Wong, "Maze Routing with Buffer Insertion and Wiresizing," *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, vol. 21, no. 10, pp. 1205–1209, Oct. 2002.

[123] J. Cong and Z. Pan, "Interconnect Performance Estimation Models for Design Planning," *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, vol. 20, no. 6, pp. 739–752, June 2001.

[124] N. Sherwani, *Algorithms for VLSI Physical Design Automation*. Boston, MA: Kluwer Academic, 1999.

[125] D. M. Warme, "A New Exact Algorithm for Rectilinear Steiner Trees," *International Symposium on Mathematical Programming*, 1997.

[126] W. C. Elmore, "The Transient Response of Damped Linear Networks," *Journal of Applied Physics*, vol. 19, pp. 55–63, Jan. 1948.

[127] S. C. Wong, G.-Y. Lee, and D.-J. Ma, "Modeling of Interconnect Capacitance, Delay, and Crosstalk in VLSI," *IEEE Transactions on Semiconductor Manufacturing*, vol. 13, no. 1, pp. 108–111, Feb. 2000.

[128] "BPTM Provided by the Device Group at UC Berkeley,." [Online]. Available: http://www-device.eecs.berkeley.edu/~ptm/introduction.html

[129] X. Tang, R. Tian, H. Xiang, and D. F. Wong, "A New Algorithm for Routing Tree Construction with Buffer Insertion and Wire Sizing under Obstacle Constraints," *Proceedings of the International Conference on Computer-Aided Design*, pp. 49–56, 2001.

[130] Z. Yang, S. Areibi, and A. Vannelli, "An ILP Based Hierarchical Global Routing Approach for VLSI ASIC Design," *Optimization Letters*, vol. 1, pp. 281–297, June 2007.

[131] C. Chu, " FLUTE: Fast Lookup Table Based Wirelength Estimation Technique," *Proceedings of the International Conference on Computer Aided Design*, pp. 696–701, 2004.

[132] L. Behjat, A. Vannelli, and A. Kennings, " Congestion Based Mathematical Programming Models for Global Routing," *Proceedings of the Midwest Symposium on Circuits and Systems*, pp. 599–602, 2002.

[133] ISPD98/IBM, "www.ece.ucsb.edu/ kastner/labyrinth/benchmarks/," 1998.

[134] ISPD2007, "http://www.ispd.cc/ispd07_contest.html," 2007.

[135] Y. Peng and X. Liu, "FREEZE: Engineering a Fast Repeater Insertion Solver for Power Minimization Using the Ellipsoid Method," *Proceedings of the Design Automation Conference*, pp. 813–818, 2005.

[136] W. T. Cheung and N. Wong, "Power Optimization in A Repeater-Inserted Interconnect Via Geometric Programming," *Proceedings of The International Symposium on Low Power Electronics and Design*, pp. 226–231, 2006.

[137] Y. P. X. Liu and M. Papaefthymiou, "Practical Repeater Insertion for Low Power: What Repeater Library Do We Need?" *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, vol. 25, no. 5, pp. 917–924, May 2006.