Rochester Institute of Technology
RIT Digital Institutional Repository

Theses

5-1-2009

Dynamic voltage and frequency scaling with multi-clock distribution systems on SPARC core

Michael Nasri Michael

Follow this and additional works at: https://repository.rit.edu/theses

Recommended Citation
Michael, Michael Nasri, "Dynamic voltage and frequency scaling with multi-clock distribution systems on SPARC core" (2009). Thesis. Rochester Institute of Technology. Accessed from

This Thesis is brought to you for free and open access by the RIT Libraries. For more information, please contact repository@rit.edu.

Dynamic Voltage and Frequency Scaling with Multi-Clock Distribution Systems on SPARC Core

by

Michael Nasri Michael

A Thesis Submitted in Partial Fulfillment of the Requirements for the Degree of Master of Science in Computer Engineering

Supervised by

Dr. Dhireesha Kudithipudi
Department of Computer Engineering
Kate Gleason College of Engineering
Rochester Institute of Technology
Rochester, NY
May 2009

Approved By:

Dr. Dhireesha Kudithipudi
Assistant Professor, RIT Department of Computer Engineering
Primary Advisor

Dr. Ken Hsu
Professor, RIT Department of Computer Engineering

Dr. Muhammad Shaaban
Associate Professor, RIT Department of Computer Engineering

Thesis Release Permission Form

Rochester Institute of Technology

Kate Gleason College of Engineering

Title:
Dynamic Voltage and Frequency Scaling with Multi-Clock Distribution Systems on SPARC Core

I, Michael Nasri Michael, hereby grant permission to the Wallace Memorial Library to reproduce my thesis in whole or part.

Michael Nasri Michael

Date

Dedication

To my parents for their pride and encouragement

To my brother and sisters for their love and support

Acknowledgements

Thanks to Dr. Dhireesha Kudithipudi for her constant advice and patience

Abstract

The current implementation of dynamic voltage and frequency scaling (DVS and DFS) in microprocessors is based on a single clock domain per core. In architectures that adopt Instruction Level Parallelism (ILP) , multiple execution units may exist and operate concurrently. Performing DVS and DFS on such cores may result in low utilization and power efficiency.

In this thesis, a methodology that implements DVFS with multi Clock distribution Systems (DCS) is applied on a processor core to achieve higher throughput and better power efficiency. DCS replaces the core single clock distribution tree with multi-clock domain systems which, along with dynamic voltage and frequency scaling, creates multiple clock-voltage domains. DCS implements a self-timed interface between the different domains to maintain functionality and ensure data integrity.

DCS was implemented on a SPARC core of UltraSPARC T1 architecture, and synthesized targeting TSMC 120nm process technology. Two clock domains were used on SPARC core. The maximum achieved speedup relative to original core was 1.6X. The power consumed by DCS was 0.173mW compared to the core total power of $\sim$ 10W.

Contents

List of Figures

List of Tables

Glossary

Abbreviation	Description	Definition
PLL	Phased Locked Loop	page 34
DVFS	Dynamic Voltage and Frequency Scaling	page 9
DFD	Digital Frequency Divider	page 34
CMOS	Complementary Metal Oxide Semiconductors	page 10
RBB	Reverse Body Bias	page 5
CPU	Central Processing Unit	page 12
I/O	Input Output	page 7
DBS	Demand Based Switching	page 9
OS	Operating System	page 8
BIOS	Basic Input Output System	page 7
CMP	Chip Multi-Processing	page 12
FTC	Foxton Technology microController	page 12
VID	Voltage IDentification	page 10
A/D	Analog to Digital	page 11
BJT	Bipolar Junction Transistor	page 10
IR	Current Resistance product	page 12
VCO	Voltage Controlled Oscillator	page 12
BIST	Built-In-Self-Test	page 12
TLB	Translation Lookaside Buffer	page 16
FPU	Floating Point Unit	page 16
ILP	Instruction Level Parallelism	page iii
IFU	Instruction Fetch Unit	page 16
EXU	EXecution Unit	page 37

Abbreviation	Description	Definition
ALU	Arithmetic and Logic Unit	page 17
SPU	Stream Processing Unit	page 17
LSU	Load and Store Unit	page 16
TLU	Trap Logic Unit	page 16
PC	Program Counter	page 16
MAU	Modular Arithmetic Unit	page 16
MMU	Memory Management Unit	page 16
ITLB	Instruction TLB	page 16
DTLB	Data TLB	page 16
FFU	Floating-point Frontend Unit	page 16
FRF	Floating-point Register File	page 16
ECL	Execution Control Logic	page 17
IRF	Integer Register File	page 17

Chapter 1

Introduction

1.1 Thesis Objectives

Due to the improvements in die size and process technology, chip power density has increased drastically over the years [1] [7] [8]. On-chip thermal and power management has been developed with adaptive designs that monitor temperature and power consumption, and dynamically optimize the operating point for maximal power efficiency in real time [4] [9]. This thesis provides an in depth look at the latest dynamic power management techniques implemented in different architectures. The objectives and contributions of this thesis are outlined as follows:

- Design a methodology that adopts existing thermal and power management techniques and enhance the power-speedup performance.
- Implement the methodology on a general purpose processor as a test environment for simulation and functionality verification.
- Identify implementation power overhead imposed on the system and estimate the power-saving and speedup gain.

1.2 Thesis Chapter Overview

This thesis document starts with an introduction of power optimization techniques at different levels of abstraction in chapter 2, followed by an overview of OpenSPARC architecture that is used as a testbed for this thesis in chapter 3. Chapter 4 presents DCS implementation and its interaction with SPARC core. Simulation results and performance analysis are covered in chapter 5. Finally, chapter 6 concludes this thesis with a discussion of the results, and a look into possible future work and improvements.

Chapter 2

Thermal and Power Management Background

In this chapter, an introduction on power management is covered in section 2.1, followed by an overview of power management techniques applied in modern systems in section 2.2. Foxton Technology Microcontroller that implements DFVS in microprocessors is covered in section 2.3, followed by related work in section 2.4.

2.1 History

In 1965, Gordon Moore has introduced an analysis of the transistor's manufacturing cost with respect to the number of transistors. In addition, Moore's study observed the growth of the number of transistors on an IC over the years, and predicted that the number of transistors will double every 18 months [10] [8]. The industry managed to maintain the transistor density growth for over four decades.

The increase in the number of transistors on an IC is due to the reduced transistor size and increased die size. The transistor size has scaled down by 30% since 1987, resulting in a decrease of the chip area by 50% and a decrease of the total parasitic capacitance by 30% for the same number of transistors. The gate delay was reduced by 30% allowing an increase in maximum clock frequency by 43% from 10s of megahertz to over 3 gigahertz. The die size has scaled up by 25% per technology generation, as shown in *figure 2.1* [1]

[11].

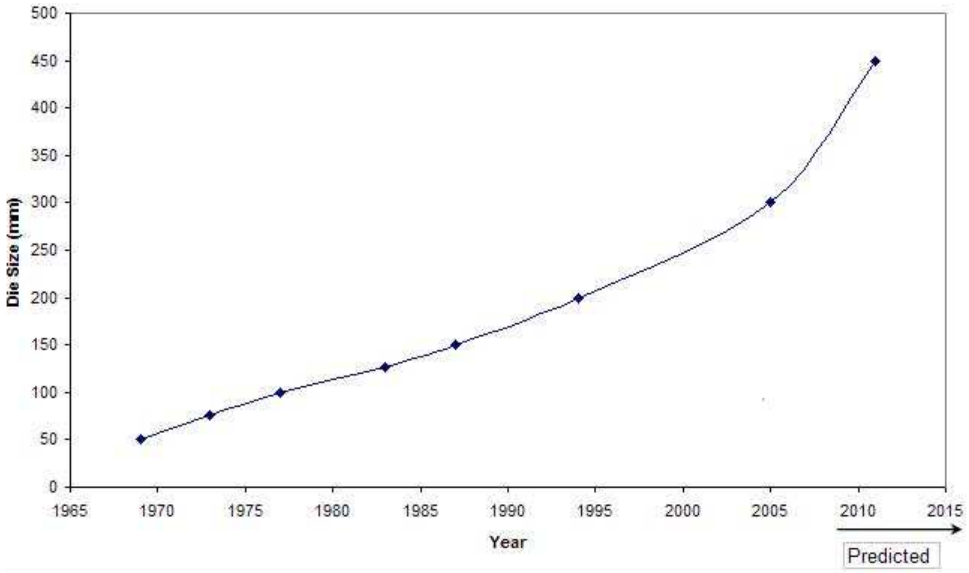

Figure 2.1: Die Size Growth Trend since 1969 [1].

As a result, the microprocessor power density has grown exponentially and the IC package is unable to dissipate the excessive heat. It is evident that the power growth is reaching certain limits at which the ICs can't function properly [12], and more efficient power management solutions were developed.

As for thermal management, die temeprature was sustained in the first few microprocessor generations by mounting a heatsink and a fan on the package to dissipate the excessive heat generated from operating at high frequencies. However, this approach was insufficient with the advances made in process technology and die sizes, where the transistor density has increased to the extend where the generated temperature was higher than what the heatsink and the fan can dissipate. Innovative designs were developed that integrate thermal management with power management, and temperature measurement can no longer be isolated from power management.

In addition, chip packaging contributes significantly in heat dissipation. Package with greater thermal conductivity has better heat dissipation efficiency. However, as discussed before, all these thermal management innovations are insufficient compared to the growth in power consumption.

In this chapter, several power management techniques are introduced in section 2.2.

Two of the latest dynamic power management techniques are discussed in section 2.2.3. Finally, section 2.3 presents an overview of an on-die microcontroller implementation, which is a significant hardware resource for dynamic power management.

2.2 Power Optimization Techniques

The power of an IC for a fixed temperature and operating voltage increases linearly with the clock frequency. Reducing the frequency to zero yields a non-zero power, which is referred to as the static or leakage power. The dynamic power is due to the charging and discharging of capacitances in the IC, and can be represented via the relationship

$$P_{dynamic} = \alpha.C.V_{DD}^2.f \tag{2.1}$$

In technology generations of sizes 0.18um and larger, the static or leakage power is negligible compared to dynamic power. As the size is reduced, the static power becomes significant component of the total power, due to the increase in the sub-threshold leakage (resulting from smaller threshold voltage).

Techniques are developed at different levels of abstraction to reduce both static and dynamic power. The following sections will cover these techniques.

2.2.1 Circuit Level

Several techniques may be applied to optimize static power, such as Reverse Body Bias (RBB) and Stack Effect. RBB is biased on that the sub-threshold leakage decreases if the body of the transistor is based to a negative voltage with respect to the source of the transistor (V_{SB} <0) [13][14]. Stack Effect places a transistor in the pull-up or pull-down network without affecting the input load. Stacking two OFF-transistors will reduce the sub-threshold leakage compared to a single OFF-transistor [15].

On the other hand, dynamic power can be optimized based on the CMOS logic family used for the design, such as Mirror Circuit, Pseudo-nMOS and dynamic CMOS [16][17].

2.2.2 Logic Level

A number of techniques at logic level are discussed in the following subsections.

Asynchronous Circuits

Asynchronous circuits are circuits that are autonomous and not governed by system clock. The advantages of asynchronous circuitry are higher execution speeds and lower power dissipation due to the fact that no transistor ever transitions unless it is performing useful computation (the power consumptions is data-dependent). However, several challenges are faced when designing asynchronous systems; most CPU design tools are based on a clocked CPU. Modifying design tools to handle asynchronous systems requires additional testing to ensure the design avoids metastability problems [18] [19].

Clock Gating

Clock gating saves power when data activity is low by disabling the portions of the circuitry where flip-flops do not change state. ”Perfect Clock Gating” is used when various clock gating techniques are approximations of the data-dependent behavior exhibited by asynchronous circuit. The power consumption approaches that of an asynchronous circuit [20] [21].

2.2.3 System Level

The power at the circuit level is optimized by reducing both static and dynamic power of the entire circuit, while at the logic level is by eliminating redundant circuitry to achieve greater power efficiency. On the other hand, power optimization at the system level differs from that of circuit or logic levels. The objective is to turn OFF the inactive modules of the system. In microprocessors, the modules fall under computation, communication, or storage units [22]. The following subsections discuss several techniques at the system level.

I/O Optimization

I/O optimization aims towards reducing the communication occurrences between the microprocessor and I/O devices, and eliminating redundant computation. In display devices, for example, a small memory is implemented as a buffer or cache, thus reducing the occurence of memory accesses. Also in I/O devices, such as Universal Serial Bus (USB), the device and its chipset can be put to sleep when idle. [23].

Demand Based Switching (DBS)

In any process technology, the transistor can operate at lower voltage supplies if the operating frequency is reduced, which saves both dynamic and static power. In a regular microprocessor, the system operates at a fixed frequency and voltage supply, regardless of its workload. DBS allows the microprocessor to operate at multiple frequency/voltage settings, as shown in *figure 2.2*.

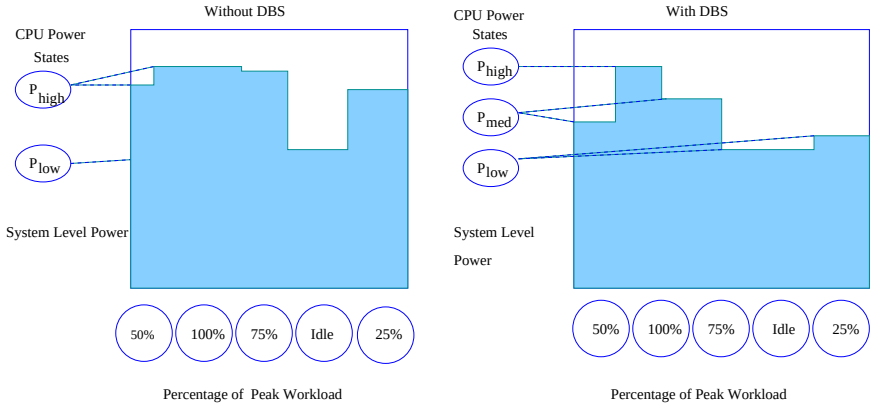

Figure 2.2: Comparison of Percentage of Peak Workload with and without DBS [2].

The OS and BIOS has to support DBS. The microprocessor automatically operates at the lowest setting consistent with optimal application performance. The OS monitors processor utilization and switches frequency and voltage accordingly to maintain optimal performance. Power is tailored to match the system workload.

DBS was first introduced by Intel, it reduced average system power consumption and cooling cost up to 25% with minimal performance penalty, based on ``computational´´ needs

[2].

Dynamic Voltage and Frequency Scaling (DVFS)

DVFS is an on-die technology implemented on Core Duo processor to maximize performance and minimize power consumption (static and dynamic). DVFS was developed as a result of implementing a new process technology and doubling the number of cores, allowing each core to control its dynamic power consumption [3].

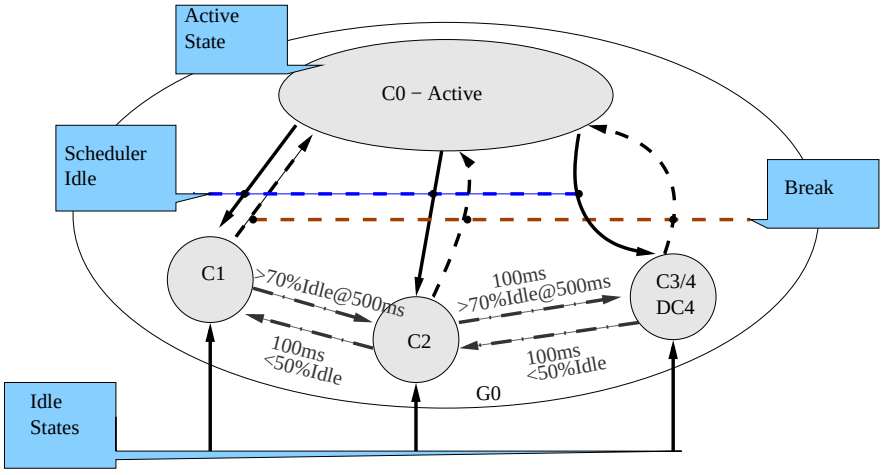

Figure 2.3: Chip Multi-Processing Idle State Management [3].

Without DVFS, controlling the static power is problematic, since the static power depends on the total area and the process technology. DVFS manages static power by putting the core in different power-saving states, as shown in *figure 2.3*. These states are termed sleep states (C-states) and marked as C_0 to C_n. The switching is performed by a combination of software (OS) and hardware elements. The increase in static power savings per state is achieved by employing more aggressive means as the C-state deepens, as the following shows:

- C_0 state is when the core is active. When idle, the OS tries to maintain a balance between the amount of power it can save and the overload of entering and exiting to/from that sleep state.

- C_1 state has the least power saving but can switch ON and OFF almost immediately. Only processor-centric measures are employed: instruction execution is halted and the core clocks are gated.

- In C_2 and above, platform-level measures are included: the processor doesn't access the bus without chipset consent. The front side bus is placed in a lower power state, and the chipset initiates power-saving measures.

- In C_3, the processor's internal Phase Locked Loops are disabled.

- In C_4, the processor's internal voltage is lowered to the point where only content retention is possible, but no operations can be performed.

- In Deep C_4 state (DC_4), voltage has to be further lowered to reduce the static power. Unfortunately, lower voltage impacts data retention, and small transistor data arrays such as the L2 cache are affected. In order to enter DC_4, a mechanism is initiated that can dynamically shutdown the L2 cache which allows a lower internal voltage in the processor.

DVFS handles dynamic power by changing the frequency/voltage setting in each state, same as in DBS. In deep sleep states, the frequency/voltage setting is reduced since the core has lower ``computational´´ needs, which reduces the dynamic power.

2.3 Foxton Technology Microcontroller (FTC)

The reliance on CMOS technology in digital systems has forced designers to develop more efficient techniques at the system level to cover different aspects of processor's power consumption, as discussed in section 2.2.3. However, the implementation of the techniques at the system level require significant amount of hardware resources, thus introducing a relatively large overhead. Moving power managemnet techniques, such as DBS or CMP, to firmware can significantly reduces the overhead and the maintenance cost. Implementing an on-die microcontroller provides a valuable hardware resource for dynamic power management, and a suitable environment for firmware implementation of the power management techniques.

The use of a microcontroller might raise some skepticism, as it is a great power and area overhead. However, analysis and implementation have proven otherwise [24]. For example, Intel's microcontroller (Foxton Technology Microcontroller or FTC) is developed for Montecito processor (of Itanium family), and implemented on the same die of the processor. Montecito is a dual-core, dual thread processor which operates at 1.6 GHz speed and built on 90 nm technology. The processor has a power envelope of 100W. The microcontroller consumes 0.5W (0.5٪ of the total power) and only 0.5٪ of the total area. The performance reduction is no more than 10٪. The estimated processor power excluding the microcontroller is ~ 300W [25].

FTC is implemented based on the voltage-frequency relation with power. Montecito processor has thermal and power sensors. FTC acquires the sensors' readings; on which FTC determines the voltage operation ID (VID) and maintains a certain power envelope [4].

The following subsections will cover the different components of the power management system that interface with FTC.

2.3.1 Thermal Measurement

Temperature sensors are one of the main inputs to FTC microcontroller. Die temperature can be measured using different methods [4]. The variation in measurement methods is due to accuracy, speed and resolution requirements for temperature readings [26].

The most accurate method is implementing an on-die temperature sensor. The thermal sensor consists of a current source, which drives a diode with a constant current, as shown in *figure 2.4*. The diode's output drives the input of an A/D converter. The measured voltage is converted into a temperature value by using a calibrated V_{BE} value and the characterized temperature coefficient (˜-1.7 mV/^{o}C). V_{BE} is determined after manufacturing by heating the die to a known temperature and recording the thermal voltages in on-die fuses [24].

On-die sensor is an expensive solution; since BJT transistors are being implemented on CMOS technology, and the sensors require post-silicon calibration [26]. Ring Oscil-

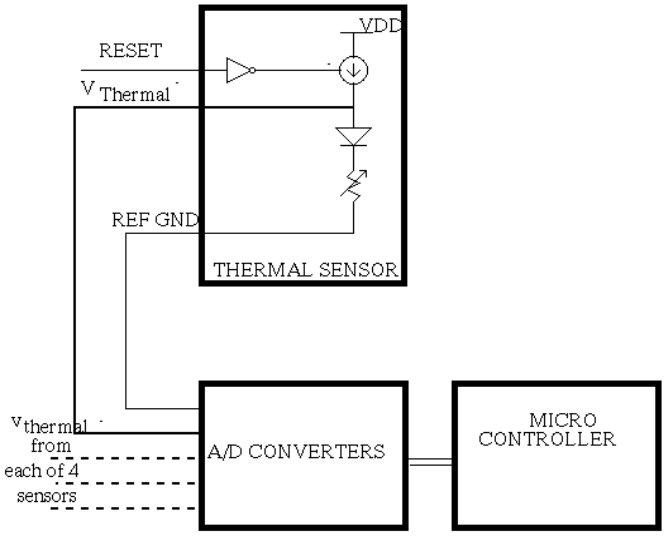

Figure 2.4: Thermal System Block Diagram [4].

lator is an alternative solution that utilizes existing hardware and can be easily implemented within the component's layout. A ring oscillator is a counter that consists of an odd number of inverters connected in series, whose switching frequency is proportional to the die temperature in the oscillator area. The value that the counter holds reflects the temperature that the oscillator resides [27][28].

Temperature can also be measured without the use of sensors. Measuring the power consumption can provide a rough estimate of the die temperature (power measurement is discussed in the following subsection). Power can also be estimated by monitoring the traffic (this method is used in estimating the power consumption of the memory) [29].

2.3.2 Power Measurement

Power measurement is essential to thermal management; it is used to evaluate the temperature of some components such as memory, and maintain a certain power envelope. Power measurement can be achieved by measuring both voltage and current. Voltage measurement is performed using A/D converters. Each converter is composed

of an analog multiplexer on the front end to select the input, a Voltage Controlled Oscillator (VCO), and a counter [24]. Current measurement requires additional precision series resistors; which requires additional power. However, exploiting the package parasitic as a resistor is an ideal alternative to adding resistors. Power can be calculated by knowing the package resistance and measuring the current-resistance (IR) drop from the edge connector to the center of the die [24]. Core power can be represented by

$$P_{Core} = V_{Die}I_{Die} = V_{Die}(V_{Connector} - V_{die})/R_{Package} \tag{2.2}$$

2.3.3 Microcontroller

FTC is an on-die custom microcontroller located between the two CPU cores. FTC operates at fixed clock rate of 900 MHz, and utilizes a fully bypassed five-stage pipeline. The microcontroller uses a Harvard architecture with program and data memory spaces, and supports 50 instructions of classes: Arithmetic, load/store, branches, and procedure calls. The instructions are 16 bit fixed length. For real-time control support, a timer block is implemented with programmable time bases. BIST engine is implemented for self-test of FTC's embedded memories, and a programmable trigger block for debug support [24].

2.4 Related Work

2.4.1 ThresHot

In chip Multi-Processing (CMP) , there exists temporal and spatial temperature variations which introduces the side-effect of thermal cycling and reduce the lifetime of the chip. ThresHot improves performance by dynamically scheduling jobs, aiming towards reducing thermal emergencies, which results in improving both job throughput and the chip reliability by reducing the thermal cycling effect [30].

2.5 Conclusions

With current process technology trends, die heating and power consumption have become a major concern to designers, and several power management techniques had to implemented, as discussed in this chapter. Techniques, such as DBS, DVFS and FTC, which are implemented in commercial microprocessors based on x86 architecture, have been proven to enhance power-throughput performance. UltraSPARC, which is the architecture used as a test environment for this thesis, implements a number of these methods, such as clock gating and thermal measurement to manage its power. In the following chapters, an enhanced power management technique will be introduced based on the techniques covered in this chapter.

Chapter 3

Test Environment

3.1 Introduction

This chapter will discuss the test environment, UltraSPARC T1 architecture, that is used in this research in section 3.2, followed by an overview of SPARC core. S1 core project is presented in section 3.2.3, followed by Wishbone bridge which is used in S1 core project in section 3.2.3.

3.2 UltraSPARC T1

3.2.1 Introduction

UltraSPARC T1 is a SUN microprocessor that implements the 64-bit SPARC V9 architecture. UltraSPARC T1, codename ¨Niagara¨, targets commercial applications such as application servers and database servers. The processor is designed to meet the demands for high throughput performance for servers; which is achieved by implementing multiple cores and hardware multi-threading, in addition to lowering power consumption resulting in a significant performance per Watt improvement with a very small investment in area, and avoiding the need for complex high frequency design techniques [31].

3.2.2 SPARC Core

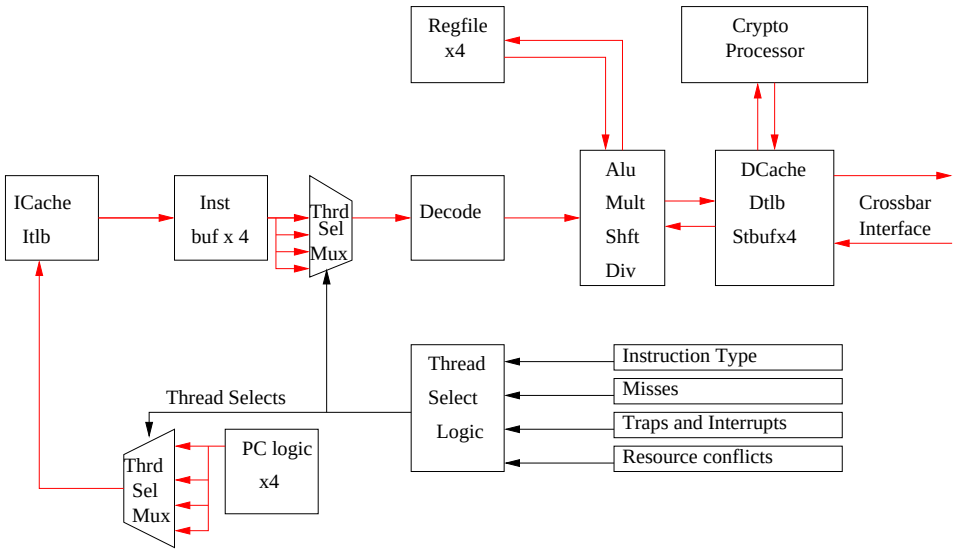

Figure 3.1: SPARC Core Block Diagram [5].

UltraSPARC T1 contains eight SPARC processor cores; each has full hardware support for four threads. A full register file is implemented per thread. The four threads share the instruction, the data caches, and the TLBs [32]. The core has single issue, six stage pipeline, as shown in *figure 3.1*. The six stages are:

1. Fetch
2. Thread Selection
3. Decode
4. Execute
5. Memory
6. Write Back

Each core consists of the following units:

Instruction Fetch Unit (IFU) consists of the following pipeline stages fetch, thread selection, and decode. IFU also includes an instruction cache complex. Two instructions are fetched each cycle, but only one is issued per clock, reducing the instruction cache

activity. There is only one outstanding miss per thread, and only four per core. Duplicate misses do not issue requests to the L2 cache.

Load/Store Unit (LSU) consists of memory and writeback stages, and a data cache complex. The data cache has an 8KB data, 4-way, 16B line size, and a single ported data tag. A dual-ported valid bit array is implemented to hold cache line state of valid or invalid. A pseudo-random replacement algorithm is used. The loads are allocating, and the stores are non-allocating. The data TLB operates similarly to the instruction TLB.

Trap Logic Unit (TLU) consists of trap logic and trap program counters. TLU has support for six trap levels. Traps cause pipeline flush and thread switch until trap program counter (PC) becomes available. TLU also has support for up to 64 pending interrupts per thread.

Stream Processing Unit (SPU) has a Modular Arithmetic Unic (MAU) for crypto (one per core), and supports asymmetric crypto (public key RSA) for up to 2048B size key. MAU can be used by one thread at a time.

Memory Management Unit (MMU) maintains the contents of the Instruction Translation Lookaside Buffer (ITLB) and the Data Translation Lookaside Buffer (DTLB). ITLB resides in IFU, and DTLB in LSU.

Floating-point Frontend Unit (FFU) interfaces to the FPU, and decodes flosting-point instructions. FFU includes the floating-point register file (FRF). Floating-point instructions like move, absolute value, and negate are implemented in FFU, while others in FPU.

Execution Unit (EXU) is the execute stage of the pipeline, and has a single arithmetic unit (ALU) and shifter. ALU is also used for branch address and virtual address calculation. The integer multiplier has a 5 clock latency and a throughput of half-per-cycle for area saving. the integer multiplier is shared between the core pipe (EXU) and

the modular arithmetic (SPU) unit on a round-robin basis. A non-restoring divider is implemented, allowing for one divide outstanding per SPARC core. If MUL/DIV thread is fetched, it will be rolled back and switched out if another thread occupies the MUL/DIV unit.

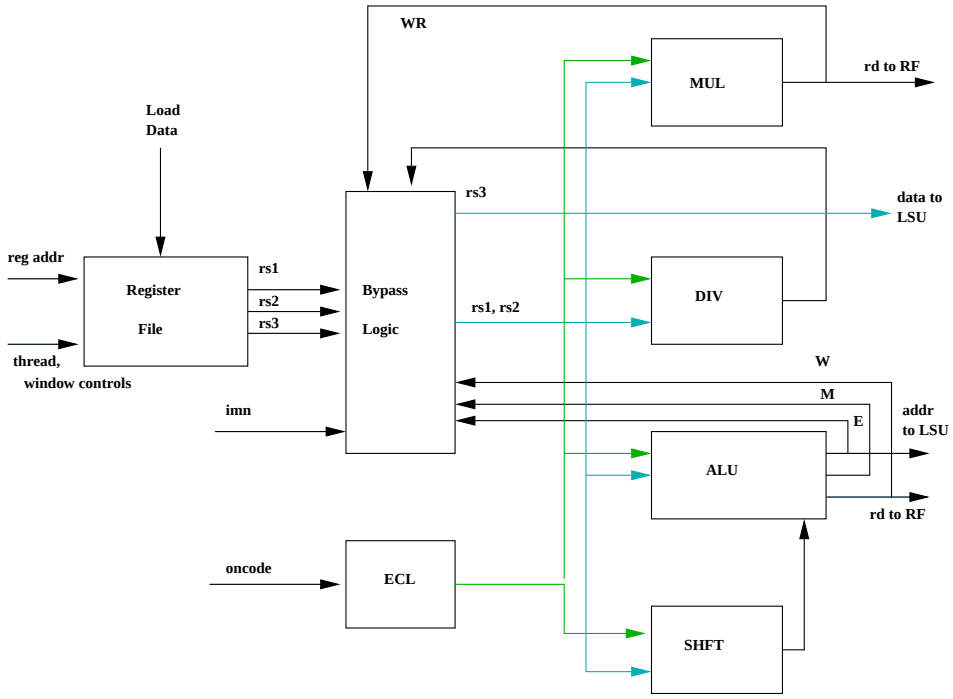

Figure 3.2: Execution Unit (EXU) Block Diagram [5].

EXU contains these four subunits arithmetic and logic unit (ALU), shifter (SHFT), integer multiplier (IMUL), and integer divider (IDIV), as shown in *figure*3.2. The execution control logic (ECL) block generates the necessary select signals that control the multiplexors, keeps track of the thread and reads of each instruction, implements the bypass logic, and generates write-enables for the Integer Register File (IRF). The bypass logic block does the operand bypass from the E, M, and W stages to the D stage.

3.2.3 OpenSPARC T1

OpenSPARC T1 is an open source version of the original UltraSPARC T1 design. The open source includes all the pieces required to build a chip including specifications and ISA, chip design and verification suite, and performance simulators [5].

Simply RISC S1 Core

Simply RISC S1 Core (code name **Sirocco**) is a project derived from OpenSPARC T1 microprocessor and shares its SPARC v9 core. The S1 core takes only one 64-bit SPARC core from the design and adds a Wishbone bridge, a reset controller and a basic interrupt controller [33]. This research uses SRISC project as the main testbed.

Wishbone Bridge

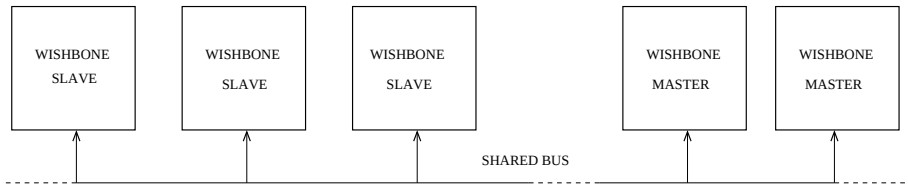

Figure 3.3: Wishbone Shared Bus Topology.

Wishbone bridge is an open source hardware computer bus intended for integrated circuit communications. The purpose of the bus is to allow the connection of different hardware components such as IP cores. A large number of open-source designs for CPUs and auxiliary peripherals are using Wishbone Interface [34]. Wishbone adapts well to common topologies such as point-to-point, many-to-many, or crossbar switches. Simply RISC S1 core uses a Wishbone bridge with a shared bus as shown in *figure* 3.3.

Wishbone Communications

Wishbone bus adopts a standard data transfer handshaking protocol between the bus users as shown in *figure* 3.4. The operations in the read and write cycles are elaborated below:

Write cycle:, Wishbone Master or the host presents an address on its $ADDR\_O$ output for the memory address that it wishes to read, then sets WE_O output to specify a write cycle. The host defines where the data will be sent on $DATA\_O$ output using its SEL_O signal (selects the memory block or the peripheral device) [6].

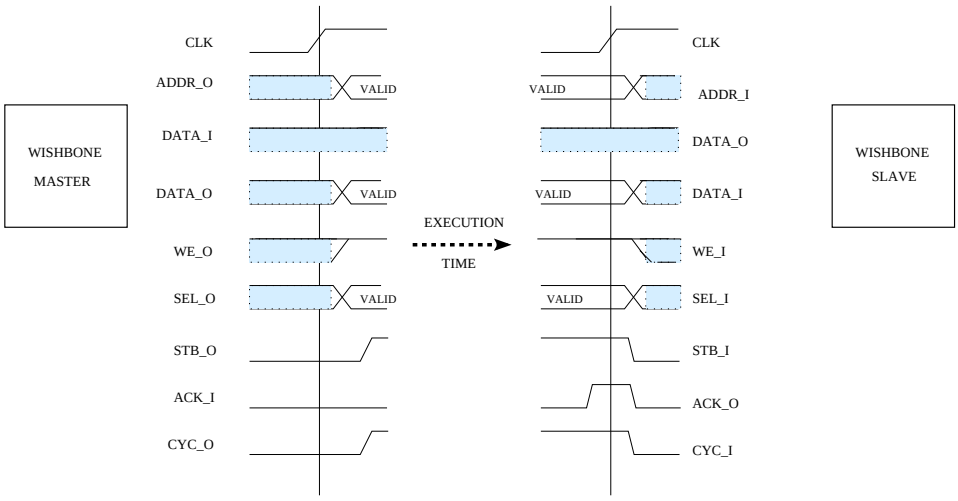

Figure 3.4: Wishbone Data Transfer Handshaking Protocol / Write Cycle [6].

Wishbone Slave receives the address at its $ADDR\_I$ input and prepares to receive the data. The host sets STB_O and CYC_O outputs, indicating that the transfer is to begin. The slave, monitoring its STB_I and CYC_I inputs, reacts to this assertion by storing the data appearing at its $DATA\_I$ input at the requested address and setting its ACK_O signal. The host, monitoring its ACK_I input, responds by negating the STB_O and CYC_O signals. At the same time, the slave negates ACK_O signal and the data transfer cycle is naturally terminated.

Read cycle:, the host presents an address on its $ADDR\_O$ output for the address in memory that it wishes to read, then clears its WE_O output to specify a read cycle. The host defines where it expects the data to appear on its $DATA\_I$ line using its SEL_O signal as shown in $figure$ 3.5.

The slave receives the address at its $ADDR\_I$ input and prepares to transmit the data from the selected memory location. The host sets its STB_O and CYC_O outputs indicating that the transfer is to begin. The slave, monitoring its STB_I and CYC_I inputs, reacts to this assertion by presenting the valid data from the requested location at its $DATA\_O$ output and sets its ACK_O signal. The host, monitoring its AACK_I input, responds by latching the data appearing at its $DATA\_I$ input and clearing STB_O and

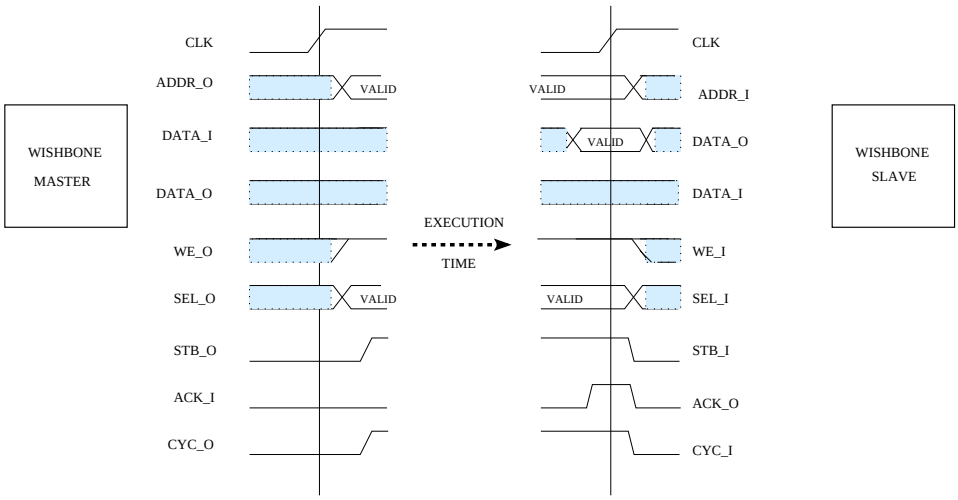

Figure 3.5: Wishbone Data Transfer Handshaking Protocol / Read Cycle [6]

CYC_O signals. At the same time, the slave negates *ACK_O* signal and the data transfer cycle is naturally terminated.

In OpenSPARC T1 microprocessor, the eight SPARC cores make use of a proprietary protocol with the rest of the chip; this protocol is often referred to as PCX/CPX protocol, where PCX stands for ``processor-to-cache Xbar ´´and is used for the outgoing requests from SPARC cores, and CPX stands for ``cache-to-processor Xbar ´´and is used for the incoming packets [5]. As for Simply RISC interface, the main block designed specifically for the S1 core is the ``SPARC core to wishbone master interface bridge ´´that translates the requests and return packets of the SPARC core into the wishbone protocol.

The main specifications of the wishbone bridge implemented in Simply RISC are summarized as follows:

1. The address bus is 64 bits.
2. The data bus is 64 bits supporting 8, 16, 32 and 64 bit accesses.
3. Data transfer ordering is Big Endian.
4. Supports single Read/Write cycles.

3.2.4 Conclusions

In this chapter, UltraSPARC T1 architecture was introduced, with emphasis on the execution stage, as DCS is implemented on the execution modules. The chapter also presents an overview of Simply RISC S1 core project, which is used as thesis testbed. Simply RISC was chosen due to its simplicity compared to OpenSPARC T1 since it only implements a single RISC core, which satisfies the requirements.

Chapter 4

Methodology

4.1 Introduction

In this research, an implementation of **D**ynamic voltage and frequency scaling with multi **C**lock distribution **S**ystems (DCS) on the multiplier unit of SPARC core is discussed. DCS is introduced to optimize power at a finer granularity.

The following sections will cover DCS implementation in details. In section 4.2, a behavioral description of DCS will be covered. Design implementation is discussed in section 4.3, followed by an overview of data synchronization in DCS in section 4.4.

4.2 DCS Approach/Design

DVFS dynamically controls the entire core frequency/voltage (F/V) setting based on its temperature, power consumption, and computational needs. The maximum F/V setting is determined by the power envelope. A single overheated component (hot spot) within the core can be the root cause of exceeding that power envelope, due to its heavy computation. A hot spot can change the global F/V setting, and lower the throughput. Montecito architecture, of the Itanium family processor, is an example to this [24]. In the execution stage in Montecito, the hot spots with heavy computation are the floating point unit and the integer unit.

DCS applies F/V settings on a relatively smaller granularity, resulting in multiple

F/V zones within the core. The hot spots are monitored, and the F/V setting of the module containing only the hot spot, is reduced. Therefore, the core can maintain its throughput and global F/V setting.

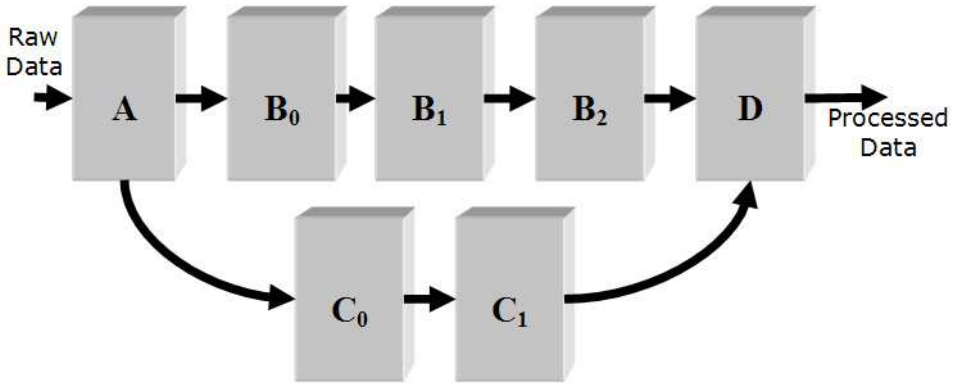

Figure 4.1: Example of Data Flow Fork.

DCS introduces multiple independent F/V zones within the core (multiple clock domains), allowing the different core modules to operate asynchronously. To further illustrate, an example of a processor flow that forks into two parallel paths is shown in *figure* 4.1.

The assumption made in this example is that each module in the processor (i.e. A, B, C, and D) has its own clock distribution tree (i.e. independent F/V zone). In the scenario where the hot spot is in one of the B stages, B's frequency is reduced, while A, C and D operate at a higher frequency. Since C maintains the original F/V setting, the throughput is not affected. In terms of performance, the original processor (without DCS) will drop its global clock frequency (lower its F/V setting) which results in a lower throughput, while DCS maintains the throughput and eliminates the hot spots, so its throughput, relative to the original processor, is greater. The same scenario applies if the hot spot is in one of C stages.

In the scenario where the hot spot is in D, the throughput is reduced same as in the original processor with global clock distribution system (i.e. single F/V zone). The same scenario applies if the hot spot is in A. In this type of scenarios, the F/V setting of all

zones have to be reduced.

Based on the example above, an increase in throughput occurs in three conditions:

- *A certain parallelism has to exist within the core:* Parallel paths allow the separation of hot spots while maintain the throughput. This is achieved by exploring the core modules, where in the execution stage of the core, multiple execution units, such as the multiplier and ALU, can operate in parallel to each other.

- *The location of the hot spots:* maintaining the same throughput is possible if and only if the hot spots are located in parallel paths. Otherwise, a bottleneck can appear if the hot spot is located in an un-parallel path, and the throughput is dropped. This is achieved by analyzing the power consumed by each parallel path or module, or by the core execution stage modules in this case.

- *Maintaining data integrity:* B and C get their data from A, while D gets its data from B or C. In an asynchronous environment, some synchronization protocol needs to be implemented to ensure the a synchronous flow of data. This is achieved by implementing a self-timed interface between the different modules that ensures data integrity.

4.3 DCS Implementation

DCS design is implemented on the execution stage in the SPARC core. The execution unit has four parallel paths; multiplier, divider, ALU, and shifter, as shown in *figure* 3.2. The design separates the multiplier's clock system from the core's, which gives PCU control over its frequency. The other paths operate on the same global clock system.

A high level overview of the design modules and their interaction with SPARC core is shown in *figure* 4.2. DCS consists of the following units:

- Power Control Unit (PCU) is the main design module, it reads the temperature and determines the operating frequency of the parallel paths.

- Timer is the module that triggers PCU to execute, it represents the stimuli environment of PCU.

- CLK_EMU emulates the VID to frequency converter, this module is an emulation of the voltage to frequency converter circuitry.
- TEMP_EMU emulates the temperature sensors located at the hot spots in SPARC CORE.

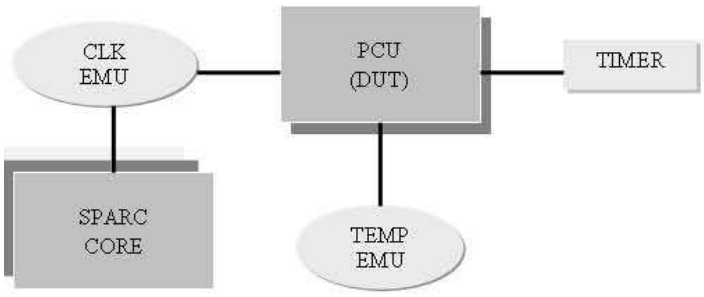

Figure 4.2: Design Basic Modules and their Interaction with the Core.

The following sections cover a more detailed discussion of each module and their interaction with each other, followed by a top level overview of the design and its interaction with SPARC core.

4.3.1 TEMP_EMU

TEMP_EMU generates random temperature values over time for the simulation duration, based on the formula below:

$$T_{(n)} = \alpha * T_{(n-1)} + \Delta \tag{4.1}$$

Where $T_{(n)}$ is the temperature provided to PCU. $T_{(n-1)}$ is the previous temperature value. α is a decay factor ranges between 0 to 1. Δ is the randomly generated sensor reading.

$T_{(n)}$ consists of two components; $\alpha * T_{(n-1)}$ and Δ. $\alpha * T_{(n-1)}$ component accounts for the previous measured temperature, while Δ accounts for the delta sensor reading generated since the last reading was taken. The assumption made here is that the sensor

reading is represented with an incremental counter that resets every time the sensors are sampled.

The reason behind accounting for the previous temperature reading is that the die needs time to dissipate heat. α value depends on the process technology, die thermal conductivity, and the location of the thermal sensor with respect to the hot spots [26]. Each sensor would have a different α value.

If the area around the thermal sensor generates heat, Δ would have a positive value. If no heat is generated, Δ would have a zero value, and $T_{(n)}$ value would decrease due to the decay in $\alpha * T_{(n-1)}$. The temperature range is from 0^oC to 125^oC, where 125^oC is the maximum temperature at which the transistors can operate.

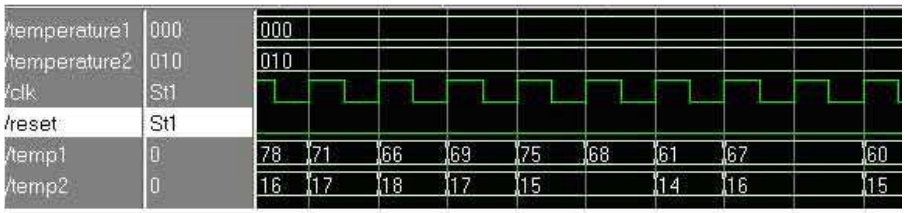

Figure 4.3: Temperature Generation in TEMP EMU Unit.

The designed module for TEMP_EMU is *clk* driven, it has two outputs; each represents a single path temperature. Four modes of operation are implemented; *High*, *Medium*, *Low*, and *Random*, as shown in table 4.1.

Table 4.1: TEMP_EMU Modes of Operation

Operation Mode	Temperature Range (in ^{o}C)
High	91 - 125
Medium	51 - 90
Low	0 - 50
Random	0 - 125

Each mode of operation reflects the traffic in the module, for example, mode *High* reflects high traffic in the module, and so on. Mode *Random* is used for long term simulations that have variable traffic. Variable traffic results in different temperatures.

These modes are used for simulation purposes, as shown in $figure$ 4.3, where $temperature1$ and $temperature2$ are the two sensors operation modes. $temperature1$ is set to mode

Medium which results in the generation of *temp*1, and *temperature*2 is set to mode *Low* which results in the generation of *temp*2.

4.3.2 CLK_EMU

CLK_EMU takes VID values from PCU, and determines the clock frequency. CLK_EMU uses counters that overflow when reaching a certain maximum threshold, this threshold is re-evaluated as VID changes. When the counter overflows, the clock signal switches. The clock signals switching frequency is based on the counter's threshold.

4.3.3 TIMER

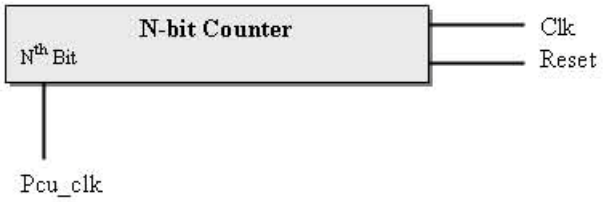

Figure 4.4: TIMER Module Block Diagram.

TIMER triggers a start signal for PCU FSM to run a complete loop. The triggering frequency is synchronized with the temperature sensors frequency; both frequencies are set to a fixed value. Figure 4.4 shows a block diagram for the TIMER module. *clk* is the global system clock. *pcu_clk* is the clock signal for PCU, it is connected to N^{th} bit of the clock frequency scaled down by 2^N. Also connecting to N^{th} bit assures that *pcu_clk* has equal high and low time.

4.3.4 PCU

The Finite State Machine (FSM), shown in $figure$ 4.5, represents PCU's behavior; it starts in the *Idle* state waiting for a trigger signal from the timer. The frequency at which the timer triggers the signal depends on how often FSM loop needs to be executed. FSM loop frequency depends on the reading update frequency of the temperature sensor. The following paragraphs will discuss the activities happening in each state.

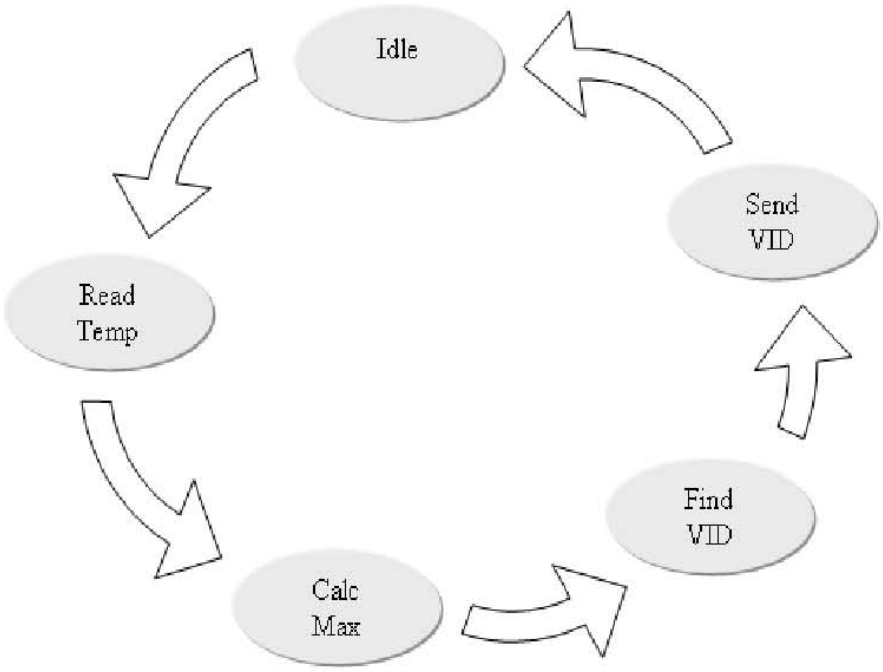

Figure 4.5: Power Control Unit State Sequence Starting with Temperature Reading and Ending with VID Sending.

In *Read_Temp* state, PCU reads sensors temperature generated by TEMP_EMU. The values generated represent real temperature values in Celsius. Real sensors will produce voltage readings that reflect actual temperatures. The sensors are located to reflect the temperatures of the hot spot in each path in the core (two sensors; one in the multiplier, and the other in the ALU).

Table 4.2: VID vs. Temperature LUT in Find VID State

Temperature Range	VID
112 - 127	6
96 - 111	5
80 - 95	4
64 - 79	3
32 - 63	2
0 - 31	1

In *Calc_Max* state, the maximum temperature of all paths is calculated (the multiplier and the ALU). This maximum temperature determines the core's maximum tem-

perature and controls the frequency of the non-parallel paths in the core, and the paths that have the same global clock system (not controlled by PCU).

In *Find_VID* state, the temperature values for each path determine VID of that path. A look-up-table (LUT) is used to find VIDs (VID LUT will be further discussed later in this section). The table contains six entries; each entry represents a power state (F/V setting) in which the core can operate. Greater VID value results in sending the module to deeper sleep mode, as discussed in section 2.2.3. The sensor's temperature range (0 to 125^{o}C) is broken down to six fields, with range of 15^{o}C for high temperature ranges, and 30^{o}C for lower states.

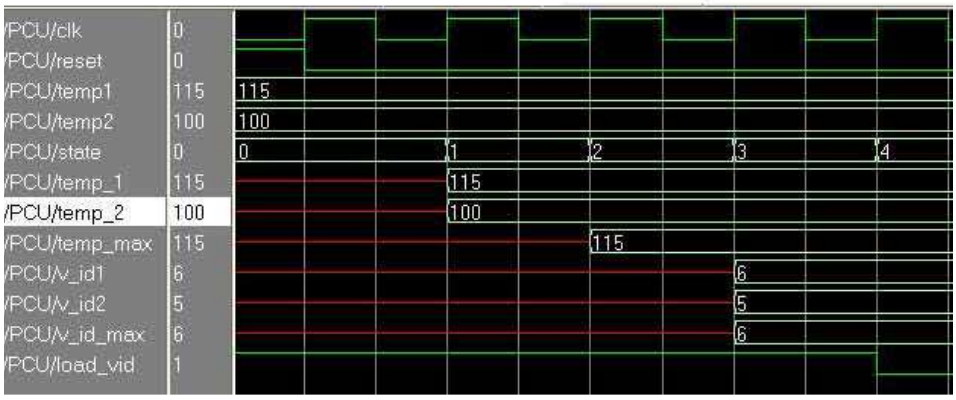

Figure 4.6: PCU Simulation Showing FSM State Flow.

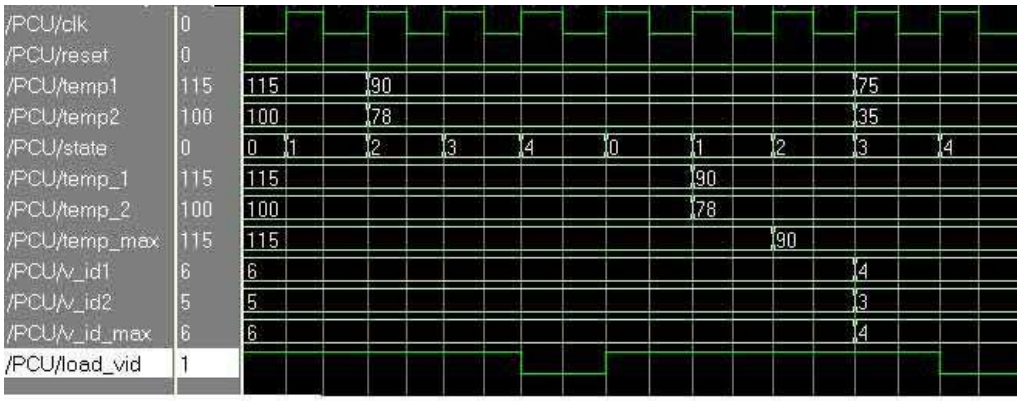

Figure 4.7: PCU State Flow with Fluctuate Temperature Simulation.

In *Send_VID* state, VID values are sent to CLK_EMU which converts VIDs to equivalent frequencies that can operate at those voltages. It enables a load signal for

CLK_EMU, which is used to load the new VIDs. CLK_EMU provides the clock signals to the proper path. A single PCU FSM loop is shown in *figure* 4.6. *temp*1 and *temp*2 are the input temperatures in Celsius, *state* indicates PCU's current state; $IDLE$, $READ\_TEMP$, $CALC\_MAX$, $FIND\_VID$, and $SEND\_VID$ respectively. In $READ\_TEMP$, $temp\_1$ and $temp\_2$ registers are updated with the input temperatures. In $CALC\_MAX$, $temp\_max$ register is calculated. In $FIND\_MAX$, v_id1, v_id2, and v_id_max are fetched from LUT in *table* 4.2 and sent to output. In $SEND\_VID$, $load\_vid$ is reset to indicate that the VIDs values are ready for CLK_EMU to consume.

The VID LUT mentioned above and shown in *table* 4.2 was introduced for simulation purposes. The table divides the temperature ranges into specific fields to merely introduce a range of inputs for PCU. The actual number of entries and the field of each entry depends on a number of variables, such as the frequency and voltage operating points, type of thermal sensors implemented, and the important temperatures; such as 125^0C which is a critical temperature that the chip should never reach.

A simulation of PCU with fluctuating temperatures is shown in *figure* 4.7. *temp*1 and *temp*2 temperature values vary with respect to time. PCU samples the temperature values in $READ\_TEMP$ state (state_0 in the simulation), calculates the maximum temperature in state_2, and sends the ready signal in state_4. The simulation shows multi iterations of PCU flow.

4.3.5 Top Level Overview

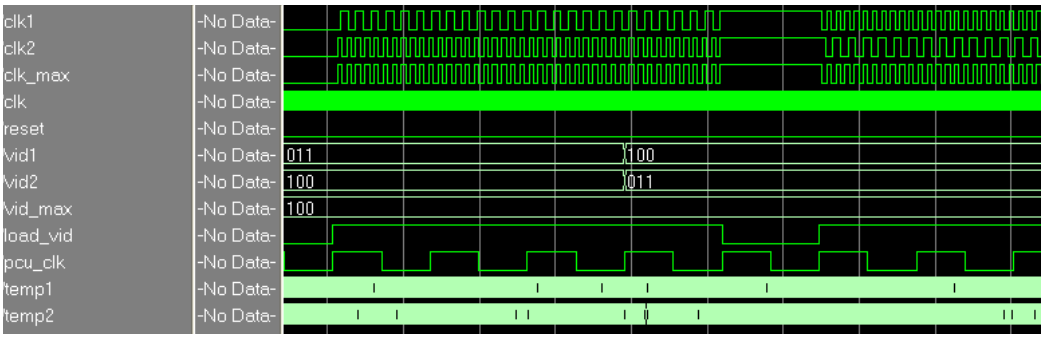

Figure 4.8: Clock Frequency Change with respect to VID Change Simulation.

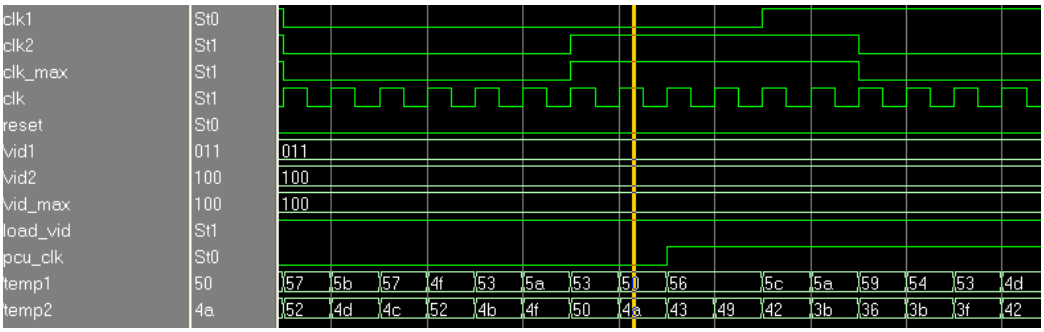

Figure 4.9: VID Change with respect to Temperature Change Simulation.

A simulation of the top level design is shown in *figure* 4.8, *clk*1 is the clock signal that enters MUL unit, *clk*2 is the clock signal that enters the core (all non-MUL modules), *clk_max* is the core global clock signal, which in this case, is the same as *clk*2. *clk* is the global system clock, *vid*1 and *vid*2 are VID values that control MUL and core frequencies respectively. *load_vid* is the control signal from PCU to CLK_EMU trigger CLK_EMU counters to change their threshold frequencies. *pcu_clk* is PCU clock signal. *temp*1 and *temp*2 are MUL and core's temperatures respectively.

In the waveform *clk*1 is the clock signal that goes to the first path, *clk*2 is the clock signal that goes to the second path. The waveform shows these signals in three different intervals, the first interval shows that *clk*2 has a higher frequency than *clk*1. In the second interval, *clk*1 has a higher frequency than *clk*2. In the third interval, both have the same frequency. In all intervals, *clk_max* has the highest frequency of both. *clk_max* is the clock signal that goes to the un-parallel paths of the execution stage.

The simulation shows 5 cycles of PCU state machine. In the 1st cycle (*IDLE* state), PCU performs no activities. In the 2nd cycle, the temperature is sampled and is used to determine VIDs. In the 3rd cycle, PCU determines the operating point of MUL and core. In the 4th cycle, VID registers are updated, represented by *vid*1, *vid*2, and *vid_max*. In the 5th cycle, the VID values are sent to the CLK emulator which based on VID changes the clock frequencies of the paths.

A closer look at the temperature values that caused the transition of VID values is shown in *figure* 4.9. The waveform shows that at the rising edge of the 2nd cycle of

pcu_clk (PCU goes to READ_TEMP state), *temp*1 is higher than *temp*2, this is when PCU samples the temperatures. *vid*1 changes from 011 to 100 and *vid*2 changes from 100 to 011.

4.4 Data Integrity in Asynchronous Path

Each of the paths in the execution stage can operate at different frequencies and independent to each other. In *figure* 3.2, the parallel paths in discussion are MUL, DIV, ALU, and SHFT. MUL has a separate clock distribution system, while DIV, ALU, and SHFT use the same global clock system of the core. MUL interfaces with EXU using wishbone protocol as shown in *figure* 4.10.

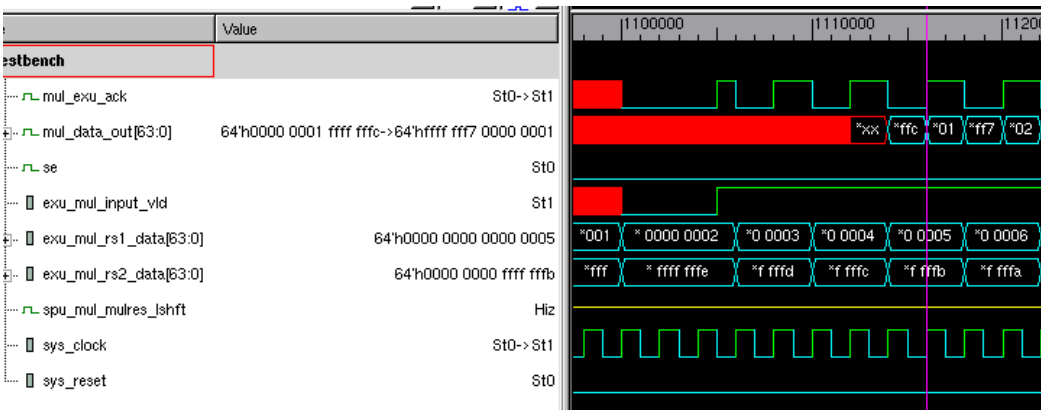

Figure 4.10: Multiplier Unit Simulation Showing the Number of Clock Cycles for First Valid Output.

The unit receives a new instruction every falling edge of *mul_exu_ack* signal. MUL starts processing when *exu_mul_input_vld* is asserted, the signal reserves the unit and indicates that the input data is available. MUL asserts its *mul_exu_ack* output signal to indicate that it received the data to process, then resets it to indicate the completion of data reception. EXU monitors *mul_exu_ack* signal and puts new data set on *exu_mul_rs*1*_data* and *exu_mul_rs*2*_data*. At the next rising edge of MUL clock signal, it sets *mul_exu_ack* to acknowledge data reception, and so on.

MUL propagation delay is 5 clock cycles. By the third *mul_exu_ack* cycle, MUL would have processed the first set of data. Notice that the first output data (i.e. 0x1ffffffc)

is available after five clock cycles and at the third falling edge of *mul_exu_ack* signal. This output corresponds to the input data when *exu_mul_input_vld* went high (i.e. 0x2 and 0xfffffffe).

EXU stage monitors the acknowledge signal as an indicator when to consume the output data, it doesn't use the system clock to synchronize data flow. This hand-shaking mechanism controls MUL frequency and ensures its independence to system clock as explained in the following section.

4.5 Self-Timed Interface

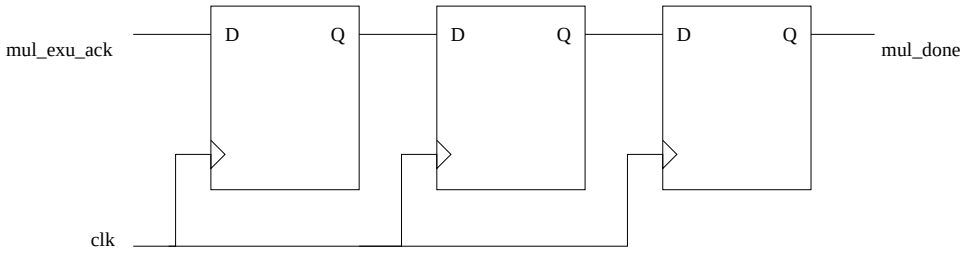

Figure 4.11: Self-Timed Interface at Wishbone Master.

Self-timed or asynchronous circuits operate without a global clock that rules the operation of the whole system. Self-timed elements synchronize themselves by hand-shaking signals. The absence of a global clock make the behavior of the system to be non-deterministic and permits the implementation of highly concurrent systems [35].

In SPARC core, EXU unit is the wishbone master that drives MUL unit. The self-timed interface is represented by the implementation of wishbone interface between EXU and MUL modules. *mul_exu_ack* output signal indicates when data is taken at the rising edge, and when data is processed at the falling edge. Every data packet entering MUL unit takes 3 acknowledge cycles to process. The Self-Timed interface latches the acknowledge signal to consume the output when ready, as shown in $figure$ 4.11.

4.6 Clocking Overhead

Although the objective is to enhance core performance by implementing dynamic voltage and frequency scaling to relatively smaller domains, dividing core clock domain into multiple domains introduces an overhead. In this section, an overview of the clock over will be presented.

A basic clocking system consists of a Phased Locked Loop (PLL) and a clock distibution tree. PLL generates high frequency clocks required by processors. The clock distribution tree may be implemented using H-tree clock-distribution network, matched RC trees, or grid structure, or a combination of two or more [17].

Assuming a basic clock system, PLL generates a fixed clock frequency which gets distributed over the network. Introducing DCS into this clock system would require implementing multiple Digital Frequency Dividers (DFD) to provide multiple frequency settings, in addition to a switch that passes the clock frequency based on requested setting from DCS.

In modern processors on the other hand, components operate at multiple frequencies; the front side bus might operate at 100 megahertz while the core operates at frequencies greater than 1 gigahertz. The processor already contains multiple DFDs, and assuming that the existing DFD cover the necessary frequency settings for DCS, the overhead in this case is represented by the clock switch.

In addition, introducing multiple clock domains compared to single clock domain will reduce the overall clocking hardware, since with multiple clock domains, the clock tree in each domain is independent of the trees in the other domains, which eliminates the wiring that is used to propagate the clock from one domain to the others.

4.7 Summary

In this chapter, the methodology was discussed in details. DCS applies dynamic voltage and frequency scaling on smaller granularity within the processor core. The core modules are represented by the execution unit components such as the multiplier and the ALU. DCS replaces the core single clock distribution tree with a multi-clock do-

main systems, and along with dynamic voltage and frequency scaling, it creates multiple clock/voltage domains. Frequency/Voltage scaling is determined based on the temperature in each domain. DCS also implements a self-timed interface between the different domains to maintain functionality and ensure data integrity.

Chapter 5

Results and Analysis

5.1 Introduction

In this chapter, exuction paths are analyzed to determine ``DCS friendly´´paths in section 5.2, followed by a more detailed analysis of MUL unit which is used in DCS is covered in section 5.3. In section 5.4, throughput and power gain using DCS is discussed, the section covers different parameters that contribute to speedup and power analysis.

5.2 SPARC Core Parallel Paths

Each core has hardware support for four threads. A register file is implemented per thread. The four threads share the instruction, the data caches, and the TLBs. The core has a single issue, six stage pipeline, as shown in *figure* 5.1. Each execution component was analyzed to determine the suitable components to separate from the core global clock distribution system. The execution stage of the pipeline contains the following units:

- **ALU:** takes only one clock cycle to complete its execution, this relatively small propagation delay of ALU minimizes the desired throughput gain from this unit. In addition, ALU adder is used in Load/Store address calculation, changing ALU frequency will result in timing misalignment, which introduces a hazard of mis-

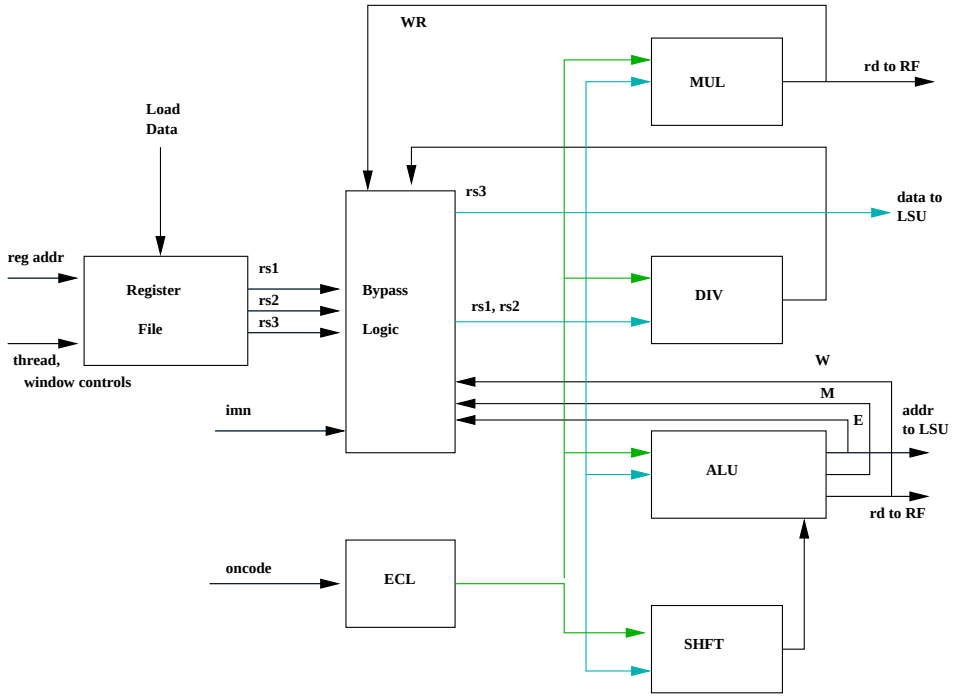

Figure 5.1: Execution Unit (EXU) Block Diagram [5].

calculating the memory addresses. Introducing asynchronous elements to ALU will only increase the propagation delay of the unit.

- **Shifter:** consumes significantly low power (a few mW), compared to the total EXU power. Therefore, it is not a feasible solution to implement DCS on this unit. In this case, the introduction of additional hardware for DCS might be costly in terms of power.

- **Multiplier (MUL):** takes five clock cycles to complete execution of a single instruction. The unit is pipelined, so the throughput can be one instruction per clock cycle. MUL possesses Self-Timed elements (Handshake signals) to ensure data integrity. MUL consumes $\sim$ 10% of total core power. So it is an ideal choice to implement DCS.

- **Integer divider:** contains a non-restoring divider, and supports one outstanding divide per core. The unit consumes $\sim$ 10% of total core power. The unit represents

a suitable component for DCS implementation, however it was not chosen as the execution time is indeterministic since it is data-dependent.

For analysis purposes, it is easier to measure speedup when the unit propagation delay is fixed. In addition, a fixed propagation delay results in a more accurate power analysis since the synthesis tool used measures only average power, which would be in-accurate in case of variable propagation delay.

5.3 Multiplier Unit

MUL is a common unit used by both SPU and EXU. When both units require multiplication processing, the control over MUL is scheduled on a round-robin basis. In each clock cycle, MUL would alternate processing data between EXU and SPU.

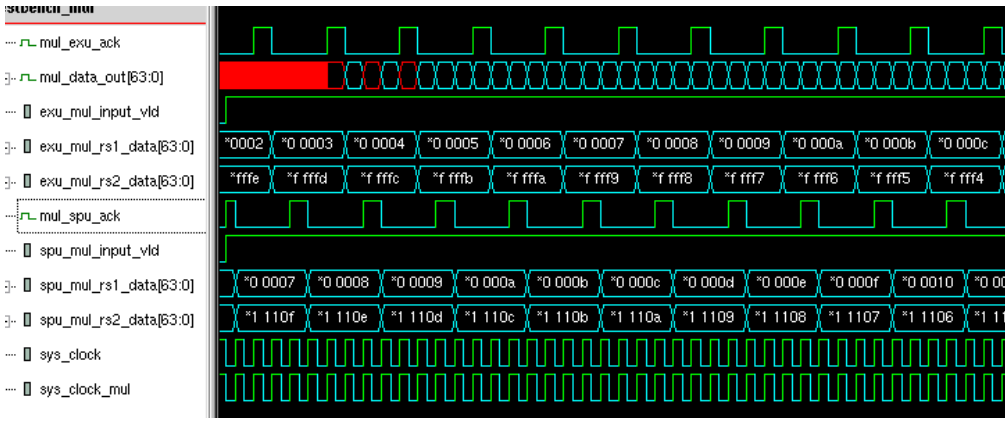

Figure 5.2: Multiplier Unit Simulation with Both EXU and SPU Requesting Service.

A simulation of MUL unit with both EXU and SPU active and requesting control is shown in *figure* 5.2. It shows both EXU and SPU requesting service from MUL by asserting *exu_mul_input_vld* and *spu_mul_input_vld* respectively. MUL services both units on round-robin basis, it asserts *mul_spu_ack* to indicate to SPU the reception of SPU data from *spu_mul_rs*1*_data* and *spu_mul_rs*2*_data*, then resets *mul_spu_ack* to indicate the completion of the SPU write cycle. SPU changes its input by the end of each write cycle.

Once MUL services SPU, the next cycle will be dedicated to EXU; it services EXU by asserting *mul_exu_ack* signal to indicate the reception of data from *exu_mul_rs1_data* and *exu_mul_rs2_data*. Then *mul_exu_ack* is reset to indicate the completion of the EXU write cycle. EXU changes its input by the end of each write cycle.

As for reading from MUL, a different data path is used separate from the write flow. *mul_data_out* is a common read path that services both EXU and SPU on round-robin basis. However, the *mul_exu_ack* and *mul_spu_ack* signals that are used for writing are also used for reading at the same time. Each reset of the acknowledge signal indicates the completion of both the read and write cycles for the corresponding unit. MUL puts the output data on *mul_data_out* and resets *mul_exu_ack* signal if the data is directed to EXU, or *mul_spu_ack* if the data is directed to SPU.

5.3.1 Multiplier Functionality with DCS Design

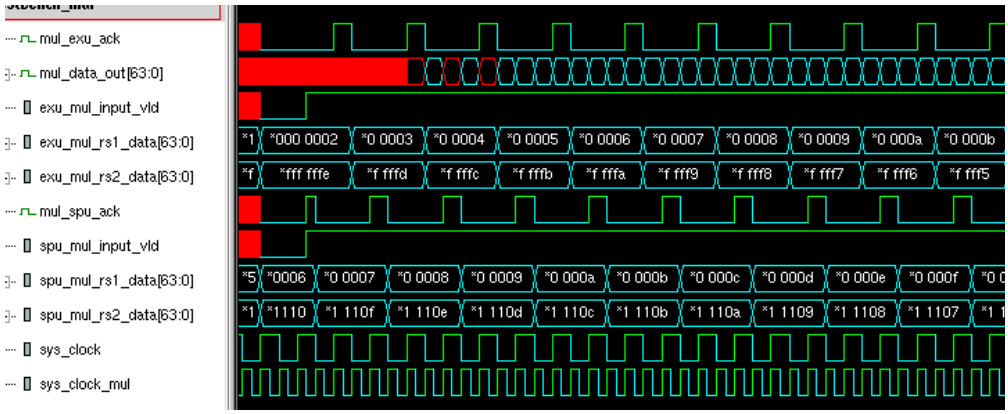

Figure 5.3: Multiplier Unit Simulation at 2X Core Frequency.

DCS design permits the multiplier unit to operate asynchronous to the global clock system by separating MUL clock distribution tree from the core's. DCS controls MUL frequency based on the temperature readings from both the core and MUL. This results in two modes of operation: Overclocking and Underclocking.

- *Overclocking mode:* is entered when the temperature of both the core and MUL is low. In this mode, MUL can operate at a higher frequency than that of the core's,

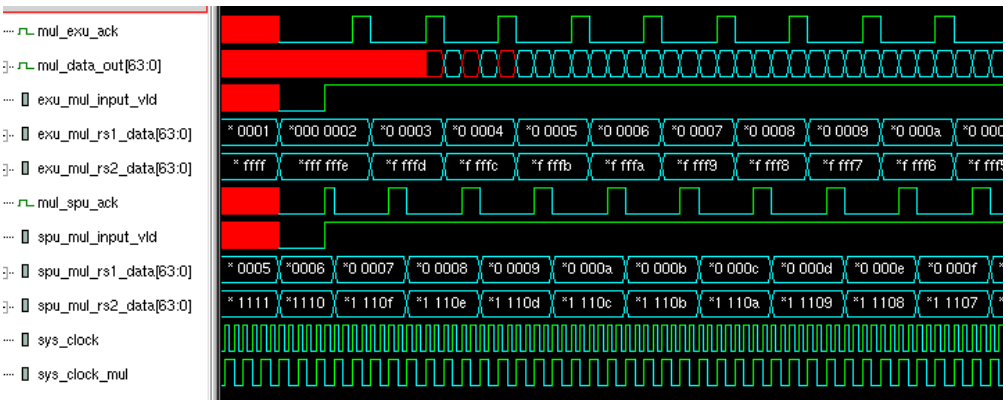

Figure 5.4: Multiplier Unit Simulation at 1/2X Core Frequency.

and since MUL is used by both EXU and SPU, its throughput is not limited to EXU issue rate, as shown in *figure* 5.3. The simulation waveforms show MUL operating at a double frequency rate of the core's; where *sys_clock* is the core clock frequency, and *sys_clock_mul* is MUL clock frequency. The simulation also shows that MUL is fully utilized since both EXU and SPU are issuing a new set of input data every cycle of *sys_clock*. However, this also limits the maximum frequency at which MUL can operate to 2X of *sys_clock* frequency rate.

- *Underclocking mode:* is entered when the temperatures of MUL and/or core are high. DCS reduces MUL frequency and maintains core frequency, as shown in *figure* 5.4. Since MUL processes only multiplication operations, core throughput is not affected as long as the multiplication percentage is relatively low (core pipeline is always full). Further analysis will be discussed in the following sections.

5.4 Performance Analysis

5.4.1 Throughput

DCS was implemented to control the multiplier clock frequency independent of core frequency, allowing up to 2X increase in core frequency. Increasing MUL frequency any higher will result in EXU missing MUL acknowledge signals.

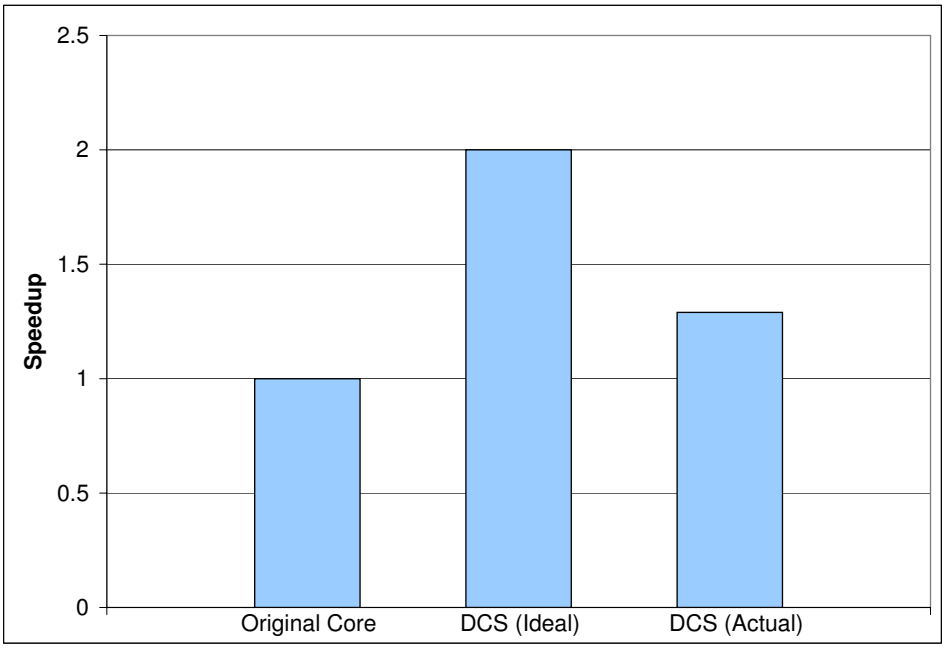

Figure 5.5: DCS Ideal and Actual Speedup with no Cache Miss.

To achieve the ideal or the maximum possible speedup with DCS, some assumptions were made:

- MUL is assumed to always have data to process, where the overhead of MUL is 5 clock cycles and keeping the pipeline full will overlap this overhead.
- No cache miss is allowed, where any cache miss will result in disrupting MUL execution and longer execution time.
- No data dependency exists, where any data dependency can cause a delay or bubbles in the execution flow if there are no proper forwarding mechanisms.
- The operations stored in the instruction cache are only of multiplication type, where other types of operations will allow the core to overlap execution since the core contains several execution units to process in parallel.

In this ideal case, the maximum expected performance equals the maximum frequency increase allowed for MUL, which is 2X, as shown in *figure* 5.5. However, this ideal case analysis is merely presented to show the maximum speedup limit of DCS.

The measured speedup for DCS is based on Amdahl's Law indicated in the following formula

$$SpeedUp = \frac{Performance_{DCS}}{Performance_{Original}} = \frac{Execution_{Original}}{Execution_{DCS}} \tag{5.1}$$

where in the ideal scenario, the execution time of DCS is half the execution time of the original core. It is assumed that any instruction fetch overhead will converge to 0 over long period of execution (assuming that there is no cache miss in the ideal case).

To get a more realistic speedup analysis, a test program is run. The program consists of a number of instructions that vary from simple operations like shifting and ORing, to more complex operations like addition and multiplication. The multiplication operations are 20% of the total instructions. The code ended with a jump instruction to the beginning of the code, resulting in an infinite loop that repeats the same instructions.

The executed program in this section was based on 20% MUL operations. The reason for using this percentage is that it has the optimal speedup-MUL_percentage ratio. However, section 5.4.3 will discuss speedup analysis with variable MUL operations percentage.

The executed program does not allow cache miss, since Simply RISC test environment that is used for DCS has removed all memory hierarchy from the design, the only form of memory left was L1 cache. The next memory level is the external memory communicated through Wishbone bridge.

The achieved speedup from executing the program is 1.29, as shown in *figure* 5.5. Although the multiplication operation percentage was 20% of the total instructions, the achieved speedup is 1.29. This can be explained by the fact that a single multiplication operation takes 5 clock cycles to complete, while other instructions take only one clock cycle, multiplication has higher weight on the overall execution time.

A throughput gain is achieved only when the executed code contains multiplication operations, otherwise, allowing MUL to operate at a higher frequency will result in wasted energy.

L1/L2 Cache

Although the model used for simulation (Simply RISC) lacks the capability to simulate L2 cache, Simply RISC still fetches instructions and data from an external memory through Wishbone bridge, which can be considered as L2 cache. The code or the benchmark used in the previous section was modified to fetch data from the Wishbone external memory, which allows misses in L1 cache. Since the data cache is large enough to hold the entire data (operand parameters) used in the code, a cache flush instruction was appended to the end of the loop code to empty the data cache and forcing the core to fetch the data again in the next iteration.

As anticipated, the execution time has increased, due to the fetch overhead, as shown in *figure* 5.6. The graph shows that DCS speedup has dropped from 1.29 to 1.21 when L1 cache miss is allowed. This increse in execution time results from the delay of fetching data again. The graph also shows the actual speedup of DCS with and without L1 cache miss compared to the original core, and the ideal case of DCS.

Although the increase in execution time in SimplyRISC doesn't represent OpenSPARC fetch delay, since they use different bus interfaces to L2 cache, it does provide an overview of L1 cache miss cost to DCS speedup.

As for L2 cache, SimplyRISC uses an external memory to memic higher memory structure. However, any cache miss in L2 cache will result in simulation failure. Any further analysis on L2 cache has to be performed on a test environment that supports the full memory hierarchy, such as OpenSPARC.

5.4.2 Power

The core and DCS design have been synthesized using Design Compiler, a tool by Synopsys. The targeted process technology is TSMC 120nm. The tool calculates the average power of the circuit. The core static power was relatively small, and is on average 1.2mW, so it can be ignored. The core dynamic power is $\sim$ 10W. The overhead power represented in DCS design is relatively small. The total power consumed by DCS is 0.173mW.

We will assume throughout this section an ideal speedup for analysis purposes unless

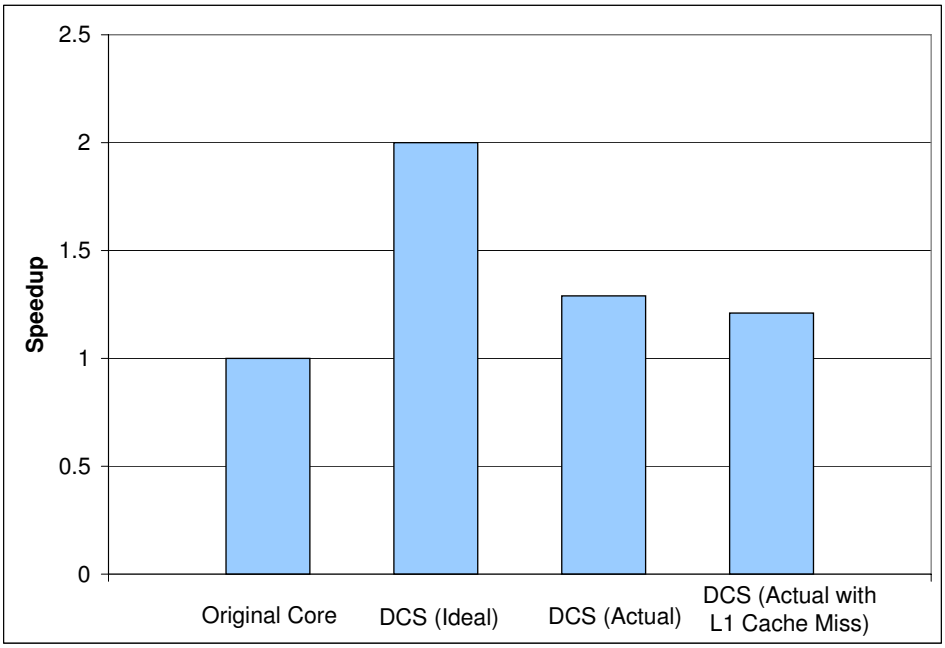

Figure 5.6: DCS Actual Speedup with and without L1 Cache Miss Relative to Ideal Case.

indicated otherwise.

Power can be analyzed from two different aspects; the first aspect is when core power has to be reduced due to a hot spot that causes the core to overheat (Underclocking Mode). The second aspect is when overclocking the multiplier unit to increase throughput while maintaining core frequency (Overclocking Mode).

Underclocking Mode

In the original core design, if a hot spot causes the core to overheat, the core frequency has to be reduced. In a core with DCS, if a hotspot causes the core to overheat, only MUL frequency has to be reduced, assuming that the hot spot is located in MUL area.

Synthesis results show that when reducing MUL frequency by half, the average dynamic power of core with DCS was reduced from 9.77W to 9.16W, as shown in *figure* 5.7, while the ideal speedup ratio increased to 2. The graph also shows that in the case of the original core, a hotspot might result in dropping the entire core frequency to reduce

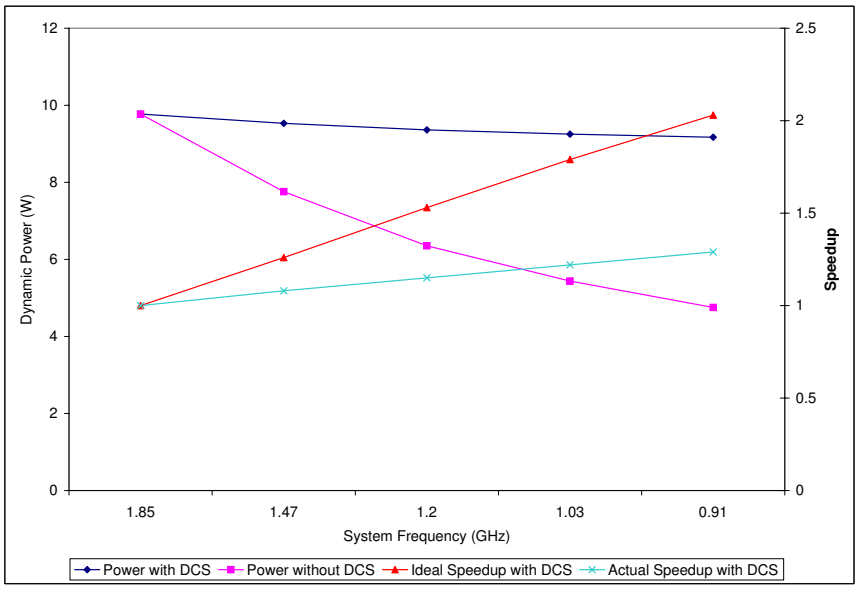

Figure 5.7: DCS Power and Speedup vs. Frequency Relation when Underclocking Multiplier Frequency.

power consumption. In this case, the core power drops from 9.77W to 4.88W. However, if dropping MUL power is sufficient to eliminate the hotspot, then core throughput would have been maintained, which explains the increase in speedup with DCS.

The ideal speedup is merely an upper bound to DCS speedup. The actual speedup of the executed code is 1.29 for 20% MUL percentage. This is a more realistic measurement since the ideal speedup assumes that MUL pipeline is always full, which implies that the executed code is mostly multiplication operations.

In the ideal scenario, the assumption made is that MUL pipeline is always full. Dropping down MUL frequency when the pipeline is full will reduce MUL and core throughput, since the entire flow is passing through MUL path. We can conclude that we can never reach the ideal case in the underclocking mode, since we can't assume MUL pipeline to be full and drop its frequency at the same time without affecting throughput.

Assuming that MUL has zero or few processes to execute in the ideal case is also unrealistic. Since it contradicts with the fact that the hot spot is located in MUL area and this hot spot would have not occurred if MUL was not processing data.

Excluding the two extremes (MUL pipeline being full and pipeline being empty), the ¨perfect¨or the ideal workload is a mix of multiplication and other instructions to ensure that other paths are being utilized to overlap the delay in MUL path.

Overclocking Mode

MUL can be overclocked while maintaining core frequency, it allows the core to have higher throughput with relatively small power increase.

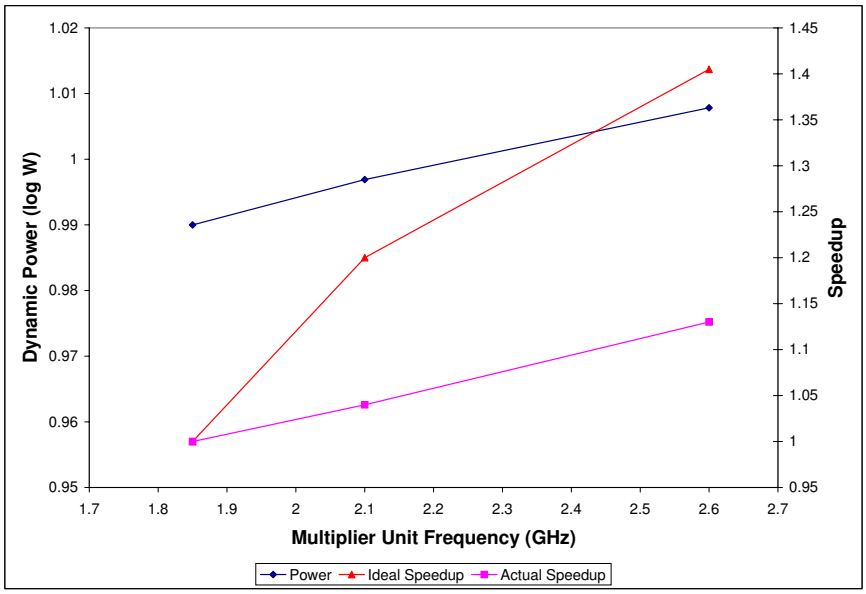

Figure 5.8: DCS Power, Speedup and Frequency Relation when Overclocking Multiplier Frequency.

In underclocking mode, MUL frequency can be increased up to 2X core frequency while maintaining functionality. However, when we synthesize the core with DCS, the maximum frequency at which MUL can operate is 2.6GHz on TSMC 120nm. Thus limiting MUL maximum frequency to 1.4X core frequency, as shown in *figure* 5.8. The expected ideal speedup was 1.4, while the achieved actual speedup was 1.13. When overclocking, a higher speedup is achieved when MUL flow is fully utilized during execution. The increase in power due to the increase in MUL frequency in overclocking

mode is only 4%.

As a result, there are two modes of operation at which the multiplier unit can operate; overclocking and underclocking mode. When the core is overheating due to a hot spot in the multiplier path, the unit will enter the underclocking mode to eliminate the hot spot. When the core temperature is within normal range, the multiplier unit will enter the overclocking mode to increase the throughput.

5.4.3 Throughput Measuerment with Different Benchmarks

Although a realistic program is when it contains a mix of instructions, some applications can exhaust one arithmetic unit more than the other, such as image processing applications. In this section, DCS is analyzed using different benchmarks that vary MUL operation percentage, as shown in *figure* 5.9.

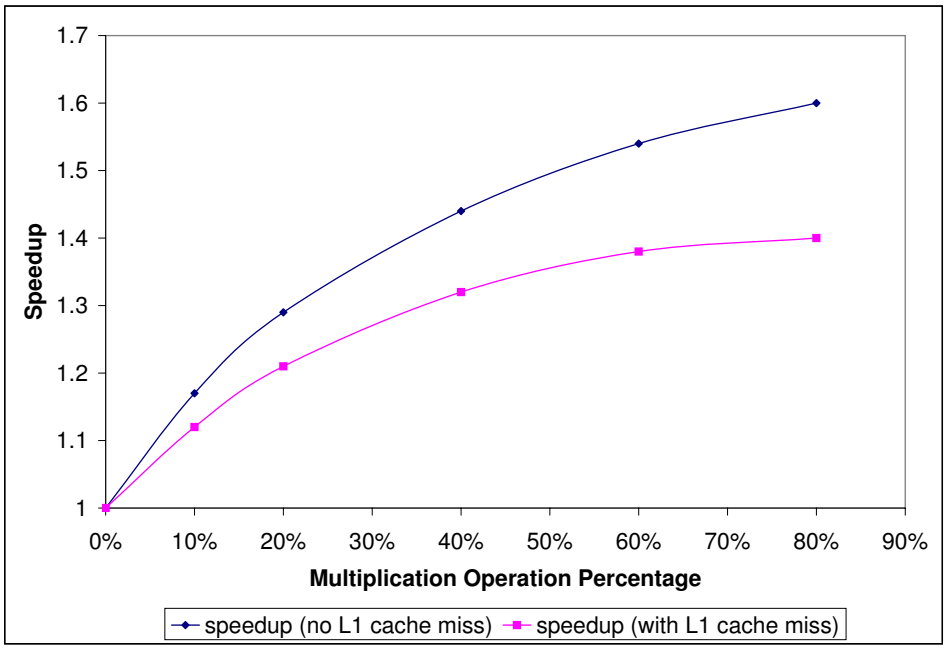

Figure 5.9: DCS Speedup with Respect to Multiplication Percentage with and without L1 Cache Miss.

As anticipated, the speedup increases when MUL percentage increases. However, the increase is non-linear for with and without L1 cache miss. This non-linear behavior

can be explained as follows:

EXU issues one instruction with each clock cycle, and can issue consecutive instructions to the same arithmetic unit as long as there is no data dependencies between the instructions. If dependency exists between ALU or shifter instructions, the forwarding unit in EXU will forward the results in order to keep the pipeline full. However, if dependency exists between MUL or DIV instructions, core would stall the following instructions until operands are ready. Due to this behavior and to the fact that the executed program has data dependency, increasing MUL operation percentage will increase MUL data dependency, which results in the increase of MUL instruction stalls. Thus, the non-linear increase in speedup when increasing MUL operation percentage.

When L1 cache miss is allowed, additional delay is added to the execution time. Since the core issues the instructions with the expectation that the data is available in the data cache. When the instruction reaches EXU stage and data is not available, the thread is rolled back and a data fetch request is issued. The rolled back threads will have to be re-issued. This roll back behavior results in wasted execution time and reduces speedup.

5.5 Other Parameters to take into Consideration

This section will discuss a number of parameters that affect DCS performance, these variables in addition to what has been discussed throughout the chapter contribute to the effectiveness of a power management system. The purpose of this section is to revisit these issues after DCS has been adequately presented and analyzed.

5.5.1 Thermal Reading

Thermal reading is the main input for any power management system. The type of thermal sensors used varies with the required accuracy, noise tolerance, and the available layout space. Thermal sensors implemented using a current source and a diode produce most accurate readings. However, these sensors require implementing BJT transistors in CMOS technology, which is an expensive solution, in terms of area and power consumption, in addition to the required post silicon calibration [26].

Ring oscillators can also be used as thermal sensors, a relatively cheap alternative to BJT sensors that can be easily implemented within the component's layout, in terms of area and power. However, ring oscillators lack the linear behavior throughout the required temperature range [26].

Temperature can also be read by estimating the power consumed by each component (by measuring voltage and knowing the package parasitic resistance), however this approach lacks the accuracy that on-die thermal sensors would provide.

Overall, thermal sensors are high maintenance components, they are process variation sensitive, noise intolerant, and require post-silicon calibration. The performance-power gain has to overweigh their cost. The type of thermal sensor restricts the implementation of the power management system; where the area and the power consumed by the sensors, in addition to the circuitry associated with the sensors, can limit the number of sensors that can be implemented.

5.5.2 Test Environment

The achieved performance gain from implementing DCS varies with different architectures. This section lists a number of variables introduced by SPARC architecture and Simply RISC test environment that affect DCS performance. First, DCS is implemented by separating the multiplier's clock system only, and excluding the other parallel paths (divider, ALU, and shifter) as discussed in section 5.2.

The implementation on MUL path limits the performance gain to multiplication instructions only. Proper utilization of the parallel paths significantly contributes to the performance gain.

Second, the assumption made in this thesis was that the hot spots occur in the multiplier unit, therefore its frequency had to be dropped. However, the power consumed by the multiplier is only 12% of the total core power, and hot spots may occur in other units, such as IFU which consumes 19% of the total core power. This limits the power-savings anticipated from MUL, although extending DCS to include other power-consuming components, such as DIV, which will reduce the overall power consumption.

Third, the propagation delay of the parallel path contributes significantly to the per-

formance gain, where the multiplier unit has a 5-stage pipeline (takes 5 clock cycles to process a single set of data), while ALU takes only one clock cycle to process a single set of data.

Chapter 6

Conclusions

In modern processors, power is managed at run time by an on-die power management system. This system continuously monitors the processor power, temperature and work load, and determines the optimal operating point of each component within the processor. This power management approach was successful in controlling the power of all components of the die, from the main computing components such as the core, to the supporting components such as the memory controller. However, this power management system addresses components power as a single entity, regardless of the power variation within each component, and in components of high power demands such as the core, power varies greatly between the core's different modules. This research is unique in that it addresses power variation of the different modules within the core.

The novel approach that is presented in this thesis (called DCS) introduces multiple asynchronous clock distribution trees within the core, these clock trees are controlled by the central power management system rather than the core global clock system. Having multiple clock trees permits an independent control over power of different modules, thus the ability to eliminate the hot spots without affecting throughput. Data integrity is preserved by implementing self-timed elements that ensures the asynchronous flow of data between the modules in different clock trees.

DCS was implemented on the multiplier module of the core, it allows the module to operate in two different modes based on the core's temperature. When the core is overheating due to hot spots in the multiplier module, the module enters the underclocking

mode. In underclocking mode, the MUL frequency is reduced to lower its power, while the core frequency is maintained, thus preserving core throughput.

When the core's temperature is within normal range, the multiplier module enters the overclocking mode. In overclocking mode, MUL frequency is increased, and since the module has a relatively large propagation delay of 5 clock cycles, the overall throughput improves for both the multiplier module and the core.

Performance was tested using several benchmarks that vary in the multiplication percentage. The maximum achieved speedup was 1.6 with 80% multiplication percentage. When the multiplication percentage is 20% , speedup in underclocking mode was 1.29, and 1.13 in overclocking mode, with only 4% power increase. The overhead power introduced by DCS was only 0.173mW compared to ~10W consumed by a SPARC core.

The approach presented in this thesis is proven to improve speedup, however it introduces additional complexity to processor design, since it needs to implement self-timed elements to ensure data integrity in synchronous systems. This imposes an interface overhead and additional validation effort to ensure functionality. The design of the power management system that calculates the optimal operating point requires having a good model of the core that allows conducting an analytical study to determine the most efficient design.

Future Work

- Alternate power measurements techniques can be explored; where Design Compiler calculates the average power. In addition to varying voltage based on voltage/frequency setting to get more accurate power measuremets.

- Implementing DCS on OpenSPARC processor to get analysis that account for L2 cache, and modifying the design to maintain data integrity, and removing the Wishbone Bridge, resulting in the ability to measure performance at full capacity.

- Extending the design to parallel units other than MUL is another area to explore. ALU was not implemented since it only takes one clock cycle, and is used from

Instruction address calculation. However, if address calculation was removed to a separate adder, ALU can also run on an asynchronous clock system.

Bibliography

[1] Shoemaker K. Yung R., Rusu S. Future trend of microprocessor design. pages 43–46, Sep 2002. [cited at p. iv, 1, 3, 4]

[2] Addressing power and thermal challenges in the datacenter: Solutions for optimizing density,availability, and cost. *Intel Solutions White Paper: Power and Thermal Management*, June 2004. [cited at p. iv, 7, 8]

[3] Power and thermal management in the intel core duo processor. *Intel Technology Journal*, 10(2), May 2006. [cited at p. iv, 8]

[4] McGowen R. Adaptive designs for power and thermal optimization. pages 118–121, Nov 2005. [cited at p. iv, 1, 10, 11]

[5] Dwayne L. The u1trasparc t1: A power-efficient high-throughput 32-thread sparc processor. pages 27–30, Nov 2006. [cited at p. iv, v, 15, 17, 20, 37]

[6] Altium designer help materials open content. wishbone communications, Feb 2008. Available at http://wiki.altium.com/display/ADOH/Wishbone+Communications. [cited at p. iv, 18, 19, 20]

[7] Rusu S. Trends and challenges in vlsi technology scaling toward 100nm. pages 194–196, Jan 2002. [cited at p. 1]

[8] Bowden M. J. Moore's law and the technology s-curve). *Steven Alliance for Technology Management*, 8, 2004. [cited at p. 1, 3]

[9] Enhanced intel speedstep technology for intel pentium m processor. *Intel Solutions White Paper: Power and Thermal Management*, March 2004. [cited at p. 1]

[10] Moore G. Cramming more components onto integrated circuits. *Electronics*, 38(8), 1965. [cited at p. 3]

[11] Bohr M. Thompson S., Packan P. Mos scaling: transistor challenges for the 21st century. *Intel Technology Journal*, 3(2):1–19, 1998. http://developer.intel.com/technology/itj/archive/1998.htm. [cited at p. 4]

[12] 13th Annual IEEE International ASIC/SOC Conference. *Obeying Moore's Law Beyond 0.18 micron (Microprocessor Design)*. Borkar S., 2000. [cited at p. 4]

[13] International Symposium on Low Power Electronics and Design. *Effectiveness and scaling trends of leakage control techniques for sub-130nm CMOS technologies*. Chatterjee B., Sachdev M., Hsu S., Krishnamurthy R., Borkar S., Aug 2003. [cited at p. 5]

[14] Sachdev M. Vassighi A. *Thermal and Power Management of Integrated Circuits*. Springer, 2006. [cited at p. 5]

[15] ISLPED. *Scaling of stack effect and its application for leakage reduction*. Narendra S., Borkar S., De V., Antoniadis D., Chandrakasan A., 2001. [cited at p. 5]

[16] Uyemura J. *Introduction to VLSI Circuits and Systems*. John Wiley and Sons, 2002. [cited at p. 5]

[17] Nikolic B. Rabaey J., Chandrakasan A. *Digital Integrated Circuits: A Design Perspective*. Prentice Hall, 2003. [cited at p. 5, 34]

[18] Svensson C. Afhahi M. Performance of synchronous and asynchronous schemes for vlsi systems. *IEEE Transactions on Computers*, 41(7):858–872, July 1992. [cited at p. 6]

[19] Giovanni De Micheli. *Synthesis and Optimization of Digital Circuits*. McGraw-Hill, 1994. [cited at p. 6]

[20] Otellini P. Beyond gigahertz. *Intel Developer UPDATE Magazine*, Sep 2001. [cited at p. 6]

[21] Scheffer L. Lavagno L., Martin G. *Electronic Design Automation for Integrated Circuits*. Taylor and Francis, 2006. [cited at p. 6]

[22] Giovanni De Micheli Luca Benini. System-level power optimization: Techniques and tools. *ACM Transactions on Design Automation of Electronic Systems*, 5(2), April 2000. [cited at p. 6]

[23] Nagaraj R. Energy-efficient system architecture aims to improve system energy efficiencies. *Technology @ Intel Magazine*, Mar 2006. [cited at p. 7]

[24] Bostak C. Ignowski J. Millican M. Parks W. H. Naffziger S. McGowen R., Poirier C. A. Power and temperature control on a 90-nm itanium family processor. *IEEE journal of Solid-State circuits*, 41(1):229–237, Jan 2006. [cited at p. 10, 12, 22]

[25] Bhatia R. McNairy C. Montecito: a dual-core, dual-thread itanium processor. *IEEE micro*, 25(2):10 – 20, March-April 2005. [cited at p. 10]

[26] Huijsing J. Bakker A. *High-accuracy CMOS smart temperature sensors*. Kluwer Academic Publishers, 2000. [cited at p. 10, 26, 48, 49]

[27] Boemo E. Lopez-Buedo S., Garrido J. Thermal testing on reconfigurable computers. *IEEE journal of design and test of computers*, 17(1):84–91, Jan -Mar 2000. [cited at p. 11]

[28] S. Memik S.O. Mukherjee R., Mondal. Thermal sensor allocation and placement for reconfigurable systems. pages 437–442, Nov 2006. [cited at p. 11]

[29] Shamir N. Genossar D. Intel pentium m processor power estimation, budgeting, optimization, and validation. *Intel Technology Journal*, 7(2), May 2003. [cited at p. 11]

[30] Yang J. Puchkarev V. Li L., Zhou X. Threshot: an aggressive task scheduling approach in cmp thermal design. *IEEE International Symposium on Performance Analysis of Systems and Software*, April 2009. [cited at p. 12]

[31] Yen K.C. Kumar A. Ramachandran A. Greenhill D. Nawathe U.G., Hassan M. Implementation of an 8-core, 64-thread, power-efficient sparc server on a chip. *IEEE Journal of Solid-State Circuits*, 43(1), Jan 2008. [cited at p. 14]

[32] Nunez A. Bautista T. Flexible design of sparc cores: a quantitative study. *Proceedings of the 7th International Workshop on Hardware/Software Codesign*, pages 43–47, May 1999. [cited at p. 15]

[33] Simply risc s1 core, Feb 2008. Available at http://www.srisc.com. [cited at p. 18]

[34] Opencores hardware design community, Feb 2008. Available at http://www.opencores.org/. [cited at p. 18]

[35] Cortadella J. Cornetta G. Asynchronous pipelined datapaths design techniques. a survey. May 1997. [cited at p. 33]

Appendices

Appendix A

Technical Information

Source Code

Single SPARC core can be downloaded from:

www.srisc.com

OpenSPARC source code can be downloaded from:

www.opensparc.org

Important Files in SRISC

1. This path contains all *.v RTL files of s1 core:

   s1_core/hdl/rtl/

2. This path contains the memory model, and the testbench for s1 core:

   s1_core/hdl/behav/testbench/

3. This is the filelist used for VCS simulation:

   s1_core/hdl/filelist.vcs

4. This file contains the filelist for Design Compiler, in addition to the constraint file:

   s1_core/hdl/filelist.dc

5. The following are the executable files to build/run DC or VCS:

   s1_core/tools/bin/build_dc

   s1_core/tools/bin/build_vcs

   s1_core/tools/bin/run_vcs

6. This file compiles C code to SPARC assembly:

   s1_core/tools/bin/compile_test

7. This file contains the boot code:

   s1_core/tests/boot/boot.image

8. The folder where simulation results get stored:

   s1_core/run/sim/vcs

9. The following are thesis implementation files:

   s1_core/hdl/behav/testbench/CLK_EMU.v

   s1_core/hdl/behav/testbench/PCU.v

   s1_core/hdl/behav/testbench/TEMP_EMU.v

   s1_core/hdl/behav/testbench/Timer.v

   s1_core/hdl/behav/testbench/PCU_top.v

Important Commands

1. To source s1 environment:

   source s1_core/sourceme

2. To build s1 model:

   s1_core/build_vcs

3. To run simulation on vcs:

   s1_core/run_vcs

4. A script that covers all initial steps up to running VCS:

   source s1_core/runme