MODELING AND APPLICATIONS OF CURRENT DYNAMICS IN A COMPLEX PROCESSOR CORE

by

Radu Muresan

A thesis

presented to the University of Waterloo

in fulfillment of the

thesis requirement for the degree of

Doctorate in Philosophy

in

Electrical and Computer Engineering

Waterloo, Ontario, Canada, 2003

Radu Muresan, 2003
Abstract

Measuring and modeling instantaneous current consumption or current dynamics of a processor is important in embedded system designs, wireless communications, low energy mobile computing, security of communications, and reliability. This thesis introduced a new instruction-level based macro-modeling approach for instantaneous current consumption in a complex processor core along with new instantaneous current measurement techniques at the instruction and program level. Current consumption and voltage supply waveforms of a processor core were acquired by a sampling oscilloscope through an external interrupt-based setup. Accurate measurements of current, power and energy consumption at the instruction, block, or program level were obtained from analyzing the stored current and voltage waveforms. The simulated instantaneous current waveform at the program level was generated in two steps. First, a base waveform of the current simulation was generated by the use of four basic current superposition principles applied to current approximating functions at the instruction level. Secondly, a final waveform of the simulated current was generated from the base waveform by applying a factorial adjustment as a function of the instruction parallelism and sequencing. Step-by-step current modeling procedures with numerical examples were presented. The model captured 98% of the variation of the instantaneous current for 6 complex diverse applications, with an average RMS error of less than 2.7% of the average measured mean. Energy estimates obtained by the use of the simulated current waveforms were within 1.9% of the measured values. The current measurement and modeling techniques were applied to analyze the security of applications resistant to power analysis attacks. First, a current smoothing technique was developed in order to reduce the current peaks and the overall current variation generated by an application. Second, an energy analysis technique that used energy distribution comparisons was developed to analyze the security of an implementation against simple power analysis attacks. Third, a power analysis attack resistant architecture capable of dynamically controlling the current variation limits of an application was proposed. This research is important, since for the first time power modeling has been extended to security and for the first time highly accurate instruction-based models of instantaneous current and power for complex processor cores have been developed.
# TABLE OF CONTENTS

LIST OF FIGURES ................................................................................................................................. vi

LIST OF TABLES ...................................................................................................................................... x

GLOSSARY OF TERMS ........................................................................................................................ xiii

Chapter 1: INTRODUCTION ............................................................................................................... 1

1.1 Thesis Goal and Overview ............................................................................................................... 4

Chapter 2: BACKGROUND AND PREVIOUS RESEARCH ................................................................. 7

2.1 Power and Energy Issues in VLSI Systems .................................................................................... 7

2.1.1 Energy Consumption Calculations at the Software Level ...................................................... 10

2.1.2 Previous Power and Energy Modeling Research ...................................................................... 11

2.1.2.1 Previous Related Power Models and Methodologies ........................................................... 12

2.1.2.2 Other Researched Power Models and Methodologies ......................................................... 21

2.2 Idle Low-Power Saving Modes in Specialized Processors ......................................................... 25

2.2.1 Energy Conserving Strategies at the Software Level ................................................................ 27

2.3 Overview of Timing and Power Analysis Attacks in Cryptography ........................................ 29

2.3.1 Introduction to Elliptic Curve Cryptosystems .......................................................................... 34

Chapter 3: THE PROCESSOR AND THE EXPERIMENTAL SETUP .......................................... 38

3.1 The Processor ..................................................................................................................................... 38

3.2 Current Dynamics Measurement Technique in a DSP Processor Core .................................... 41

3.3 Static Measurements of Average Currents for Software Applications .................................... 48

Chapter 4: CURRENT DYNAMICS MODELING METHODOLOGY ................................................. 50

4.1 Problem Definition ............................................................................................................................ 50

4.2 Proposed Superposition Principles for Current Modeling ....................................................... 53
# LIST OF FIGURES

<table>
<thead>
<tr>
<th>Figure Number</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>Figure 2.1</td>
<td>30</td>
</tr>
<tr>
<td>Figure 3.1</td>
<td>39</td>
</tr>
<tr>
<td>Figure 3.2</td>
<td>40</td>
</tr>
<tr>
<td>Figure 3.3</td>
<td>41</td>
</tr>
<tr>
<td>Figure 3.4</td>
<td>42</td>
</tr>
<tr>
<td>Figure 3.5</td>
<td>44</td>
</tr>
<tr>
<td>Figure 3.6</td>
<td>48</td>
</tr>
<tr>
<td>Figure 3.7</td>
<td>49</td>
</tr>
<tr>
<td>Figure 4.1</td>
<td>51</td>
</tr>
<tr>
<td>Figure 4.2</td>
<td>52</td>
</tr>
<tr>
<td>Figure 4.3</td>
<td>55</td>
</tr>
<tr>
<td>Figure 4.4</td>
<td>56</td>
</tr>
<tr>
<td>Figure 4.5</td>
<td>58</td>
</tr>
<tr>
<td>Figure 4.6</td>
<td>60</td>
</tr>
<tr>
<td>Figure 4.7</td>
<td>67</td>
</tr>
<tr>
<td>Figure 4.8</td>
<td>67</td>
</tr>
<tr>
<td>Figure 4.9</td>
<td>71</td>
</tr>
</tbody>
</table>
Figure 4.10 Variable current shape at the instruction level............................................................... 72
Figure 4.11 Variable current component abstraction circuit at the execution set level............... 74
Figure 4.12 Variable current component abstraction circuit at the linear block level .......... 76
Figure 4.13 Graphical representation of the current construction process at the linear block level ......................................................................................................................... 77
Figure 4.14 Current consumption abstraction circuit at the program level............................... 78
Figure 4.15 Current simulation methodology targeting the SC140 DSP processor ............ 80
Figure 5.1 Plots of current measurement waveforms at the instruction level............................... 84
Figure 5.2 SNR histogram....................................................................................................................... 85
Figure 5.3 Example 1 of the data collection at the instruction level............................................... 86
Figure 5.4 Example 2 of the data collection at the instruction level............................................... 86
Figure 5.5 Gamma approximating functions at 100MHz. Real measurements versus simulated functions ......................................................................................................................... 90
Figure 5.6 Gamma approximating functions at 200MHz. Real measurements versus simulated functions ......................................................................................................................... 90
Figure 5.7 Gamma approximating functions at 300MHz. Real measurements versus simulated functions ......................................................................................................................... 91
Figure 5.8 Current simulation at program level at 100MHz, 200MHz, and 300MHz .......... 93
Figure 5.9 Regression line for current ratio to frequency ratio dependency......................... 94
Figure 5.10 Example of model parameter calculation for a short linear block....................... 97
Figure 5.11 Calculating a general solution for the circuit state dependency............................ 100
Figure 5.12 Current base waveforms versus measured waveforms of model input programs .......................................................................................................................... 102
Figure 5.13 Final current simulation waveform versus measured waveform of model input programs ................................................................. 102

Figure 5.14 Current base waveforms versus measured waveforms of test programs ...................................................................................................................... 104

Figure 5.15 Final current simulation waveforms versus measured waveforms of test programs ...................................................................................................................... 105

Figure 5.16 Final current simulation waveform with ideal lambda block values versus measured averaged waveforms ........................................................................................................ 108

Figure 5.17 Simulation waveform based on calculated lambda blocks versus averaged waveform for bubvit.asm .............................................................................................................. 109

Figure 5.18 Simulation waveform based on calculated lambda blocks versus averaged waveform for polymul.asm .............................................................................................................. 110

Figure 5.19 Static simulation versus measured current. Current waveforms measured by looping and wait methods for powerco.asm .............................................................................................................. 115

Figure 5.20 Static simulation versus measured current. Current waveforms measured by looping and wait methods for powercoN.asm .............................................................................................................. 116

Figure 5.21 Static simulation versus measured current. Current waveforms measured by looping and wait methods for bubble.asm .............................................................................................................. 117

Figure 5.22 Static simulation versus measured current. Current waveforms measured by looping and wait methods for viterbi.asm .............................................................................................................. 118

Figure 5.23 Static simulation versus measured current. Current waveforms measured by looping and wait methods for bubvit.asm .............................................................................................................. 119

Figure 6.1 RC circuit ............................................................................................................................................................................. 123
Figure 6.2 Measurement 1 and Measurement 2 applied to bubvit.asm program ....................... 125
Figure 6.3 Current plots. Case 0 versus Case 2 for bubvit.asm ..................................................... 127
Figure 6.4 Current smoothening technique based on Table 6.2 applied to the polymulNIST.asm ............................................................................................................. 129
Figure 6.5 Bit prediction strategy ......................................................................................................... 132
Figure 6.6 Theoretical traces and timing setup for energy analysis methodology ......................... 133
Figure 6.7 SIMULINK block diagram for Stage 2 EASI analysis.................................................. 135
Figure 6.8 Stage 2 energy stair graph for the polymulNIST.asm program ..................................... 138
Figure 6.9 Stage 3 energy stair graph for the polymulNIST.asm program ..................................... 139
Figure 6.10 Stage 5 energy stair graph for the polymulNIST.asm program .................................. 140
Figure 6.11 Example of current traces at Stage 2 ............................................................................. 141
Figure 6.12 Stage 2 energy stair graph for the polymulNISTsmooth.asm program ....................... 142
Figure 6.13 Stage 3 energy stair graph for the polymulNISTsmooth.asm program ...................... 144
Figure 6.14 The smoothened AC current traces for polymulNISTsmooth.asm implementation at Stage 2 ............................................................................................................. 146
Figure 6.15 Example of smoothened current traces at Stage 2 EASI analysis ............................... 147
Figure 6.16 Example of smoothened current traces at Stage 2 EASI analysis ............................... 148
Figure 6.17 PAAR architecture connected to the SC140 DSP processor .................................... 150
Figure 6.18 Current projection for the SC140 executing the polymul.asm application when PAAR architecture is present ................................................................. 151
Figure A.1 Horizontal acquisition window for the TDS 8000 oscilloscope .................................. 170
LIST OF TABLES

<table>
<thead>
<tr>
<th>Table Number</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>Table 2.1 General algorithmic description of EC scalar multiplication, point addition point subtraction, and point negation</td>
<td>37</td>
</tr>
<tr>
<td>Table 3.1 Software synchronization template for the SC140 DSP processor</td>
<td>47</td>
</tr>
<tr>
<td>Table 4.1 Max, mean, NOP-level, energy and $\delta$ measurements for programs at 3 frequencies</td>
<td>57</td>
</tr>
<tr>
<td>Table 4.2 Current variation measurements for different assembly language instructions at 3 different frequencies</td>
<td>57</td>
</tr>
<tr>
<td>Table 4.3 Max and Area measurements supporting the pipelining superposition principle applied to a sequence of 3 instructions</td>
<td>61</td>
</tr>
<tr>
<td>Table 4.4 RMS and Area measurements supporting the operand value and high bank register superposition principle</td>
<td>62</td>
</tr>
<tr>
<td>Table 4.5 RMS, Max, and Area measurements supporting the parallel issue superposition principle</td>
<td>63</td>
</tr>
<tr>
<td>Table 4.6 Measurement analysis for parallelism dependency</td>
<td>68</td>
</tr>
<tr>
<td>Table 5.1 Instruction level data statistics and errors of measurement</td>
<td>83</td>
</tr>
<tr>
<td>Table 5.2 Gamma approximation functions. Errors of simulation</td>
<td>92</td>
</tr>
<tr>
<td>Table 5.3 Max, Mean values at 3 different frequencies. Simulated and measured waveforms</td>
<td>93</td>
</tr>
<tr>
<td>Table 5.4 Relative errors for Max, Mean values plus the standard deviation values of the difference waveform (IR-IS)</td>
<td>94</td>
</tr>
<tr>
<td>Table 5.5 Model parameters for circuit state dependency</td>
<td>98</td>
</tr>
</tbody>
</table>
Table 5.6 Measurements of current mean, maximum, and energy derived from the simulated and measured waveforms ...................................................... 106

Table 5.7 Error values, simulated waveforms versus measured (filtered) waveforms .......... 107

Table 5.8 Static simulation results and relative errors ......................................................................................................................... 120

Table 6.1 ALU equivalence chart for calculating the time discharge constant per program ................................................................................................................................................. 124

Table 6.2 General ALU equivalence chart for smoothening current in SC140 processor ............................................................................................................................................................................................ 127

Table 6.3 Data analysis for smoothening technique applied to bubvit.asm program .......... 128

Table 6.4 Data analysis for smoothening technique applied to polymulNIST.asm program.. 130

Table 6.5 EASI technique algorithm ......................................................................................................................................................................................... 134

Table 6.6 EASI technique for polymulNIST.asm. Stage 2 time and energy analyses, and bit 2 prediction ............................................................................................................................................................................................................................................................ 138

Table 6.7 EASI technique for polymulNIST.asm. Stage 3 time and energy analyses, and bit 3 and bit 4 predictions ............................................................................................................................................................................................................................................................ 139

Table 6.8 EASI technique for polymulNIST.asm. Stage 5 time and energy analyses, and bit 5 prediction ............................................................................................................................................................................................................................................................ 140

Table 6.9 EASI technique for polymulNISTsmooth.asm. Stage 2 time and energy analyses, and bit 2 prediction ............................................................................................................................................................................................................................................................ 143

Table 6.10 EASI technique for polymulNISTsmooth.asm. Stage 3 time and energy analyses, and bit 3 and bit 4 predictions ............................................................................................................................................................................................................................................................ 145

Table 6.11 Measurements for the programs presented in Figure 6.18 .................................... 152

Table B.1 Averaged measurements at the instruction level. AC and DC measurements 171-173
ACKNOWLEDGMENTS

The author wishes to express sincere appreciation to Professor Catherine Gebotys for her support, direction and guidance into the preparation of this thesis. She had the wisdom to motivate me and encourage me during the hardships of this research.

I am also grateful to my wife Ngamta and my children Matthew, Joel, John-Luke, Nolge, Julia Anna, and Heman for their love, understanding and encouragements during this long trial. All of the work that I have put into this research is dedicated to them. I am also thankful to my parents Iulian and Anica for their care and for directing me towards the field of computers.

And not the last but the first I am grateful to my God for being with me and giving me strength to finish this work. I give him honor by quoting here two passages from his word.

“Your heavenly Father already knows all your needs, and he will give you all you need from day to day if you live for him and make the Kingdom of God your primary concern. [Matthew 6:32-33, NLT Bible].”

“This is what the LORD says, he who made the earth, the LORD who formed it and established it--the LORD is his name: ‘Call to me and I will answer you and tell you great and unsearchable things you do not know.’ [Jeremiah 33:2-3, NIV Bible].”
GLOSSARY OF TERMS

**AD-Converter.** Analog-Digital Converter.

**AGU.** Address Generation Unit.

**ALU.** Arithmetic Logic Unit.

**AL.** Assembly Language application.

**CPU.** Central Processing Unit.

**CMOS.** Complementary Metal-Oxide Semiconductor.

**CISC.** Complex Instruction Set Computer.

**DA-Converter.** Digital-Analog Converter.

**DES.** Data Encryption Standard.

**DPA.** Differential Power Analysis.

**DSP.** Digital Signal Processor.

**DMA.** Direct Memory Access.

**EASI.** Energy Analysis for Secure Implementations.

**EC.** Elliptic Curve.

**ESP.** Early design Stage Power.

**EX.** Execution.

**HO-DPA.** High-Order Differential Power Analysis.

**ISR.** Interrupt Service Routine.

**IC.** Integrated Circuit.

**IF.** Instruction Fetch.

**ID.** Instruction Decode.

**IP.** Intellectual Property.

**IPCMOS.** Interlocked Pipeline CMOS.

**IRDB.** Instruction Reference Data Base.

**MT.** Measuring Template.

**MAC.** Multiply ACcumulate Unit.

**Max.** Maximum.

**NAF.** Non-Adjacent Form.

**NIST.** National Institute of Standards and Technology.
NMI. Non-Maskable Interrupt.
PAAR. Power Analysis Attack Resistant
PCU. Program Control Unit.
PLL. Phase-Locked Loop.
PUM. Program Under Measurement.
Pk-Pk. Peak-to-Peak.
RMS. Root-Mean-Square.
RISC. Reduced Instruction Set Computer.
RT-level, RTL. Register-Transfer Level.
SA. Software Application.
SDP. Software Development Platform.
SPA. Simple Power Analysis.
SOC. System-On-Chip.
UART. Universal Asynchronous Receiver Transmitter.
VLIW. Very Large Instruction Word.
WB. Write Back.
Chapter 1

INTRODUCTION

POWER DISSIPATION has become a critical factor in many areas of digital design [1], [2], [3], [4], especially in portable computing and personal communication applications [5]. Power dissipation issues are an important and dominant factor at all levels of embedded systems designs: technology, circuit/logic, architecture, algorithm, system and software. The increased computational demands at the system level have a direct impact on power dissipation at the lower levels. For example, at the technology level the chip area and clock frequencies are constantly increasing but as a result, power dissipation also proportionally increases by the formula [3]:

\[ P = \alpha \cdot \text{area} \cdot f_{\text{clock}} \]  

Today’s technology, affected by the widespread use of wireless communication applications, imposes a continuing need for a reduction in the cost, size and weight of portable equipment, and an increase in battery lifetime and system performance [12]. Unfortunately, these are at odds with the power dissipation factors \( \alpha \), \( \text{area} \), and \( f_{\text{clock}} \). As a result, at the design level, power is affecting the following main factors: battery lifetime through the total energy consumption of the system [5], [12], [39]; electrical limits of the design, through the peak power values [23], [44]; cost, through expensive package technologies needed to dissipate the increased amount of power; and reliability through the increasing high temperatures of chip operation [3], [23], [69], [70], [72].

In order to meet the high performance demands of the software applications, low-power design decisions taken at the technology or circuit level determine many of the architectural characteristics of today’s designs. For example, meeting low-power specifications at the technology level through lowering the voltage supply implies lowering the frequency of operation, with an increase in functional activity through pipelining and parallelism, which in turn impacts algorithm and software designs [3], [5]. Due to the importance of low-power issues and the complex relationship between the elements on the designing ladder of high performance systems, power estimation and
modeling has garnered enormous attention in recent years [1], [3]. Nevertheless, there is little research that addresses the power estimation and modeling problem at the system software level. At this level, a static and a dynamic relationship between the software application and the power consumption of the underlying hardware is encountered. The static relationship is characterized by the average power consumption of the hardware executing the software, and the dynamic relationship is characterized by the instantaneous power consumption of the hardware executing the software. So far, there are a few modeling and power estimation methodologies that address the static relationship between software and hardware [1], [3].

Besides all other power dissipation issues, at all levels of design abstraction, the static and dynamic relationship between software and hardware becomes essential from a system software perspective in the following main areas: embedded system design, wireless communications, energy conserving designs in mobile systems and portable computing, security of communication, and reliability. Embedded system designs must incorporate the specific software application so that all power specifications are met [1], [6], [7]. Wireless communication implies reliance on portable power sources such as batteries. As a result, energy conservation issues become essential. In this area, energy saving techniques that directly address the electronic part of a system must be complemented with appropriate energy saving techniques at the software level. Energy conservation issues are becoming increasingly important in the mobile wireless research area [8], and more, and more topics that address “power-aware” [9], “energy-aware” [10], or “power-conscious systems” [11] are encountered in recent technical journals. In general, a power or energy-aware or conscious system is a system whose design has been sensitive to energy considerations [11], [12]. Energy efficient designs for mobile systems and portable computing affect factors like functional utility, size and weight, maintenance of the system, and environmental factors due to the high pollutant characteristic of the batteries. The improvement in battery technology is slow [12], [13]. As reported in [10] and [71], a notebook’s microprocessor consumes up to one-third of the total energy. Therefore, energy consumption at the processor level is an important factor that has to be taken into account for increased usability and functionality of mobile systems [8] and portable computing [13].
In wireless communication systems, besides the power consumption factor, the encryption of audio and data before transmission becomes very important. Not only must encryption algorithms have high performance and low power, but more importantly they must be secure from power analysis attacks [65]. For example, power analysis attacks involve utilizing power traces (or dynamic power measurements over time) of a device to determine the users’ secret key [15], [16], [17], [18].

Finally, in the race for performance, the software implementation of an application, especially for processors with parallel computational features, can cause high peak current consumptions throughout the running application, due to grouping or sequential ordering of instructions with high current consumption. The ordering and grouping of instructions could possibly increase performance, but over time the development of high current consumption peaks can be detrimental to the processor. Excessive power dissipation in an integrated circuit (IC) diminishes the reliability of the system by creating peak powers and high levels of chip temperatures [2], [3], [69], [70], [72]. It is important, from this point of view, to be able to dynamically analyze the power consumption effects of the parallel constructs of embedded software applications.

All of the above problems imply the need for a current, power, and energy estimation methodology capable of conveying static and dynamic information that characterizes the software/hardware relationship of high performance systems. The development of accurate methods for static and dynamic current measurement, analysis, and simulation would be beneficial to any software tool targeting power and energy consumption for optimization, security, and reliability. Most of the power estimating techniques developed so far are based upon simple average-based measuring techniques. Average power consumptions per instructions measured by simple AC ammeters are the base foundation of these models. These instruments, if not equipped with true-RMS capability, will produce erroneous reading for other types of waveforms [64]. The current consumption of a processor at the instruction and the program level is a dynamic signal [19], [21] and as a result, there is a need for accurate measuring techniques that can calculate true RMS, Max, and Area values.
1.1 Thesis Goal and Overview

The goal of this research was three-fold. The first goal was to establish current measuring techniques at the software level, capable of capturing accurate instantaneous current consumption of software executing on a complex DSP processor core [19]. These measuring techniques had to provide true RMS, Max, and Area values of the current consumption at the instruction and the program level such that the effectiveness of the modeling techniques could be properly assessed. The second goal was to develop a methodology for current, power, and energy simulation at the system-software level that addressed both the static and dynamic component of software to hardware relationship [20], [21], [73]. For the first time, an instruction-level model of current dynamics was introduced and validated with various software applications at different working frequencies. The third goal was to apply the current methodology to analysis of security and reliability. Applications of energy analysis to security were demonstrated. Processor core security extras for software and hardware techniques of controlling the current levels of an application were also presented.

The thesis is organized into 7 chapters. Chapters 1 and 2 present the introduction, the background and the previous research in the area of measurement and modeling of the current, power, and energy. Chapter 2 also introduces security issues related to power. Chapters 3, 4, 5, 6 and 7 describe the new instruction-level models for current plus the security applications. Next, a short overview of the individual chapters of this thesis is presented.

Chapter 1 presents the introduction to this thesis. Chapter 2 Section 2.1 and Subsection 2.1.1 introduce issues related to power and energy consumption in CMOS type processors. The basic energy consumption formulas and the basic characteristics of common power saving techniques are introduced. Subsection 2.1.2 presents an extended overview of the previous research related to power and energy measurements and modeling in microprocessor based systems. Power related issues that address different levels of abstraction are developed. Section 2.2 presents some basic facts related to idle low-power saving modes in specialized processors. These modes are considered important in any power optimization technique. Subsection 2.2.1 presents a few facts related to energy conservation strategies in portable computing. Section 2.3 and its subsections present a short overview of power analysis attacks in cryptographic systems with special emphasis
on elliptic curve cryptosystems. This addresses the security issue of a microprocessor based cryptosystem and its relationship to the dynamic power consumption.

Chapter 3 introduces the experimental setup for current dynamics measurements. Section 3.1 describes the main features of the target processor used in this research. Section 3.2 presents the technique used for measuring instantaneous current consumption in the processor while executing a software application. This technique formed the basic measuring tool necessary in the development of the modeling of current dynamics. Section 3.3 presents the methodology used for measuring average current consumption in a processor by using a common ammeter.

Chapter 4 presents the instantaneous current macro-modeling methodology, the main focus of this thesis. Section 4.1 introduces the current dynamics modeling problem from a software level perspective. Section 4.2 and its subsections introduce the principles of instruction-level simulation for current dynamics based on four superposition principles: the total current superposition principle, the pipeline superposition principle, the random operand value and high bank register superposition principle, and the parallel issue superposition principle. Section 4.3 presents an analysis of the circuit state dependency of the processor on parallelism composition and on the execution set sequencing. Section 4.4 presents a systematic development of the abstract circuitry at the processor level used in the modeling process of the current dynamics. This section also presents the mathematical development of the model and elementary approximating functions. Subsection 4.4.1 describes the current dynamics simulation methodology for the processor applied at the program and the instruction level.

Chapter 5 demonstrates the experimental modeling results using gamma functions. Section 5.1 introduces the definitions of the errors and data statistics reported throughout this thesis and presents current measurement analysis at the instruction level. Section 5.2 presents examples of building gamma approximating functions at 100MHz, 200MHz and 300MHz. These gamma functions were probed at the 3 frequencies with short simulations at the program level. Section 5.3 presents a numerical step-by-step current modeling setup at 200MHz. The current dynamics macro-model was validated on six assembly language applications. In Section 5.4, a previous researched power model [6] is applied to the target processor and comparison results of current measurements and simulation are presented.
Chapter 6 introduces some immediate applications of the current measurement technique to security and reliability. In Section 6.1, results of smoothening the current trace of the cryptographic programs are presented. In Section 6.2, a new energy analysis technique is presented to assess the security of the implementations against simple power analysis attacks. Section 6.3 introduces a power analysis attack resistant (PAAR) architecture that could be added to an existing processor’s architecture in order to increase its security.

Finally, Chapter 7 outlines the conclusions of this thesis and states the thesis’s contributions to the state of the art. The next chapter outlines the background and the previous research in power modeling and security.
Chapter 2

BACKGROUND AND PREVIOUS RESEARCH

This chapter presents a review of background and previous research in the measurement and modeling of current, power, and energy in microprocessor based systems. This chapter also reviews cryptographic systems related to the current dynamics behavior of microprocessor based systems.

2.1 Power and Energy Issues in VLSI Systems

The following expression estimates the sources of power dissipation in CMOS devices [1], [13]:

\[
P = \frac{1}{2} \cdot C \cdot V_{DD}^2 \cdot f \cdot N + Q_{SC} \cdot V_{DD} \cdot f \cdot N + I_{\text{leak}} \cdot V_{DD}
\]

where:

- \( C \) - represents the node capacitance.
- \( N \) - represents the number of gate output transitions per clock cycle as a characteristic of the switching activity.
- \( V_{DD} \) - represents the voltage supply (constant).
- \( f \) - represents the frequency of operation.
- \( Q_{SC} \) - represents the quantity of charge per transition carried by the short-circuit current.
- \( I_{\text{leak}} \) - represents the leakage current that is determined by the fabrication technology.

The first term of Equation 2.1 characterizes the switching power of the CMOS device and quantifies the power needed for charging and discharging the circuit nodes [1]. The second term characterizes the short-circuit power of the CMOS device and quantifies the power dissipation during output transitions due to current flowing from the power supply to ground [1]. The third term represents the short-circuit power and leakage current power of the CMOS device and
quantifies the static power dissipation due to reverse bias current and sub-threshold transistor currents [1].

Various power estimating techniques in VLSI circuits are surveyed in [2]. The paper states that the simulation-based techniques at the circuit level are strongly pattern-dependent and simulating large number of input patterns for large circuits is practically impossible [2]. The more efficient power models make general simplifications as related to estimating the current consumption of the sub-circuits, power consumed by latches and combinational logic, and the importance of the switching power over the short-circuit power [2]. The probabilistic techniques, statistical techniques, and the sequential techniques, surveyed in [2], purpose to reduce the strong pattern-dependence problem and were considered weakly pattern-dependent.

The first term of Equation 2.1 is the predominant term in the total power consumption $P$ of the device, and it accounted for more than 90% of the total power dissipation in [13]. Thus most of the power optimization techniques and the power estimation models developed targeted the switching activity of the device [1], [2], [3], [5], [13], [22], [23]. Power modeling and estimation had been developed mainly as a tool for power optimization techniques. The following provides a short overview of the most significant optimization techniques that addressed four levels of abstraction of the VLSI system design [1], [2], [3], [5], [13]:

1. **Circuit Level.** Techniques at this level reduce switching activity power by targeting individual logic-gate and transistor-level combinational circuit. These techniques include:

   - **Complex Gate Design.** This approach reduces power and gate propagation delay by a process of choice ordering of gate inputs [1].

   - **Transistor Sizing.** Finding an appropriate transistor sizing minimized power dissipation for a given delay constraint [1].

2. **Logic Level.** Techniques at this level reduce switching activity power by targeting logic-level combinational and sequential circuit. These techniques include:

   - **Combinational logic optimization.** The techniques in this class are decomposed into two groups: a technology-independent level that works at the logic equation level and a technology-dependent one that maps the optimized logic equations into a target library of different
transistor size-based implementations, so that cost and power requirement constraints are met. Common technology-independent techniques include don’t-care optimization, path balancing, and factorization [1].

• **Sequential circuit optimization.** The techniques in this class operate at two levels of abstraction: the state transition graph level and the logic-gate and flip-flop level. A well known method that works at the first level is the **state encoding** for minimal area [1]. **Retiming** is an optimization method that repositions the flip-flops in a synchronous circuit such that the overall switching activity of the flip-flops is minimized [1]. **Clock gating** is a method using clock activation signals to enable clocks to registers and flip-flops only during active periods. The clock gating method can be used to turn-off or power-down functional units when they are not active [1]. **Pre-computation** is an extension of clock gating and it detects earlier in a sequential circuit the possibility of idling sub-circuits for the coming states [1].

3. **Architecture level.** There are two main classes of optimization techniques at this level: “architecture level power analysis” and “power optimizations in behavioral synthesis”.

• **Architecture level power analysis.** Techniques in this class perform direct power analysis by converting the high level design into a gate, circuit, or physical description where low-level power analysis tools can be used. Acceptable power models at this level can be built if the final lower level characteristics (circuit style, module and gate library, etc.) are fixed or at least restricted [1]. The power at this level is modeled by using various approaches [2]: estimating the switching capacitance per module, estimating the switching activity per module by using known input signal statistics or input signal statistics [2] or random input streams, and assigning average power costs to individual modules. Besides these general models, there are also many specialized models that target specialized architectures or functional blocks.

• **Power optimization in behavioral synthesis.** Techniques in this class estimate power and perform optimization of power by analyzing the mapping of a high-level design into a register-level design. The power optimizations at the register level are aimed at reducing the number of control steps, slowing down the clocks in order to support a reduced voltage supply [1], and other techniques.
4. System and software level. Instruction-level average power simulations had been developed based on average current measurements performed on individual/pair instructions or programs. The power measuring techniques apply in general to specific hardware implementations running program applications, but the approach can also be used for architecture-level simulators. Power modeling at the software level made the development of energy optimizations possible [1], [3], [6].

2.1.1 Energy Consumption Calculations at the Software Level

In this section a software application AL is considered. The execution start time of the application is \( t_0 \) and the execution lasts for a period of \( T \) seconds. At higher levels the average energy consumption of the processor while executing the application AL can be estimated in different ways:

1. When the processor’s voltage supply \( V_{DD} \), the average current consumption \( I \) per program, the total number of clock cycles \( N \) for the application AL, and the clock period \( \tau \) are known, the average energy consumption per application can be determined by the following relation (where \( T = N \times \tau \)) [6]:

\[
E_{\text{average/program}} = V_{dd} \times I \times T
\]

2. Based on [10], when further assumptions related to Equation (2.1) are made, the average power and energy consumption for the application AL can be expressed as:

\[
P_{\text{tot}} = P_{\text{sw}} + P_{\text{sta}} = C_t V_{dd}^2 f + V_{dd} I_{\text{leak}}
\]

\[
E_{\text{tot}} = P_{\text{tot}} T = C_{\text{tot}} V_{dd}^2 + V_{dd} I_{\text{leak}} T
\]

In Equation (2.3), \( P_{\text{tot}} \) is the total power (the switching \( P_{\text{sw}} \) plus the static \( P_{\text{sta}} \)), \( C_t \) is the total average capacitance being switched by the executing program per clock cycle, \( f \) is the operating frequency, \( I_{\text{leak}} \) is the leakage current, and \( C_{\text{tot}} \) is the total capacitance switched by the executing program [10].
3. Based on the instantaneous power function \( p(t) \) in an electric load, a general relation for calculating energy consumption per program application can be determined. The instantaneous power in a load and the energy consumption over a period of time \( T \) are:

\[
p(t) = v(t) \cdot i(t); \quad \text{[Watts]}
\]

\[
E_{\text{load}} = \int_{0}^{T} p(t) dt; \quad \text{[Joules]}
\]

In Equation (2.4), \( v(t) \) represents the time dependent function of the voltage applied to the load (measured in Volts), and \( i(t) \) represents the time dependent function of the current passing through the load (measured in Amperes). The voltage supply \( V_{DD} \) of a processor could be considered a DC constant voltage. As a result, power models in a processor are linearly dependent on current models. The only time dependent component in calculating the power and energy consumption of the processor is the current function \( i(t) \). It is assumed that the current function \( i(t) \) in a processor executing the application AL can be measured. As a result, the energy consumption of a processor per program is:

\[
E_{\text{program}} = V_{DD} \cdot \int_{i_0}^{T + i_0} i(t) dt
\]

The parameters \( C_{\text{tot}} \) and \( I_{\text{leak}} \) of Equation (2.3) can be approximated using modeling techniques [10], which is difficult to do. The current parameters \( I \) and \( i(t) \), and the execution time \( T \) for an application can be measured. As a result, Equations (2.2) and (2.5) are considered more practical and can be used more easily in assessing the energy consumption at the program level.

2.1.2 Previous Power and Energy Modeling Research

In this section, previous research results in the area of power and energy modeling and measurements for processors are presented. First, power models including their verification, which are relevant to this thesis work, are presented. Secondly, previous research of other power models at different levels of abstraction is presented.
2.1.2.1 Previous Related Power Models and Methodologies

In [6], the authors presented an instruction level based energy modeling approach to software in embedded systems. The results presented in [6] targeted the Intel 486DX2 processor and the results presented in [24] targeted the Fujitsu SPARClite 934 processor. The experiments for both processors were done at an operational frequency of 40 MHz. The Intel 486DX2 processor is a CISC-type architecture based on a micro-architecture technique with 5 stages of pipelining [6] and parallel stages [67]. This processor has an 8Kb on-chip L1 cache and some versions have power saving and system power managing capabilities [67]. The SPARClite ‘934 is a CMOS type processor with one on-chip floating unit, on-chip FIFOs, 8k bit on-chip cache, 136 registers, and software controlled power management features [24]. The energy estimation model was based on current measurements at the instruction and the program level. The model estimated the average energy consumption of a program based on Equation (2.2). The measurements of current consumption per program and instruction were done by constructing infinite loops of instances of a program or an instruction. The current measurements were performed separately on the CPU and the DRAM subsystem. The equipment used to perform these measurements was a standard dual-slope integrating digital ammeter with a current averaging window of 100 ms. In order to obtain stable measurements, the loop instances had to execute for much smaller time than the width of this window. The execution time of the programs was measured by a logic analyzer that detected specific bus states. The instruction level model was based on two tables of current costs: base current costs and inter-instruction current costs (overhead costs). The base costs were the average current measurements of individual instructions and the overhead costs were the differences between average current measurements for paired instructions and the corresponding individual base costs. The overhead costs represented the measures of the circuit state changes between instructions. The energy estimation of a program was calculated based on these two tables as a sum of the total base costs and the total overhead costs of all instructions executed. The paper reported that the base energy cost per instruction was not significantly affected by the register names used by the instruction or by the values of the operands handled by the instruction. Nevertheless, it was observed that instructions had smaller energy consumptions when the operands had an increased number of bits set to 1. It was also reported that instructions having memory operands showed a variation in the base current cost dependent on the memory address.
values. All of these effects in the base current values per instructions were taken into consideration using averaged measurements. The power estimation model introduced in this paper considered the following inter-instruction effects:

- **Effect of circuit state** – this effect considered that the input state that determined the switching activity in a circuit was dependent on the previous state of the circuit. For the Intel processor used, this effect was modeled by a constant value of 15mA that was added to the average base current of the program. It was also reported that there was a small impact on the circuit state effect due to the switching activity that occurred on address and data lines. The effect supposedly increased when back-to-back data reads/writes to caches were present.

- **Effect of resource constraints** – this effect was considered to be due to the pipeline activity. In general, the resource constraints led to pipeline stalls that increased the time execution of the program. Experimentally fixed energy averaged values for each type of stalls were determined and used as an adjustment to the base cost value of the program.

- **Effect of cache miss** – this effect was considered to be caused by the instruction sequencing. The cycle times reported in the processor’s reference manuals assumed cache hits. Whenever a cache miss occurred, a cycle penalty had to be considered. An experimental current value of 216mA/cycle was determined by using a cache simulator. This value was multiplied by the energy penalty cycles to estimate the energy penalty for one miss. The average penalty multiplied by the cache miss rate was added to the base cost of the program.

Large assembly language programs, when simulated, were split into basic blocks where a basic block was defined as a contiguous section of code with exactly one entry point and one exit point. Base costs for each basic block were calculated by adding up the base costs of the individual instructions in that block. The overhead adjustments due to the pipeline stalls were estimated per block and added to the basic block cost. Through a program profiler, the block composition of the path execution of the program was determined. Knowing the number of times that each block was executed in a program, the energy per program was determined by using the base cost energy values per blocks. The circuit state overhead adjustment was added to the above overall sum. For the final estimated energy consumption per program, an estimated cache penalty cost was added. In [6], the authors also presented a brief consideration of an energy modeling approach for the
memory system that considered the effects of the memory locations and address sequencing. Also, an example of an energy estimation framework for programs was presented. The energy estimation framework was verified for an example program that executed for approximately 72 clock cycles. It was reported that by using this estimation framework, the maximum difference between the estimated and measured energy costs for programs with no stalls and cache misses was less than 3% of the measured costs. Energy optimization methods that targeted reducing memory operands and execution time were illustrated by two program examples. The un-optimized versions of these programs executed for approximately 440 clock cycles, and 287 clock cycles, respectively.

In [7], the authors applied the instruction level modeling technique introduced in [6] to a Fujitsu CMOS DSP processor. This is a DSP processor with 2 pipeline stages, 4 data registers, 8 index registers, one MAC (Multiply ACCumulate) unit, and 2 on-chip memory banks. The parallel capability of the processor was limited to packed instruction execution (one ALU type instruction and a data transfer instruction). The authors proposed an instruction-level power analysis model for this DSP processor that was used to analyze various energy saving techniques applied at the software level. The model used base cost values, overhead values of pair instructions and overhead values caused by nonadjacent multiply instructions. The reduction of the table's size, due to a large instruction set architecture, was done by grouping the processor’s principal instructions into classes. The power model was validated with a program example that executed for 66 clock cycles. For this example, the average current was underestimated by 10.5%. All of the experimental values for modeling and comparison were obtained by the use of an ammeter. Based on power modeling analysis, the authors proposed energy optimization techniques applicable to the Fujitsu processor. These techniques were verified on four DSP specific applications.

In [25], the authors presented a power estimation modeling approach using energy estimations at the instruction level based on the pipeline stage composition of the architecture. A five-stage (S={IF, ID, RR, EX, WB}) load/store architecture was considered. The energy consumption of an instruction \( w_n \) that was part of a program trace \( W \) was assumed to be dependent on the type, the preceding instruction \( w_{n-1} \), and the instruction set \( w_k \) present in the pipeline. The energy consumption per instruction was approximated by a sum of the energies that characterized the individual stages of the processor:
\[ E(w_n) \geq \sum_{s \in S} U_s(w_n | w_{n-1}) + T_s(w_n) \] (2.6)

where \( U_s \) was the energy consumption per stage in absence of hazards and exceptions (assumption of 1 clock cycle per stage), and \( T_s \) was the incremental energy needed due to the presence of hazards (assumption of extra clock stalls, latency cycles). Furthermore, the \( T_s \) factor was expressed by the following relation:

\[ T_s(w_n) \approx m_s(w_n) \cdot p_s(w_n) \cdot S_s(w_n) \] (2.7)

where \( m_s \) was the typical number of stall/latency cycles while executing instruction \( w_n \), \( p_s \) was the stall/latency probability while executing \( w_n \), and \( S_s \) was the stage energy consumption per each stall/latency cycle. The authors assumed that the individual energy costs \( U_s \) and \( S_s \) per stage could be derived from simulators. Using the approximating Equations (2.6) and (2.7), the authors proposed individual approximating expressions for each individual pipeline stage considered. It was mentioned that the complexity of the spatial and temporal correlation of the instructions in the execution trace \( W \) made the estimation process of energy consumption per instruction a complex task. Also, the inter-correlation parameter \( U_s(w_n | w_{n-1}) \) had to be maintained as a bi-dimensional array with dimensions proportional to the number of instructions that compose the ISA (Instruction Set Architecture). Based on this energy estimation approach at the instruction/stage level, the authors proposed to apply the methodology to VLIW processor architectures by using a so called principle of “separability” at the functional unit level. Real separability was supposed to be applicable to the functional units and thus to the execution stage. The other portion of the processor, composed of the other stages (non-functional), was considered non-separable. In the energy estimation model at the processor level, the non-separable stages were estimated by averaged values. The execution stage was considered to be separable and the energy consumption was estimated as a sum of the individual partitions, where one partition was executing a particular stream of operations. This model was validated using Synopsys simulation tools through a simplified architecture that implemented only a set of three instructions: \{NOP, ADD, MUL\}. The estimated energy values per instructions generated by the model were compared with the estimated energy values obtained by using Synopsys Design Power at a
simulated frequency of 50MHz. The maximum estimated error was 12.62% and it was for the ADD instruction.

In [26], the authors presented a “constant parameter” type model for power estimation in a 32-bit embedded processor. The energy consumption per program was calculated as:

\[ E = T \times P_{ave} = \frac{T}{N} \sum_{i=1}^{N} P_{\tau,ave}(i) \]  

(2.8)

where \( T \) was the execution time, \( P_{ave} \) was the average power consumption of the program and \( P_{\tau,ave} \) was the average power during each clock cycle. The model proposed to use Equation (2.8) as an energy consumption estimate per program with \( P_{ave} \) of the program equal with a constant parameter. The constant parameter was calculated as the median value of the average power measurements at the instruction level performed across the ISA of the processor. The average power per instruction was determined experimentally using the i960 Cyclone Evaluation Platform that could accommodate different module versions of the processor’s family. The real power measurements were performed by means of a digital sampling scope that measured voltage drops across a precision resistor inserted in the processor’s power supply line. The measurements used the looping technique introduced in [6]. The execution time of the loop and its start/end points were determined by the use of a logic analyzer that detected the presence of specific start/end address values on the address bus. The effects of different source/destination registers, operand values, and conditional codes on power consumption were investigated and it was found that the power variation for these cases was less than 6%. For the processor considered, the authors also presented experimental results that supported a linear relationship between power and frequency of operation. It was reported in the paper that the power estimation values obtained by this model were within 8% accuracy. The accuracy of the model was demonstrated for two implementations of the i960 Embedded Processor family.

In [27], the authors explored some techniques for energy models at the instruction-level in DSP processors. The models and the results presented in this paper were based on a simulated DSP processor. The authors designed part of the Motorola DSP56k instruction set using a Verilog description and they synthesized the design to a standard-cell layout level using Synopsis and Cascade Epoch. A so-called “Mynoch” power estimator was used to perform all of the power
measurements at the instruction and the program level. Mynoch was a gate-level based simulation tool that estimated power by counting transitions from a Verilog simulation and calculated the dynamic energy consumed for each transition using the energy formula $E = 0.5 \cdot C \cdot V^2$. A regression model augmentation was used to compensate for differences between Spice-based simulation results and Mynoch results. As indicated in the paper, the power estimation error of Mynoch simulator versus Avanti’s Star-Sim simulator was due mainly to the fact that the Mynoch estimator did not account for the “intra-gate capacitance” within the registers. The power measurements reported did not include the power consumed by the pads and the memory. Using the DSP Synopsys simulation and the Mynoch power estimator, the authors presented results of four types of instruction-level models: base model [6], pair model [6], NOP model, and general model. The base modeling approach used only a base table of energy measurements per individual instructions and did not account for the inter-instruction effects introduced in [6]. The power estimating errors of this model was found to be high. The pair model used a base table of energy consumption per instruction plus a reduced pair table of averaged energy costs for classes of instructions. In this case, the simulation results were improved. The NOP model was introduced in [27] in order to reduce the difficulties of generating the pair table. The model used a base table of energy costs per instruction plus a pair table that recorded only the overhead energies of the instruction in regards to the NOP instruction. The idea of the NOP model supposed that an instruction should not depend strongly on the neighboring instructions. However, a dependency on whether the neighboring instructions were the same or different was supported. The general instruction model was built based on the architectural structure of the simulated processor used in these experiments, and consisted of four separated tables that corresponded to the main functional units: ALU, AGU, PCU, and others. The main reason of the functional unit separation was to model packed instructions. The paper also indicated that there was a data dependency of the energy estimation that must be taken into consideration by the instruction-level models. The instruction-level model predicted power estimation results per program within 8% percent of simulated-based estimates. The instruction-level power tables used in the estimation model and the power measurements per programs were based on the Mynoch power estimator reports.

In [30], the authors presented a methodology based on black-box power models that estimated average power and cycle-by-cycle energy consumption. An embedded system composed of a
microprocessor with two levels of cache, off-chip memory, and DC-DC converter connected with the interconnect was simulated at the system level. The application software was loaded into an ARM based evaluation board. Using the ARMulator’s cycle execution information, a cycle-by-cycle energy was calculated. The total energy consumption of the system per cycle was calculated as a sum of the component energy consumptions: 

\[ E_{\text{Cycle}} = E_{\text{Proc.}} + E_{\text{Interc.}} + E_{\text{Mem.}} + E_{\text{DC/DC}} + E_{\text{L2Cache}} \]

The power estimation for the processor was based on two power states: active and idle. The active state occurred when the processor used the on-chip cache and the idle state occurred when there was an on-chip code miss and the processor stalled, executing NOPs. The energy consumption per active and idle cycle was calculated based on the following formulas:

\[ E_{\text{Processor,active}} = \frac{P_m}{V_m^2 f_m} \cdot V_{cc} \quad \text{and} \quad E_{\text{Processor,NOP}} = \frac{P_{\text{proc.,NOP}}}{V_m^2 f_m} \cdot V_{cc}, \]

where \( P_m \) represented the active power specified in the processor’s data sheet measured at a given voltage \( V_m \) and frequency of operation \( f_m \). The power consumption \( P_{\text{proc.,NOP}} \) must be measured at the specified \( V_m \) and \( f_m \) values and it represented the power consumed by the processor core while executing NOPs. Similarly, other functional blocks were modeled individually based on data sheets and technical specifications available. The energy for the L2 cache module was estimated in a similar way with the processor by using the active and idle state approach. The energy consumption for the interconnect and the pins was calculated based on capacitance per unit length of the interconnect and on the number of lines switched during the cycle. The energy estimation results were validated with measurements performed on a prototype implementation. The average current consumed by the processor and the total current drawn from the battery were measured with digital multimeters. The execution time was measured using the processor’s timer. The energy simulation values were found to be within 5% of the physical measurements.

In [31], the authors introduced a high-level power estimation application “RTPow” that used RT-level information from a top-down architectural description. The power consumption was calculated as a function of the switching activity estimated from the complexity of the functional components. The “RTPow” simulator could operate as a static power analyzer at RT-level or it could work together with a cycle simulator to generate cycle power profiling from a cycle accurate stimuli file. The “RTPow” simulator was built on top of the Synopsys’ software tools (Design
Compiler and Design Power) and the estimation results required a numerical tuning in order to
guarantee a good correlation with the target technology.

In [32], the authors presented a system-level simulation framework capable of providing power
consumption contributions for the components of the embedded system while running a software
application. The system was considered to be composed of the following functional units: VLIW
core, register file and cache. The power analysis environment was built as a hierarchy of estimators.
The lowest level estimates were provided by commercial power estimation tools working at the
circuit and gate level. A Verilog description of the VLIW core was used to generate RT-level
power estimates through the “RTPow” simulator presented in [31]. The power estimation results
obtained from the “RTPow” simulator formed the data reference needed to build an instruction-
level power simulation engine that worked together with an instruction set simulator. The software
application was interpreted by the instruction set simulator by profiling each instruction on the
main architectural components. The instruction level power estimation engine was capable of
providing power estimate per cycle called “instantaneous power reports” and “time-averaged
power reports”. In this paper, a very long instruction was called a bundle and instructions within a
bundle were called syllables. For a stream of bundles, the average energy consumption per bundle
$w_n$ and the average power associated with the whole stream were approximated by the following
expressions:

$$E(w_n) = B + \alpha_n \cdot c_{syl} + m \cdot p \cdot S + l \cdot q \cdot M$$
$$P(\text{stream}) = (1 - f_s - f_M)(B + \bar{\alpha} \cdot c_{syl} + f_s \cdot S + f_M \cdot M) / T_c$$

(2.9)

where $B$ was the average energy base cost, $\alpha_n$ was the number of syllables per bundle different
than NOPs, $c_{syl}$ was the average energy consumption associate with the execution of a syllable, $m$
was the additional number of cycles due to D-cache miss, $p$ was the probability per bundle that a
D-cache miss takes place, $S$ was the core energy consumption during a pipeline stall, $l$ was the
average additional number of NOP instructions due to I-cache miss, $q$ was the probability that an
I-cache miss takes place, $M$ was the energy consumption of the core during an I-cache miss, $T_c$
was the clock period, $f_s$ was the processor time spent in stalling the pipeline, $f_M$ was the
processor time during an I-cache miss, $\bar{\alpha}$ was the average number of syllables per bundle that
were not NOPs. The model was validated with a scalable multi-cluster architecture (one to four clusters). A single cluster was a 4-issue VLIW core composed of four 32-bit integer ALUs, two 16x32 multipliers, one load/store unit and one branch unit. The RTL macro-model had been validated against gate-level simulation on selected benchmark programs and there were no physical power measurements performed. The estimation errors for the processor core were within 10% for the Max power values, 0.74% for the Average power values, and 4.1% for the RMS values.

In [36], the authors argued that in the process of modeling the power consumption in a SOC by summing up the power consumption of the SOC’s components, the effects of the components’ interaction was not modeled. The paper introduced an estimation engine capable of profiling the percentage of time and energy spent by a software application in various functional units of the core. The energy profiling methodology was applied to the core architecture introduced in [30] and various code optimization techniques for energy saving at software level (algorithmic, data, instruction flow) were considered. The energy profiler was based on the cycle-accurate energy consumption simulation introduced in [30], which were within 5% of average physical measurements performed on a prototype implementation.

In [37], the authors presented a data acquisition approach for power consumption at the instruction cycle level in commercial evaluation boards. The authors argued that the software under investigation that targeted embedded systems usually contained control structures that depended on input data and as a result, there could be a possibility of missing the energy evaluation of segments within this software. As a result, a “segment-wise” time execution and power consumption evaluation approach was proposed. All program segments were marked by trigger points and were recognized by a logic state analyzer connected to the processor’s bus. The experimental setup was composed of a SPARClite evaluation kit, a shunt resistor, a differential amplifier, an AD-converter, a clock generator, and a logic state analyzer. Power consumption was measured by amplifying and integrating the voltage drop on the shunt resistor. The shunt resistor was inserted in the power line of the processor and it was noted that it could influence the functionality of the processor. The logic analyzer triggered the measurement when a trigger point was encountered and the power integrator started delivering power measurements per cycle to the logic analyzer. The setup was not suitable for measuring power consumptions per instruction and
its frequency of operation was limited due to the slow time response of the AD-converter. A power measurement case study was presented for the bubble sort algorithm. Data input values were considered such that the total execution time of the implementation was 15625ns at 40MHz, which was 625 clock cycles. The total power measured for this application was 12459 nW. Results of static power measurements of 3 program segments were presented. The program segments analyzed had execution times under 737 clock cycles. The execution times and power measurements obtained by using the SPARCite evaluation kit were compared with results obtained by using a Strong ARM based simulator. The relative error between the measurements presented was approximately 17%.

In [14], the authors developed power prediction models for embedded DSP processors. Empirical power prediction models were derived from DSP benchmark code by using predictor variables. The predictor variables used real power measurements (meter) of code that executed on different input data and applied various code optimizations. The predictor variables reflected the switching activity at the instruction level (average switching in instruction register, data address bus, and program address bus), and the parallel structure of the DSP processors (average number of multiplier units, subtract units, and load units). The power prediction model was applied to two DSP processors and the power estimates were within 4% of the meter measurements.

In [38], the authors argued that accurate software running time and power analyses were keys to optimization in system synthesis. The execution cost of the software processes was contextually dependent and was generally impacted by the path execution of the program. As a result, the formal analysis of the embedded software was based on intervals rather than values. In these power and execution time analyses the program was divided in basic blocks and then it was represented as a directed program flow graph. Energy analysis results of a process that reads a data packet and loads a picture were presented. It was reported that process timing and power consumption of software applications was highly context dependent.

2.1.2.2 Other Researched Power Models and Methodologies

In [23], the authors presented a macro-modeling technique for estimating the energy-per-cycle dissipation in a logic circuit. The authors indicated that many applications at various level of
abstraction require knowledge of the instantaneous power dissipation as a time-waveform (transient power waveform). Some examples given were analysis of the power and ground bus networks, noise analysis, and power transients in power-down or sleep modes. It was indicated that the modeling of instantaneous power at the logic block level was difficult due to the fact that the number of vector stimulus and the variability in the shape of all different waveforms were large. As a result, the authors presented a simpler problem that was to estimate the energy-per-cycle dissipated in a circuit that was due to a given vector pair. The model introduced, classified the input vector pairs based on their Hamming distances and used a different equation-based macro-model for each Hamming distance. The results used 10 of the ISCAS-85 benchmark circuits, a 16-bit ripple carry adder, and a 10x10 multiplier. The maximum and minimum times to construct the macro-model on a SUN system were 11.73 hours, and 1.56 hours, respectively. The average estimating error for energy-per-cycle was reported to be under 20% and for the long-term average the power was reported to be under 10%.

In [28] and [29], the authors developed a power simulator called ESP that could be used in early design stages of a CMOS RISC type processor. The power estimator model used by the simulator was based on RISC type architecture with static current consumption values per each functional unit. The functional units types considered were: instruction cache, incremener, branch adder, register file, address calculator, ALU, shifter, data cache, and multiply-accumulate. The ESP executed the object code of a specific application and during a pipeline development determined what functional units were active at every stage (clock cycle). The current consumption per cycle was determined by adding the current consumption of all of the functional units that were active at that specific clock cycle. The results obtained by this power simulator indicated that the instruction cache, data cache, and register file units dominated the total power estimation. As a result, the authors suggested a power reduction method by using multiple instruction reads per clock cycle (VLIW architecture) combined with an instruction queuing technique. The ESP estimator results were compared with results obtained by a gate-level power estimator called VeriPower that used a gate or transistor-level netlist to report power per component and total circuit. It was noted in the paper that the simulation approach taken by the ESP gave poor results when estimating data-path power dissipations.
In [33], the authors parameterized averaged power models of common structures present in superscalar microprocessors that could be further integrated in architectural simulators to provide power estimates. Detailed power modeling methodologies were presented for common categories of processor units. The processor units considered were: array structures, fully associative content-addressable memories, combinational logic and wires, and clocking. The power models and simulators used to perform architectural-level power analysis were verified against industrial circuits and showed a maximum error of 10% for the low-level capacitance estimates. The relative accuracy of the models was shown to be within 10-13% on average. The major limitations found were that the models did not take in account all of the miscellaneous logic present in real microprocessors and the models were technology dependent.

In [34], the authors argued that in today’s benchmarking ratings there was need for standardized methods in reporting maximum and average power ratings for processors in order to fairly compare power-performance efficiency. The authors presented new trends in low-power architectural designs and challenges that face the power estimation research area. Cycle-by-cycle power simulation techniques must be capable of accounting for clock gating technology. The granularity of clock gating lowers to individual pipeline stages. In a processor, the turned-off units and the additional clock gating circuitry burn small amounts of power that must be considered for a proper power saving evaluation. The authors introduced a power-aware micro-architecture that was intended to be sensitive in the modeling of power to the impact of power consumption due to specialized low power saving techniques incorporated at different levels of architectural design. For example, asynchronous techniques, like interlocked pipeline CMOS (IPCMOS) with locally generated clocks, provided the benefits of “fine-grain clock gating”. In order to account for the energy saving impact generated by such techniques, the power estimator must use a micro-architectural model that incorporated power estimation modules at various block structures impacted by these power saving techniques. The authors presented a high level organization of a model processor that was general and capable of incorporating many of today’s architectural power saving techniques.

In [35], the authors addressed the need for core-based power evaluation and optimization tools. A core was defined as an Intellectual Property or IP and was a pre-designed processing level
component such as a microprocessor, memory, DMA controller, UART, bus interface, or CODEC, residing on a system-on-a-chip (SOC). The modeling of cores in this paper was based on a concept of “instruction” as an action that could describe possible behaviors of the core. The instructions must have the property so that they cover the entire functionality of the core. It was expected that the core providers performed gate-level power analysis that could be used in the construction of power lookup tables for these so-called “instructions”. The simulation approach was applied to a SOC that contained three specialized cores. The model presented was validated against gate-level power evaluations done by the Synopsis Power Compiler. The maximum power estimation error was found to be 5% when compared to gate-level power estimates.

In [39], the authors argued that optimization for minimum average power might not proportionally impact the battery lifetime and vice versa. The policies discussed proposed to target the battery life-time maximization by using a discrete-time battery model. The charge storage in the battery was modeled as a capacitor and the following discharging effects were taken in consideration: battery voltage depended non-linearly on its state-of-charge; the actual usable capacity of a battery cell depended on the discharge rate (at higher rates, the cell was less efficient at converting its chemically stored energy into available electrical energy); the frequency of the discharge current affected the amount of charge the battery could deliver; and batteries operated at a high discharge rate for a short period of time could recover available charge if the current load was temporarily reduced. The paper presented two dynamic power management (DPM) policies, namely open-loop and closed-loop. Open-loop techniques were simpler and less expensive but the closed-loop techniques were more effective in prolonging the lifetime of the battery. The DPM policies were applied to a MP3 audio player SOC. The power management system was based on 4 system states: reduced-performance, low-power active, inactive, and quiescent. The system must be able to trade performance for power consumption. The main idea of the closed loop policy was to play low-quality music when battery voltage was falling under a threshold level. The open and closed loop policies were also considered for the case when the SOC had 2 battery supplies.

In [40], the authors presented power reduction policies through the collaboration of software. When devices were idle, they could be put to sleep in order to save energy. When requests came, the devices awoke to enter working states. Power management policies were based on the
interaction between the power manager, the requesters, and the devices. In general power management was implemented in two ways: no-direct interaction of the requesters to the power manager and requester-controlled power manager. The paper presented methodologies for power management that fell between these two extremes. The experimental environment for this work was a notebook computer, an Ethernet PC-card, and a special PC-card extender that provided probing points for measuring power. The measurements of power were performed by the use of a data acquisition card. The power management schemes introduced in this paper (power management by process scheduling and requester device requirements) proved to be up to 70% more power efficient than the no power management scheme or the power management scheme that used process information.

2.2 Idle Low-Power Saving Modes in Specialized Processors

The idle low-power saving modes have a dynamic and a static characteristic. The dynamic power management techniques seek to save power by disabling internal functional units based on a dynamic idle computational situation. On the other hand, the static power management techniques are in effect at a global level when the processor is idle for any extended period of time. Most of the static power management modes are software programmable. As a result, an idle time software loop that is entered in any idle situation can be incorporated into an application. The loop contains the code necessary to set the enabling bits for one of the static power modes that is suitable to the input conditions. Wake-up signal events like external interrupts, internal counters, timer interrupts, or system management interrupts can be used in low-power implementations. These idle modes have different names, but have common characteristics across different implementation platforms. Due to the intermittent activity within a system, the static idle/low-power saving modes have become very effective modes in the energy conservation protocols and software applications.

For example, the PowerPC microprocessor [71], targeting portable computers, incorporates a variety of features to reduce power dissipation. It also includes three static power management modes called: doze mode, nap mode and sleep mode [71].
Another example of a specialized microprocessor that targets communication systems and has special low-power dissipation features is the Intel StrongARM SA-1110 microprocessor [41]. This processor contains a power management unit that controls the transition between three basic modes of operations: run, idle, and sleep. These modes are used to reduce power dissipation when some functions are not needed, or when the system's power supply is low, or out of regulation [41]. Idle mode is entered via software and sleep mode is entered via software or by asserting one of two input pins that indicate a power supply fault [41]. Idle mode is exited through an interrupt and sleep mode is exited through a preprogrammed wake-up condition. If none of the power management modes is active, then the processor is in the normal run mode. The following is a short description of the processor’s modes [41]:

- **Run mode**: This mode is the normal operating mode of the SA-1110 when all power supplies are enabled, all clocks are running, and all on-chip units are functional.

- **Idle mode**: This mode allows a software application to stop the CPU when not in use while continuing to monitor interrupt service requests. When an interrupt occurs, the CPU is reactivated. In the idle mode the CPU clock is stopped and all CPU state information is saved. This allows a switching back to run mode, starting the operation exactly where it left. During idle mode, all other on-chip resources are active (real-time clock, operating system timer, interrupt controller, general-purpose I/O, and power manager).

- **Sleep mode**: Sleep mode offers the greatest power saving and the lowest level of available functionality. In the transition from run or idle to sleep mode, the SA-1110 performs an orderly shutdown of on-chip activity. Internally, the power to the majority units of the processor is switched off from 1.5 V to 0 V. An internal sleep state machine watches for a preprogrammed wake-up event to occur in order to initiate a wake-up sequence.

Finally, an example of a high performance low-power processor specially designed for mobile applications is the Mobile Intel Pentium III Processor-M [42], [43]. This processor employs low-power fabrication technologies with dynamic switching of the voltage and frequency between peak and battery optimized modes based on user needs [43]. It also has a special idle mode called Deeper Sleep that is a Deep Sleep State at a lower voltage. Besides this new idle mode, the processor features other previous idle modes implemented by the Intel processors: AutoHALT,
Stop-Grant, Sleep, and Deep Sleep. All of these modes are controlled by special signals, which condition the power management state machine to enter in one of these saving modes. Each saving mode has a power dissipation characteristic and most of them can be controlled by software.

In general, the main purpose of all of the power saving modes presented is to cut off the distribution of the clock to their core units and thus reduce the power consumption by reducing the switching activity. However, there are large current transients as the core enters or exits these modes [23]. If these modes are to be used in low-power software applications, then the applications must be designed such that the processor has long sleep periods. By having long sleep periods, one is assured that the impact of the transient energy consumption has a small impact on the overall savings. Power state changes have overheads and delays, and a device should sleep only when the overhead can be justified [40]. In order to properly estimate the battery lifetime and the effect of the power saving modes there is need for dynamic power measurements and modeling at the application level.

2.2.1 Energy Conserving Strategies at the Software Level

There are low-power techniques that use power down schemes in order to save battery lifetime [12]. Also, there are low-power designs that lower the frequency of a circuit and increase the parallelism in order to maintain the throughput [6]. Assuming a well designed processor with low power features the final instruction set architecture will have a specific power consumption characteristic that reflects all the low-power electronic features. Therefore, the software performance from the energy conservation point of view is a reflection of the underlying hardware. The power-conscious software applications must take into consideration the power saving features provided by the hardware, and also must employ efficient implementation techniques at the software level such that the execution time and the current consumption of an application are minimized. A general architectural characteristic of the specialized microprocessors for low-power is the idle low power saving modes.

Power-conscious software applications for low-power systems should take into consideration the idle saving modes. At the software level, the strategies based on the idle modes are called (in this
thesis) idle software strategies. Within these strategies, the energy saving process is accomplished through the fact that the processor operates at low levels of its available functionality. The energy or power savings generated by the software implementation while using the idle mode states are called idle savings. On the other hand, the software implementations can utilize different software saving strategies that are not related with the idle saving modes. Within these strategies the energy saving process is not a direct reflection of the underlying hardware. In this thesis, these strategies are called non-idle software strategies. The energy or power savings generated by the non-idle software strategies are called non-idle savings. The more practical energy calculation Equations (2.2) and (2.5) can be used in assessing the energy savings generated by a software energy conserving strategy. A larger class of energy conserving techniques is presented in [11]. Here two techniques that can be categorized as non-idle strategies are outlined:

- **Compiler techniques.** Compiler design techniques have the main goal of minimizing energy consumption per application by optimization for speed in order to reduce $T$ or by optimizations such as reducing the number of memory accesses or code size that reduce $i(t)$. It is known that instructions with memory access operands have higher energy costs than instructions with register operands [6], [45]. As a result, it was indicated in [11] that the energy consumption per program could be saved by suitably assigning the live variables of the program to registers with optimized spills to memory. In [46], it was indicated that by assigning the loop variables to registers, a reduction in energy costs per program could be obtained. The target processor of this thesis was a specialized wireless communication processor with dual bank registers [47]. When registers from the upper bank were used in an execution set, there was a code size penalty through extra prefix-words added to the machine code [47] and there was also an extra energy cost penalty for the execution sets that used the upper bank registers [19]. For such processors, it was indicated in [48] that an optimal register allocation was important in order to reduce code size and energy costs.

- **Power efficient data structures.** An important domain where data set structures are used is data base management [49]. In communication applications, the dynamic data sets of records are very common [11] and are used across the layers of the communication protocol [49]. A model
for estimating power dissipation of manipulating data structures based on table organizations was proposed in [49].

2.3 Overview of Timing and Power Analysis Attacks in Cryptography

This section describes some basic principles of power analysis attacks used in cryptographic systems. In general, a cryptographic system uses a protocol to encrypt an input message as a function of a secret key and generates an output message that can be securely transmitted to the outside world. Unfortunately, there are many ways of collecting information about the computational activity that takes place inside an electronic system and thus making vulnerable the security of hiding the secret key in such systems. Known ways of leaked side channel information include: execution time, power consumption, radio frequencies, magnetic field values, and faulty hardware [16], [17], [18].

Each elementary operation performed by a circuit is considered to activate specific functional blocks together with a specific network of control signals and data path. The overall number of transistors activated is considered to be operation-dependent and thus has a main role in determining the amount of current, power, and energy consumption per operation. The constant flow of current through an integrated circuit is considered to be the main information leak of the cryptosystem. The information related to the operations performed by the cryptosystem can be obtained by current, power, or energy consumption measurements. These measurements can use direct techniques, when the equipment or an auxiliary element used for conveying measuring information is inserted in the power circuit of the cryptographic system [17], or electromagnetic inductive techniques.

Figure 2.1 presents a general direct measurement setup that could provide proportional voltage variation with the current consumption of a cryptosystem [17]. Similar measurement techniques were also used in [26] and [37]. High frequency sampling equipments can be used to sample and store the voltage variations while the system is executing the cryptographic application. The samples can be stored and transferred to a PC where further analysis can be performed [17]. Any setup capable of providing power consumption information on a cryptographic system can be used to develop a power analysis attack.
The general idea of a power analysis attack is that the attacker does not have access to the secret information (secret key) of the user, but he can access any non-secret information (algorithm implementation, public keys, portions of cipher text, hardware particularities of the system and so on) and he can access a power consumption measurable point on the hardware.

The power analysis attack, first introduced by Kocher in [15], has become an area of increased research and many hardware and software countermeasures have been proposed [17], [18], [50], [65]. Kocher et al. outlined in [17] that as long as the cryptographic protocols are implemented by software running on electronic circuitry, the danger of a power analysis attack exists and thus the security communication protocols should be designed based on the power attack assumption. Also, it was outlined that the hardware or software countermeasures can only increase the difficulty of the cryptanalysis, but it should not be considered that any of these countermeasures can eliminate completely the possibility of a power attack [17]. In [15], Kocher had indicated that due to the complex nature of cryptographic applications there may be a detectable time dependency based on different data inputs. Performance optimizations bypass unnecessary
operations or employ special instructions that can be localized in a current or power trace [15]. Knowledge (encryption key and input data dependency) about algorithm structure and some of the current consumption characteristics of the system, made it possible to identify and localize key dependent information within a power trace. Carefully measuring time executions required to perform private key operations had led to finding fixed Diffie-Hellman exponents, factor RSA keys, and even penetrated more complex cryptosystems [15]. In [15], Kocher presented a cryptanalysis example for a Diffie-Hellman and RSA private-key operations. For example, the cryptanalysis considered the following algorithm for computing $R = y^x \mod n$, where $x$ was $w$ bits long:

\begin{verbatim}
Let $s_0 = 1$
For $k = 0$ upto $w-1$
    If (bit $k$ of $x$) is 1 then 
        {Let $R_k = (s_k \cdot y) \mod n$}
    Else 
        {Let $R_k = s_k$}
Let $s_{k+1} = R_k^2 \mod n$
EndFor
Return ($R_{w-1}$)
\end{verbatim}

The attacker's goal was to find $x$, the secret key. The attack was developed such that the knowledge of a string of $b$ successive bits of the exponent $x$ could determine the bit $(b+1)$ of $x$. It was assumed that the system under attack computed $R = y^x \mod n$ for several values of $y$, while the attacker knew $y$, $n$, and the computation time. Other assumptions considered in this cryptanalysis attack were that the exponent $x$ was fixed and the attacker knew the design details of the system. This attack introduced by Kocher in [15] was called a timing attack and its principles would work virtually with any cryptographic implementation that does not run in fixed time as related to its data input. The general timing attack was treated as a signal detection problem where the signal consisted of the timing variations due to the target exponent bit, the noise resulted from measurement inaccuracies, and the unknown exponent bits [15]. The signal detection attack had to be conducted on a large number of timing measurements. The timing attack had an error-detection property such that after an incorrect exponent bit guess there was no further useful information. Due to the error-detection property of the attack, the number of the samples required
for a timing attack was proportional with \( w \), the size of the exponent [15]. The experimental results presented in [15] were for a 120 MHz Pentium processor.

In [15], Kocher also presented three basic countermeasures to the timing attack:

- Equalize the execution times for all operations of the protocol. This method is not portable from one platform to another due to compiler optimization features and hardware particularities of the system related to cache organization.

- Randomize the timing measures by introducing random delays within the processing time of a cipher text. This method increases the number of samples an attacker must collect.

- Use of the blinding signature method that prevents the attacker from knowing the data input.

The main conclusion presented by Kocher in [15] was that vulnerable algorithms, protocols, and systems must be revised in order to incorporate measures to resist timing cryptanalysis.

**Power analysis attacks**, like the timing analysis attack, base their success mainly on collecting information about the cryptosystem from leaked information through power. These attacks base their success on the fact that the amount of power consumed within an electronic system varies depending on the instructions and data handled by the processor [16], [6].

In the **Simple Power Analysis** attack (SPA), the attacker directly observes the system’s power consumption. Large features of cryptographic algorithms can be identified visually or through simple measurement techniques on a power trace captured while the cryptosystem performs a secret operation. Specific block structures within the cryptographic implementations can be identified by higher or lower power consumptions. For example, in some cryptosystems, operations like multiplication, squaring, permutations, shifts, and so on, can be easily identified through power consumption analysis [16], [17]. It has been proven that the SPA could be effectively used in smart cards and other low frequency cryptosystems for finding information about the algorithmic sequencing and secret key material and thus, the secret information of the cryptosystem was made vulnerable to an attacker [15], [16], [17]. In [17] Kocher et al. showed that from a single power trace the SPA could reveal the sequence of instructions in a smart card that
performed a DES operation. As a result, the SPA could be used to attack cryptographic implementations that had the execution path of the algorithm dependent on the processed data. It was indicated in [17] that the SPA attack can be prevented by avoiding procedures that use secret sensitive information for conditional branching operations, by creating constant execution path algorithms, and by implementations of symmetric cryptographic algorithms. However, in microprocessors with large operand-dependent power consumption features, the SPA can pose a serious threat even for the constant execution path code [17].

**Differential Power Analysis (DPA)** attack [16] is another attack that is more powerful than the SPA and it uses statistical analysis techniques over a large number of operations. In [16], Kocher et al. developed a DPA attack that targeted a DES implementation. In [54], Coron generalized the DPA attack introduced by Kocher et al. [16] to elliptic curve (EC) cryptosystems and described a DPA attack on EC Diffie-Hellman key exchange and EC El-Gamal type encryption. Coron suggested that a naïve implementation of an EC scalar multiplication might be vulnerable to the SPA attack. The author presented a possible scenario of a DPA attack that targeted a double-and-add scalar multiplication algorithm that was resistant against SPA attacks. For a fixed scalar $k$, the algorithm was executed $m$ times with $m$ distinct points. The power consumption associated with the $i$th execution was marked as $C_i(t)$. For a specific bit $s_i$ of the binary representations of the points considered a correlation function $g(t)$ was calculated as:

$$g(t) = \langle C_i(t) \rangle_{i=1,2,...,m|s_i=1} - \langle C_i(t) \rangle_{i=1,2,...,m|s_i=0}$$

The idea of the attack was that the average power consumption for the points with $s_i = 1$ was different than the average power consumption for the points with $s_i = 0$. Considering the possibility of power analysis attacks against EC card implementations, Coron suggested three countermeasures: randomization of the private exponent, blinding the point $P$, and randomized projective coordinates.

In [18], Hasan proposed a series of three countermeasures applicable to EC cryptosystems that had an ultimate effect of increasing the number of sample traces needed by the attacker in order to successfully attack the system. The methods proposed were key masking with localized operations, random rotation of key, and random insertion of redundant symbols. It was also shown that
applying all these methods to an elliptic curve cryptosystem in a combined form made the
differential signal needed for a DPA attack to be attenuated to a level that made the DPA attack
infeasible with technologies available today.

**High-Order Differential Power Analysis** [16] (HO-DPA) is another power analysis technique
that uses information collected across multiple events within the cryptographic application. This
technique uses signals collected from multiple sources, signals collected using different measuring
techniques, and signals with different temporal offset.

Countermeasures to the power analysis attacks include software and hardware techniques.
Developing any software or hardware countermeasures of power analysis attacks for cryptographic
applications must take into consideration all of the existing power analysis attack techniques. Some
of the hardware techniques like random process interrupts and noisy power consumption had
been already proven vulnerable on smart cards implementations [50].

### 2.3.1 Introduction to Elliptic Curve Cryptosystems

The elliptic curve-based cryptosystems are implemented on a large class of systems and platforms.
Elliptic curve cryptography is considered more efficient than RSA technology because it offers
same level of security with less number of bits per secret key [54], [60]. Important implementation
advantages related to processing time and secret key size made the elliptic curve (EC)
cryptosystems very practical [18]. It is important to consider and analyze the resistance of elliptic
curve based cryptosystems to timing, SPA, DPA, and HO-DPA attacks. The remainder of this
section introduces some basic concepts and implementation facts for elliptic curve cryptography.

Consider a Galois field $GF(2^n)$ and a standard Weierstrass elliptic curve $E$ over this field:

$$y^2 + xy = x^3 + ax^2 + b$$

(2.10)

In Equation (2.10), the parameters $a, b \in GF(2^n)$ with $b \neq 0$. The idea of an EC based
cryptosystem is to create an algebra that can use the elliptic curve defined over the finite field
$GF(2^n)$. If one defines an identity element $O_\infty$ such that $P + O_\infty = P$ and $O_\infty + P = P$
then the points on the elliptic curve form a cyclic group [51] ($P$ is a point on the elliptic curve). Given two
points on an elliptic curve, \( P = (x_1, y_1) \) and \( Q = (x_2, y_2) \), the main basic operation defined for the cyclic group is the addition of points over the elliptic curve, such that \( R = P + Q = (x_3, y_3) \).

In calculating \( R \) there are 2 distinct situations [52]:

a. If \( P \neq Q \), one can calculate \( c = \frac{y_2 - y_1}{x_2 - x_1} \) and the coordinates for \( R \) as follows:

\[
\begin{align*}
x_3 &= c^2 + c + x_1 + x_2 + a \\
y_3 &= c(x_1 + x_3) - y_1
\end{align*}
\]  

(2.11)

b. If \( P = Q \), one can calculate \( c = x_1 + \frac{y_1}{x_1} \) and the coordinates for \( R \) as follows:

\[
\begin{align*}
x_3 &= c^2 + c + a \\
y_3 &= x_1^2 + (c + 1)x_3
\end{align*}
\]  

(2.12)

The inversion operation over the finite field \( GF(2^n) \), necessary to calculate \( c \), is considered the most time consuming operation of the point addition in an elliptic curve cryptosystem [52], [53]. Other basic operations defined over the elliptic curve field are:

- Negation of a point \( P(x, y) \): \( -P = (x, y + x) \).

- Multiplication of a point \( P(x, y) \) with an integer \( k \), called point multiplication or scalar multiplication: \( kP \). This multiplication is performed by adding the point \( P \) with itself \( k \) times.

The elliptic scalar multiplication \( kP \) defined above is a fundamental operation in the elliptic curve cryptosystem protocols and is the main target of the power analysis attacks [18]. A general scalar multiplication algorithm (Algorithm 13 of [60]) is given in Table 2.1, with implementation details on doubling, adding, and subtracting points [53]. In Table 2.1, before the scalar multiplication is performed the binary representation of \( k \) is converted to its non-adjacent form \( NAF(k) \) by using an algorithm introduced by Solinas [61].

The scalar multiplication algorithm is central to the Diffie-Hellman and ElGamal key agreement EC protocols. For example, in the Diffie-Hellman EC key agreement scheme [52], [55] Alice and Bob, two communicating parties, first agree to use a specific curve, field size, a base system for
computations, and a public base point \( B \) on the chosen curve. Alice and Bob choose a random bit pattern \( k_A \) and \( k_B \), respectively to be their private key. Bob computes \( P_B = k_B B \) over the elliptic curve, which becomes his public key, and sends the computation result to Alice. Alice computes \( P_A = k_A B \), which becomes her public key, and sends the result to Bob. Then they both compute the shared secret based on the following relation: \( P_S = k_A(k_B B) = k_B(k_A B) \). The \( x \) component of the shared secret becomes the secret key that can be used for encryption of secret information. New public keys can be generated every time when a communication session begins. Each side has to transmit its public key to the other side before the common secret key can be computed.

In the ElGamal EC key agreement scheme [52], [55], Alice and Bob (two communicating parties) first agree to use a specific curve, field size, a base system for computations, and a public base point \( B \) on the chosen curve. Similar to the Diffie-Hellman protocol, Alice and Bob calculate their public keys based on the private keys and the base point. The parties must exchange the public keys \((P_A, P_B)\) before a secret information exchange can take place. For example, if Alice wants to send secret information to Bob, first she embeds the information on the elliptic curve and gets a message point \( P_m \). Then she chooses a random bit pattern \( v \) and computes the following points: \( P_v = vB \) and \( P_u = P_m + vP_B \). Alice then sends both points, \( P_v \) and \( P_u \), to Bob. To extract the message point, Bob computes:

\[
\begin{align*}
P_S &= k_B P_v \\
P_m &= P_u - P_S
\end{align*}
\]

The only points an attacker sees in this scheme are: \( P_A, P_B, P_v, P_u \). In this protocol, the public key can stay public since at every new communication there is a new random element \( v \).

The algorithm presented in Table 2.1 was used by an elliptic curve scalar multiplication implementation called polymulNIST.asm. The target processor of this implementation was the SC140 DSP processor [65]. The polymulNIST.asm implementation and its subroutine polymul.asm were the target implementations of the simple power analysis techniques and the results presented in Chapter 6. Also, the energy analysis technique for secure implementations presented in Chapter 6 made use of the EC information presented in this section. The next chapter presents the equipment and the experimental setup used in this thesis.
Table 2.1. General algorithmic description of EC scalar multiplication, point addition, point subtraction, and point negation.

<table>
<thead>
<tr>
<th>Algorithm 1. Binary NAF method for point multiplication.</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>INPUT:</strong> $NAF(k) = \sum_{i=0}^{l-1} k_i 2^i, ; P \in E(GF(2^n))$</td>
</tr>
<tr>
<td><strong>OUTPUT:</strong> $kP$</td>
</tr>
<tr>
<td>1. $Q \leftarrow O_\infty$.</td>
</tr>
<tr>
<td>2. For $i = (l-1)$ down to 0 do</td>
</tr>
<tr>
<td>2.1 $Q \leftarrow 2Q$</td>
</tr>
<tr>
<td>2.2 If $k_i = 1$ then $Q \leftarrow Q + P$</td>
</tr>
<tr>
<td>2.3 If $k_i = -1$ then $Q \leftarrow Q - P$</td>
</tr>
<tr>
<td>3. Return $(Q)$</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Algorithm 2. Adding 2 points</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>INPUT:</strong> $P = (x_1, y_1), Q = (x_2, y_2)$</td>
</tr>
<tr>
<td><strong>OUTPUT:</strong> $R = (x_3, y_3) = P + Q$</td>
</tr>
<tr>
<td>1. If either point is $O_\infty$, return the other point as the sum.</td>
</tr>
<tr>
<td>2. If: $x_1 = x_2 : when y_1 \neq y_2; return O_\infty$</td>
</tr>
<tr>
<td>otherwise use doubling rule.</td>
</tr>
<tr>
<td>3. If:</td>
</tr>
<tr>
<td>$x_1 \neq x_2 : then ; (x_1, y_1) + (x_2, y_2) = (x_3, y_3); ; where$</td>
</tr>
<tr>
<td>$x_3 = c^2 + c + x_1 + x_2 + a, ; y_3 = c(x_1 + x_2) + y_1, ; and ; c = \frac{y_1 + y_2}{x_1 + x_2}$</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Algorithm 3. Doubling a point.</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>INPUT:</strong> $P = (x, y)$</td>
</tr>
<tr>
<td><strong>OUTPUT:</strong> $2P$</td>
</tr>
<tr>
<td>1. If:</td>
</tr>
<tr>
<td>$x = 0 : then ; 2(x, y) = O_\infty; otherwise ; 2(x, y) = (x_3, y_3); ; where$</td>
</tr>
<tr>
<td>$x_3 = c^2 + c + a, ; y_3 = x^2 + (c + 1)x_3, ; and ; c = (x + \frac{y}{x})$</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Algorithm 4. Subtract 2 points.</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>INPUT:</strong> $P = (x_1, y_1), Q = (x_2, y_2)$</td>
</tr>
<tr>
<td><strong>OUTPUT:</strong> $R = (x_3, y_3) = P - Q$</td>
</tr>
<tr>
<td>1. Negate point: $Q = (x_2, y_2) as -l(x_2, y_2) = (x_2, x_2 + y_2)$</td>
</tr>
<tr>
<td>2. Use Algorithm 2 for adding the points: $R = P + (-Q)$</td>
</tr>
</tbody>
</table>
Chapter 3

THE PROCESSOR AND THE EXPERIMENTAL SETUP

The main focus of this chapter is the SC140 DSP processor core and the experimental setup necessary to conduct instantaneous current measurements. An interrupt-based synchronization technique for current measurements is introduced in Section 3.2. This technique enabled the collection of instantaneous current traces at the instruction and program level. These traces could be further used for complex current measurements and analysis. Section 3.3 presents a static measuring technique that was used in comparison analysis between the average power models developed in previous research, and the current dynamics model developed in this thesis.

3.1 The Processor

This section presents an introduction to the DSP VLIW processor core, SC140, as a specialized wireless communications processor that was the target of the current measurements and methodologies presented in this thesis.

• StarCore 140 Software Development Platform (SC140 SDP). The SC140 SDP is a hardware/software tool for development applications that use the SC140 DSP processor core [56]. For the real-time applications, the software is downloaded to an on-chip SRAM memory (256×16-bit SRAM) and it can be executed or debugged by using a JTAG type port. The SC140 SDP hardware was connected to a host computer as a stand-alone board using an external power supply [56]. In this connection, the communication between the host computer and the development board was done through a parallel port. The board configuration and the setup characteristics were determined through a set of jumpers. Figure 3.1 presents a general connection diagram for the SC140 SDP using the parallel port communication protocol. The measurement point for the processor core is also shown. The current and the voltage measurements were performed at the processor's power supply pin. The StarCore SC140 processor chip operates at a maximum internal frequency of 300MHz. However, the maximum external JTAG bus’s speed of operation is 40MHz. The board can provide an on-board 40MHz clock signal to the processor.
For higher clock speeds, the signal must be provided from external sources via the CLKin BNC connector [56].

- **The SC140 DSP processor.** The SC140 StarCore processor supports a wide range of wireless applications including special filtering, amplifying, and encrypting [47]. The DSP processor has special architectural features with multiply/accumulate instructions, multiple ALU blocks, special addressing modes, specialized register structures, and a very specialized instruction set. The SC140 processor is a VLIW architecture with a Variable Length Execution Set (VLES) model [47]. The instruction words are 16-bit wide and each SC140 instruction encodes an atomic (lowest-level) operation [47]. In order to execute signal-processing operations, a set of instructions can be executed in parallel. The internal block diagram of the processor as provided by the manufacturer [47] is presented in Figure 3.2. An eight-word instruction set, called a fetch set, is fetched from memory every clock cycle. Next, the PSEQ (Program SEQuencer) unit automatically detects the portion of the fetch set called an ‘execution set’ (ES) that can be executed in parallel. Each instruction that is included in the execution set is dispatched to its appropriate execution set for execution. Here the VLES model is enforced, and based on grouping constraints, the execution set that can be executed in parallel varies from one to six atomic instructions [47].
Instructions belonging to an execution set start execution concurrently after all of the instructions belonging to the previous execution sets are completed. Therefore, the execution time of an execution set is determined by the instruction in the set with the longest execution time. The SC140 pipeline consists of five stages: pre-fetch, fetch, dispatch, address generation, and execution stage [47]. A simple pipeline development for this processor with no parallel execution and no pipeline hazards is presented in Figure 3.3, where at every clock cycle a new instruction is executed. Most of the SC140 instructions take one clock cycle. Some of the change-of-flow instructions and some instructions that operate on memory require additional clock cycles. The SC140 can generate three simultaneous memory accesses per clock cycle, one program fetch and two 64-bits data accesses [47]. Nevertheless, this capability is controlled by the specific on-chip memory units. By means of a bus controller, the memory system identifies access combinations that could not be supported simultaneously and stalls the SC140 processor [47]. For example, a stall occurs when a memory unit that supports one access at a time receives a simultaneous request for two data accesses, or for one program access and one data access [47].
A special characteristic of the SC140 processor is that the register file blocks are divided into two banks: bank 1 or the lower bank that is the non-penalty bank; and bank 2 or upper bank that is the penalty bank (more code size and energy costs) [48]. The usage of a register from the upper bank by any of the instructions in the execution set generates a two-word prefix in the executable code equivalent to 4 bytes of code [47]. The current function of a software application running on the SC140 processor showed a complex dependency on data and instruction line sequencing [21], [73]. In general, the main current function for each instruction showed to be related to the functional units activated for executing the particular instruction and to the specific control logic (type) of the instruction. As a result, there was a difference in the energy costs per individual instructions of the instruction set architecture, and the energy costs per application were affected by the specific assembly language constructs used [20], [21].

3.2 Current Dynamics Measurement Technique in a DSP Processor Core

This section presents the measurement setup and the equipments used for performing the current dynamics measurements for the SC140 processor. The equipments to support these measurements included: HP 81130A Pulse/Pattern Generator, Tektronix TDS 8000 Digital Sampling
Oscilloscope, TCP202 DC/AC Inductive Current Probe, and CT1 AC Inductive Current Probe. The description of these equipments is presented in Appendix A and the equipment connection setup for the current dynamics measurement is presented in Figure 3.4.

The pulse/pattern generator was used to provide the trigger signal for the oscilloscope, the processor's clock for the SC140 SDP board, and the software synchronization signal [20], [21]. The inductive current probes were used to collect the current signals from the measuring points. The oscilloscope acquired the current waveforms from the signals generated by the current probes by the use of the sampling modules [58]. The sampling modules were connected to the acquisition channels (C, short for channel) of the oscilloscope. The oscilloscope performed all of the current waveform collections at the instruction and the program level. Single, averaged, and mean filtered current traces were collected by the oscilloscope (See Section 5.1). These current traces were analyzed by the use of the built in features of the oscilloscope and were also exported to the host computer where further MATLAB analyses were performed. The DC/AC current probe was used
to capture the current signals at the program level. This probe has the capability of capturing current signals that have a DC component and an AC component within a frequency bandwidth of [0MHz to 50MHz]. The bandwidth of the current signal at the program level proved to be much smaller than the clock frequency bandwidth and as a result, the measurements provided by the use of the DC/AC current probe were not distorted. The AC current probe has a high frequency bandwidth and increased sensitivity (See Appendix A). It was used to capture the small signal variation of the current above its DC level. All of the measurements at the instruction level were performed by the use of this current probe. Due to its increased stability and sensitivity, this probe was also used to provide all of the current signals related to the security analysis.

Both of the current probes are inductive probes and their accuracy is within ±3% range. The inductive signal collection was chosen instead of the direct measurement technique used by previous research ([6], [7], [14], [24], [26], and [37]) for two main reasons:

- It was found that for the SC140 DSP core the direct current measurements altered the functionality of the programs analyzed and the measurements performed in this way were compromised.

- The direct measurements introduced a voltage variation, in a 1Ω resistor, at the processor’s power supply within [70mV to 300mV] range and that was within [3.68% to 15.81%] ratio of the processor’s measured voltage supply of 1.898V.

The current consumption at the program level was dynamic in nature with asynchronous behavior. As a result, in order to perform true measurements there was need for synchronization between the execution of the software application, the processor’s clock, and the oscilloscope [20]. Within the measuring setup of Figure 3.4, the synchronization problem was solved as follows:

1. The processor’s clock was externally provided at a working frequency by the pattern generator.

2. The processor was kept in the wait state until a measurement signal was issued.

3. The measurement signal was an external interrupt signal applied to the NMI pin of the processor [47] that was also the oscilloscope’s trigger signal.
4. The program under measurement (PUM) was encapsulated with a series of non-operational type instructions (NOP) before and after entering the wait state.

5. The period of the measurement signal delimited a sample frame and determined the maximum execution length of the PUM. When the interrupt signal was asserted, the processor was already in the wait state. The interrupt signal caused the processor to switch from the wait state and execute an interrupt service routine (ISR) and then the PUM. As a result, the program execution was synchronous with the interrupt signal and the trigger signal to the oscilloscope.

6. At the end of the program execution the processor was introduced again in the wait state and continuous samples were collected as long as the measurement signal was activated.

The signal diagram of the described software synchronization mechanism is presented in Figure 3.5. The program under measurement was part of a software interrupt routine that was called a "measuring template" (MT). The MT could be developed in C or assembly language using the SC140 SDP software tools. The basic block structure of the MT assembly language used is shown
in Table 3.1. The target software application was developed and debugged using the host system and then it was incorporated within the MT. In order to acquire a stable current waveform for any software application, the application had to start and execute under the same conditions in every sampling frame of the oscilloscope. A sample frame for the current waveform was triggered by the rising edge of the trigger signal and its length and position were determined by the oscilloscope's horizontal synchronization setups (See oscilloscope functionality, Appendix A). The wait periods \( (T_{\text{WAIT}}) \) and the NOP periods \( (T_{\text{L,NOP}} \text{ and } T_{\text{L,NOP}}) \) were visually identifiable on the current waveform. The execution time of the entire PUM or portions of PUM could be determined precisely by the use of software triggered signals that were collected at an output port called EXE. The triggering points were placed at the beginning and at the end of an execution period. Approximated values could be also determined by visually inspecting the current traces. Due to the oscilloscope’s sampling limitations, (See Appendix A) direct current measurements were possible on software applications that had an execution time of less than 50ms. Programs with longer execution times had to be divided in smaller blocks that executed for less than 50ms and the synchronization had to be done at the block level.

Figure 3.6 presents an example of a synchronized waveform for a short assembly language program. The current waveform was stored and displayed on the oscilloscope screen. Measurements and calculations involving various math functions were performed after the waveforms were stored into the acquisition memory. The following describes the usage of the oscilloscope’s measurement channels used in Figure 3.6 and the energy calculation based on the stored waveforms. Channel 6 (C6) recorded the instantaneous DC power supply voltage at the processor pin. The reference waveform R2 stored the instantaneous current of the processor measured by the AC/DC current probe that was connected to C5. The math waveform M1 calculated the instantaneous current of the processor measured by the AC current probe that was connected to C3. C4 recorded the synchronization signal generated by the pattern generator and C1 was connected to the EXE port signal.

The energy consumption per application was calculated based on Equation (2.5) of Section 2.1.1 as an integral of the product between the current and the voltage waveforms over the execution period of the application:
In Equation (3.1), the end points of the integration interval were identified by the rising and the falling edges of the EXE signal. These end points were identified in Figure 3.6 as $T_{START}$ and $T_{END}$, and were measured in seconds.

Due to the low level of current variation per instruction, the current consumption per individual instruction was measured by the use of the AC probe (see Section 5.1). In order to obtain accurate measurements by using this probe, the measurement errors caused by the DC component and by the measuring template had to be eliminated. In principle, one or more individual instructions were grouped in a short program called PUM$_i$ and this program was placed inside the measuring template to obtain a program called MT(PUM$_i$). Another program called MT(PUM$_{NOP}$) was generated as a measuring template that included a program under measurement composed of NOP instructions. The total number of NOP instructions used in a PUM$_{NOP}$ had to be equal with the total number of clock cycles executed by the corresponding PUM$_i$. Current measurements were performed and stored for both programs, MT(PUM$_i$) and MT(PUM$_{NOP}$). The final AC current consumption per instruction (current variation above NOP-level) was calculated as a math waveform [58] by the oscilloscope as:

$$M_i = 0.2 \cdot \left[ C_3(MT(PUM_i)) - C_3(MT(PUM_{NOP})) \right]$$  \hspace{1cm} (3.2)

In the math waveform calculations, the constant 0.2 accounted for an amplification factor of 1:5 introduced by the CT1 probe [58]. By subtracting the current consumption record caused by the MT(PUM$_{NOP}$) program, the DC current component of the MT(PUM$_i$) program, the template effects caused by the setup blocks, and the probe’s measuring error were eliminated. In order to minimize the measurement noise from these waveforms, the calculations of Equation 3.2 were performed on averaged measurements taken over collections of 400 current samples (See Section 5.1). To minimize the relative error of measurements in time and between instructions, the averaged calculated waveforms $M_i$ were applied to a built-in mean (centered) filter [58]. The filter
mode of the oscilloscope was set to “centered” and the filter rise time was set to at least 20ns. The filtered waveforms were exported to the digital sampling data base on the host computer where further function analysis through MATLAB programs could be performed.

| ; Software synchronization application |
| ; Program and instruction measuring template |
| org p:$0 |
| jmp START |
| org p:NMI ; nonmaskable interrupt zone |
| NMI: jmp ISR ; interrupt service routine |
| org p:DATA ; data test pattern zone |
| Initialize memory data with test patterns: |
| $0000 …, $FFFF …, $AAAA …, $5555 …, |
| $mixed …, $mixed …, $random …, $random …, |
| org p:START ; beginning of template program |
| START: Initialize 2 verification counters in memory zone |
| counter #1 and counter #2 |
| LOOP: Initialize stack pointer |
| Initialize stack pointer memory zone with random patterns |
| Initialize vector base table register |
| Initialize operation modes in status register |
| Initialize all internal data and address registers with: |
| random patterns or specific test values |
| ei ; enable interrupts |
| nop |
| wait ; put processor in wait state |
| ; to allow for software synchronization |
| nop ; return point after interrupt service routine |
| ; Begin verification block |
| Increment counter #2 |
| ; End verification block |
| jmp LOOP ; prepare the next execution of PUM |
| org p:ISR ; interrupt service routine |
| ISR: di ; disable interrupts |
| ; Begin Ti-nop block |
| 1000 nops block |
| ; End Ti-nop block |
| ; Begin optional verification block |
| Increment counter #1 |
| ; End optional verification block |
| jmp PUM ; program under measurement |
| APUM: ; point of return after PUM is executed |
| ; Begin Te-nop block |
| 1000 nops block |
| ; End Te-nop block |
| rte ; return from exception |
| org p:PUM ; program under measurement |
| PUM: ; Begin PUM block |
| PUM block – program under measurement |
| ; End PUM block |
| jmp APUM |
| end |

Tabel 3.1. Software synchronization template for the SC140 DSP processor.
3.3 Static Measurements of Average Currents for Software Applications

This section describes the static measurement methodology that was used in Section 5.4 to compare the research presented in this thesis with previous research developed in [6]. A simple low cost but less accurate approach to measuring and analyzing average current, power, and energy consumption in a processor is to use an ammeter based method [6].

In this thesis, the static current consumption measurements were performed by the hardware setup presented in Figure 3.7. Using Equation (2.2) of Section 2.1.1, the static measurement setup was used to obtain energy consumption approximations at the software level. The ammeter, a Fluke 867 Graphical Multimeter (See Appendix A), provided the average DC current consumption value at the processor pin while the processor executed the software application (the $I$ parameter of Equation 2.2).
In order to properly estimate the energy consumption, the application was executed in an infinite loop. The parameter $\tau$, which was the clock cycle period, was set by the pattern generator at: $\tau = 1 / f_{clk}$. For a given data input, the parameter $N$ (the total number of clock cycles per application), could be determined by the use of a simulator. However, in the SC140 experiments, differences between the number of clock cycles reported by the simulator and the real number of clock cycles executed by an application were observed. This was due to the fact that the simulator did not take into consideration the memory stalls introduced by the bus controller (See Section 3.1). As a result, the value of $N$ was determined from the oscilloscope’s waveforms by the use of the EXE signal (See Section 3.2).
Chapter 4

CURRENT DYNAMICS MODELING METHODOLOGY

This chapter introduces the abstract circuitry and the mathematical basis for the current dynamics modeling methodology at the software level. The current model, an instruction-level based model, is used to model the instantaneous current consumption of a processor while executing a software application. Four basic current superposition principles and a factorial dependency on parallelism formed the basic building blocks of the current dynamics model used for a parallel issue processor. The principles were derived experimentally using the setup presented in Section 3.2.

4.1 Problem Definition

A general assembly language application AL composed of instructions of any possible type, including program sequencing and control type, can be decomposed into basic blocks [6]. Figure 4.1 presents an example of such an application. The control type instructions formed marker points within the application and were used to delimit the basic blocks. In Figure 4.1, through segment lines \( b_j \), a common basic block delimited by branching points was used and through \( l_i \), a block formed by a loop was used. The loop blocks were themselves complex blocks and could be divided further into other common basic blocks or other loop blocks. In the short example presented, it was considered that \( l_0 \) and \( l_1 \) were simple basic blocks, which executed repeatedly a number of times determined at execution time by a loop counter (hardware loops [47]). Depending on data inputs at execution time, the program of Figure 4.1(a) could execute different sequences of blocks. For example, the data ‘input1’ could generate at execution time a trace through the program’s blocks like the one indicated in Figure 4.1(b) and the data ‘input2’ could generate a trace like the one indicated in Figure 4.1(c). It was assumed, as in previous research [6], that each individual instruction within the program made a contribution to the specific current consumption while executing. As a result, the current consumption of the application was dependent on the type of the individual instructions executed at run time (See Figure 4.2).
Furthermore, it was considered that the execution time period of the program was $N \cdot \tau$ (where $N$ was the total number of machine cycles and $\tau$ was the clock period). The value of $N$ may be dependent on the data input to the application $AL$. For a given data input, it was assumed that the application started the execution at time $t_0$ and was traceable in the sense that at any given point in time, it was possible to determine exactly the instruction that was being executed. The processor’s execution steps followed the sequence of instructions from the trace and one step was equal with the processor’s clock cycle period $\tau$. 

Figure 4.1. Illustration of an example of a general structure assembly language application for a DSP processor. The block decomposition with their delimiting points is shown in (a). In (b) and (c), two possible control path scenarios for different data inputs to the running application (a).
Figure 4.2. Example of possible instantaneous power waveforms for the application presented in Figure 4.1. The waveform $f_1(t)$ represents the power variation for the application generated by the instruction sequence of trace1 while $f_2(t)$ represents the power variation for the instruction sequence of trace2.

Considering general type applications like the one described above, the problem studied in this thesis is:

- **Developing an instruction-level macro-model that was capable of estimating the instantaneous current or power consumptions from a program trace for any given AL application running at any frequency within the working range of the processor.**
4.2 Proposed Superposition Principles for Current Modeling

The instruction level modeling approach considered instantaneous current and power analysis values at the functional block level that were composed of a large number of elementary CMOS circuits. The model reflected the global switching capacity of the processor as a function of the hardware logic architecture and of the data manipulated at the instruction level. The current simulation methodology used elementary functions called atomic functions to approximate the instantaneous current consumptions at the instruction level. This gave a macro-model characteristic to the current model. In this thesis, the instantaneous current model at the program level was also called a current dynamics model since the simulated waveforms captured the dynamics of the current. The current dynamics model approximated the real current dynamics of a program by a sum of atomic functions that represented the current consumption of the individual instructions of the program. The simulated waveform obtained in this way was called a base waveform. The final simulated waveform was obtained from the base waveform by applying a factorial adjustment that was dependent on the program parallelism and sequencing. The simulated waveforms of the programs were compared with real current measurements obtained through the setup in Section 3.2. The current dynamics model was developed through a systematic decomposition approach. The current activity that took place was simulated at the instruction level based on the functional activity of the processor.

The research assumed that the voltage supply $V_{dd}$ to the processor was constant and was equal to 1.898V (the measured DC voltage value of the power supply), and the frequency of the processor's clock signal was 100MHz, 200MHz or 300MHz. In the pipeline development logic of an individual instruction (See Figure 3.3 of Section 3.2) it was assumed that an increase of the switching activity was present as the instruction passed from the pre-fetch stage to the execution stage. The pipeline hardware logic is more complex when multiple parallel instructions are issued. The simulation of the instantaneous current consumption in this case becomes very complex and difficult due to the hardware logic complexity introduced by the parallel execution capability [14]. As a result, the instantaneous current simulation was simplified by the use of four current superposition principles that were assumed to reflect the behavior of the hardware logic of the processor. These superposition principles were empirically derived from current measurements performed at the
instruction and the program level. The current superposition principles used were called ‘total current superposition principle’, ‘pipeline superposition principle’, ‘random operand value and high bank superposition principle’, and ‘parallel issue superposition principle’. Based on these current superposition principles, a base current waveform could be generated at the program level by a generalized current superposition technique validated for the SC140 DSP processor core.

4.2.1 Total Current Superposition Principle

Measurements performed at a fixed frequency showed that the current consumption of the SC140 DSP processor executing strings of NOP instructions was at a minimum value and it was constant. This current value was called a NOP-level and it was denoted by $I_0$. Measurements performed on other individual instructions and short or long executable programs showed currents, with a variable component, that were above this minimum value. For example, Figure 4.3 presents current consumption waveforms and measurements for two assembly language applications (powerco.asm, powercoN.asm) at three different frequencies (100MHz, 200MHz, and 300MHz). The execution time for the powerco.asm was 167 clock cycles and for the powercoN.asm was 207 clock cycles. At the beginning and at the end of the program block, long series of NOP instructions were executed (See Section 3.2). The NOP-level for these current measurements is indicated in Figure 4.3 by arrows.

Measurements at the instruction-level for the SC140 processor showed that the impact of the switching activity developed during the execution of an instruction was visible for a period of time that was much greater than the clock cycle or the execution time of the instruction. This period of time was denoted with $\delta$ and it was considered to be equal with the period of time while $M_i$ (See Equation 3.2 of Section 3.2) was greater than zero. It was also found that at the program level there was a time needed for the current to reach the stable DC level (NOP-level) after the last instruction of the program was executed. This time was similar in value with the time variation present at the instruction level and it was also denoted with $\delta$. The time duration $\delta$ and the shape of the current variation per instruction showed consistency across the entire instruction set architecture (ISA), and was believed to be affected mainly by the capacitive characteristics of the
underlying technology of the processor and very little by the frequency of operation. The measurements that varied at different frequencies of operations were the NOP-levels and the maximum current variations. In order to support the assumptions of the current shape and current variation $\delta$ at the instruction level, Figure 4.4 presents examples of current measurements taken for four different assembly language instructions at three different frequencies. Note that the current waveforms presented in this figure record only the current variation above the NOP-level. The NOP-levels at the instruction level are the same with the NOP-levels at the program level. The instructions presented in Figure 4.4 completed execution after 5 pipeline stages (25ns at 200MHz) but showed current variations over much greater time intervals (See Table 4.2).
Table 4.1 summarizes the maximum (Max), mean, NOP-level, energy, and the time variation $\delta$ measurements of the current waveforms presented in Figure 4.3. The $\delta$ measurements represented the time duration needed by the current to reach a current level smaller than (NOP-level + 0.05*NOP-level) after the last instruction of the program was executed. The average $\delta$ variation over the three frequencies was approximately 1398 ns and the variation showed little dependency on frequency. The average NOP-levels at 100MHz, 200MHz, and 300MHz were approximately 37.5mA, 73.7mA, and 104.0mA, respectively.
Table 4.1. Max, mean, NOP-level, energy and $\delta$ measurement for programs at 3 frequencies.

Table 4.2 presents the summary of the approximate time durations and the maximum values for the waveforms presented in Figure 4.4. The table records the maximum values for the instructions and two time variation measurements. The time variations $\delta_5$ and $\delta_{10}$ represented the durations while the current waveforms were above 5% of the maximum values, and 10%, respectively. The mean value of the $\delta_5$ values at 200MHz was 1452.25ns. Over the three frequencies the mean values for $\delta_5$ and $\delta_{10}$ were 1368ns, and 1604 ns, respectively. Combining these results with the results presented in Table 4.1, a time variation value $\delta \in [1400\text{ns},1600\text{ns}]$ was considered to be a good approximation for the $\delta$. The conclusion of these measurements was that the atomic functions used to approximate the current consumption at the instruction level must preserve the shape, the current variation amplitude and the time variation $\delta$.

Table 4.2. Current variation measurements for different assembly language instructions at different frequencies.
Since similar results of NOP-level and current variation measurements were obtained on all other programs and individual instructions analyzed, the ‘total current superposition principle’ was introduced. This principle is described electrically by the abstract circuitry presented in Figure 4.5. This circuit modeled the processor’s instantaneous current at its highest level of abstraction where the total current of the processor was assumed to have 2 basic components:

- The constant component that was controlled by the ‘BASE BLOCK’.
- The variable component that was controlled by the ‘DYNAMIC BLOCK’.

The ‘BASE BLOCK’ was assumed to collectively represent the functions of the hardware logic that were constantly active when strings of non-operational instructions were executing on the processor. The value of the current through this block was measured for the SC140 processor at 200MHz and it was found to be within the following current interval: \( 72mA \leq I_0 \leq 74mA \).

The ‘DYNAMIC BLOCK’ was assumed to collectively represent the functions of the hardware logic that were activated specifically for performing the functionality of the operational instructions. The current through this block proved to have a complex dependency on hardware...
logic, frequency, instruction types, addressing modes, data operands, instruction sequencing, and so on.

Following the abstract circuitry of Figure 4.5, the total current consumption of the processor at its first level of abstraction was approximated by the following general current relationship:

\[ i(t) = I_0 + i_{SW}(t) \]  \hspace{1cm} (4.1)

In Equation (4.1), \( I_0 \) was the constant current that was equal with the NOP-level value and \( i_{SW}(t) \) was the variable current of the dynamic block.

### 4.2.2 Pipelining Superposition Principle

The pipelining superposition principle was applied to the dynamic component \( i_{SW}(t) \) of the total current and it was derived from experimental measurements. The goal of the principle was to simplify the current simulation of the variable component \( i_{SW}(t) \) by simulating the functionality and the switching activity of the pipeline hardware logic of the processor. Based on this principle, the current model assumed that the pipeline switching activity for each instruction could be approximated by atomic current waveforms and that the switching activity for consecutive instructions that did not execute in parallel was delayed in time by at least one clock cycle period. The exact delay time was assumed to be equivalent with the number of execution clock cycles reported for each instruction plus the number of clock cycle stalls due to pipeline hazards and memory conflicts (See Section 3.1).

In support of this principle, Figure 4.6 presents an exact print of the oscilloscope display with all of the measurements performed for the example considered. Current waveforms, maximum current values (Max), and area measurements covered by the current waveforms (Area) are presented. The example analyzed considered current measurements for a short sequence of three assembly language instructions denoted as I1, I2, and I3. The actual assembly language instruction denoted by I1, I2, and I3 were in this order: mac d4,d5,d6; mpy d0,d1,d3; and move.l d6,(r0).
Figure 4.6. Pipelining superposition principle applied to a sequence of 3 instructions: I1: mac d4,d5,d6; I2: mpy d0,d1,d3; I3: move.l d6,(r0).

The measurements performed included the following situations:

- M4 trace - recorded the current measurement of the entire sequence I1, I2, and I3 executed serially.
- M5 trace - recorded the current measurement of the sequence I1, NOP, NOP executed serially.
- M6 trace - recorded the current measurement of the sequence NOP, I2, NOP executed serially.
- M7 trace - recorded the current measurement of the sequence NOP, NOP, I3 executed serially.

Table 4.3 summarizes the Area and Max value measurements recorded for the above traces. The table results show that the measurements of maximum and area for the traces considered met the conditions of the 'pipelining superposition principle': $M_4 \cong M_5 + M_6 + M_7$. Similar results with good approximations were obtained when other sequences of instructions were considered.
<table>
<thead>
<tr>
<th>Pipeline sequence</th>
<th>Max current value [uA]</th>
<th>Area measurement [pAs]</th>
</tr>
</thead>
<tbody>
<tr>
<td>M5: I1, NOP, NOP</td>
<td>298.7</td>
<td>195.8</td>
</tr>
<tr>
<td>M6: NOP, I2, NOP</td>
<td>210.4</td>
<td>151.7</td>
</tr>
<tr>
<td>M7: NOP, NOP, I3</td>
<td>320.8</td>
<td>216.7</td>
</tr>
<tr>
<td>Total = M5+M6+M7</td>
<td>829.9</td>
<td>564.2</td>
</tr>
<tr>
<td>M4: I1, I2, I3</td>
<td>824.0</td>
<td>558.8</td>
</tr>
<tr>
<td>Approximating error</td>
<td>0.71%</td>
<td>0.96%</td>
</tr>
</tbody>
</table>

Table 4.3. Max and Area measurements supporting the pipelining superposition principle applied to a sequence of 3 instructions.

4.2.3 Random Operand Value and High Bank Register Superposition Principle

This section presents the random operand value and high bank register superposition principle that was applied to the dynamic component $i_{sw}(t)$ of the individual machine instructions. The goal of the principle was to incorporate the current variation effects per instruction due to the operand values, register names, and memory locations of the instructions. Based on this principle, the atomic functions approximated averaged waveforms of individual instructions operating on mixed and random operand values. This approximation was assumed to compensate for the overall effect of the above effects. When high bank registers were used by an execution set, there was an increase in the current consumption as related to the execution set with no high bank registers [19]. As a result, the current variation increase due to the presence of high bank registers was approximated by a fixed atomic function that had a maximum current parameter of 0.1795mA at 200MHz.

In support of this principle a summary of measurement results performed for the exclusive-or assembly language instruction “eor da,db” is presented in Table 4.4. This table presents RMS and area results. The measurements considered various data operands with and without the presence of high bank registers. The results showed that the current consumption per instruction increased with the number of zero bits present in the data operands (row 2 versus row 3 and row 4), with the number of switching bits (row 3 versus row 4), and when a high bank register was used (row 5 versus row 2, row 3 and row 4). The experimental results presented in this table were supported by
many similar current measurements performed at the instruction level. As seen from the table, the input data operands had a significant impact on the variable current amplitude. Variations of up to 30% were encountered. Since programs perform mainly on mixed data values, a mixed/random type composition for the operands was used when the current consumption at the instruction level were measured.

<table>
<thead>
<tr>
<th>Instruction</th>
<th>Operand 1</th>
<th>Operand 2</th>
<th>Result</th>
<th>RMS [mA]</th>
<th>Area [pAs]</th>
<th>Reg.&gt;7 ?</th>
</tr>
</thead>
<tbody>
<tr>
<td>eor d3,d1</td>
<td>x:00000000</td>
<td>x:00000000</td>
<td>x:00000000</td>
<td>0.103</td>
<td>118.2</td>
<td>No</td>
</tr>
<tr>
<td>eor d3,d1</td>
<td>x:aaaaaaaa</td>
<td>x:55555555</td>
<td>x:fffffff</td>
<td>0.093</td>
<td>102.9</td>
<td>No</td>
</tr>
<tr>
<td>eor d3,d1</td>
<td>x:3c626421</td>
<td>x:56095bf1</td>
<td>x:686b5bf1</td>
<td>0.086</td>
<td>87.12</td>
<td>No</td>
</tr>
<tr>
<td>eor d3,d11</td>
<td>x:3c626421</td>
<td>x:56095bf1</td>
<td>x:686b5bf1</td>
<td>0.185</td>
<td>207.3</td>
<td>Yes</td>
</tr>
</tbody>
</table>

Table 4.4. RMS and Area measurements supporting the operand values and high bank register superposition principle. The value is represented in hexadecimal as indicated by x: value.

4.2.4 Parallel Issue Superposition Principle

The parallel issue superposition principle applied to the dynamic component $i_{sw}(t)$ of the total current is described in this section. The goal of the principle was to capture the switching activity due to the logic that controlled the parallel functionality of the processor. For the SC140 processor all of the instructions that could execute in parallel were grouped in execution sets (ES) and multiple functional units were activated in parallel in accordance with the instructions issued simultaneously (See Section 3.2).

The validity of this current superposition principle was supported by real current measurements conducted at the instruction and execution set level. For example, Table 4.5 provides measurements of RMS, maximum (Max), and area performed on the following execution set ES:

- ES : [ clr d1 move.w #>$3e8,d2 clr d3 move.w #<$0,d0 ]

where ‘clr’ was a clear register instruction and ‘move.w’ was an immediate register load instruction. The measurements were performed on the individual components I1, I2, I3 and I4, their sum (TOTAL) and the entire execution set ES.
Based on the parallel issue superposition principle the current waveform of the execution set could be approximated by the sum of the individual current waveforms. As explained in Section 4.2.1 it was assumed that the current time variation per execution set was equal with the current time variation per individual instruction. From the results of this experiment it was concluded that the following current waveform summation provided a good approximation for the parallel instruction waveform M4 of the execution set ES:

- \( M4(ES) \cong M4(I1) + M4(I2) + M4(I3) + M4(I5) \)

where \( M4(In) \) represented the current variation waveform of the instruction \( In \).

<table>
<thead>
<tr>
<th>Execution set</th>
<th>RMS [mA]</th>
<th>Max [mA]</th>
<th>Area [pAs]</th>
</tr>
</thead>
<tbody>
<tr>
<td>I1: [ clr d1 ]</td>
<td>0.173</td>
<td>0.295</td>
<td>204.2</td>
</tr>
<tr>
<td>I2: [ move.w #&gt;$3e8,d2 ]</td>
<td>0.209</td>
<td>0.354</td>
<td>247.2</td>
</tr>
<tr>
<td>I3: [clr d3]</td>
<td>0.174</td>
<td>0.297</td>
<td>203.8</td>
</tr>
<tr>
<td>I4: [ move.w #$0,d0 ]</td>
<td>0.113</td>
<td>0.198</td>
<td>126.3</td>
</tr>
<tr>
<td>TOTAL</td>
<td>0.669</td>
<td>1.144</td>
<td>781.5</td>
</tr>
<tr>
<td>ES: [ clr d1 move.w #=&gt;$3e8,d2 clr d3 move.w #$0,d0 ]</td>
<td>0.729</td>
<td>1.258</td>
<td>842.9</td>
</tr>
</tbody>
</table>

Table 4.5. RMS, Max, and Area measurements supporting the parallel issue superposition principle.

### 4.3 Circuit State Dependency on Parallelism and Parallelism Variation

This section presents some characteristics of the circuit state dependency of the dynamic component \( i_{SIV} (t) \) on parallelism. Specifically, the number of instructions per execution set and the parallelism variation (difference between the numbers of instructions within consecutive execution sets) were used as variables. It was observed that sequences of execution sets and the composition of the execution sets had a certain impact on the current consumption that was not reflected by the current superposition principles [21]. The SC140 processor did not have cache support nor hardware support for data hazards, control hazards, and structural hazards. However, the processor used pipeline stalls when memory conflicts were present (See Section 3.1 and
Section 5.3). These stalls were simulated by simply delaying the superposition waveform (atomic function) by a number of clock cycles that were equivalent with the length of the stall. As a result, the only circuit state effect considered was the effect of circuit state due to execution set composition and sequencing. In [6], the switching activity of a circuit was considered to be a function of the present inputs and the previous state of the circuit, which defined circuit state dependency [6]. Similarly, in the current dynamics model the actual current measurement of an instruction might be different from the measurement used. This was because at the time when the measurements for the instruction data base were collected, the instruction sequencing and the execution sets composition of the measuring template was different than that of the real programs and thus, the overall switching activity in the real programs was different. This implied the need for a current adjustment in the simulation process after the current superposition principles were applied. The adjustment applied was called a factorial adjustment since a multiplicative factor that was dependent on the variables defined above was used in the simulation process at the linear block level (See Section 4.4 and Section 5.3).

In the general study of the circuit state dependency for the SC140 processor, groups of execution sets were subjected to two sets of current measurements. First, each execution set was measured individually and its current trace and maximum value were recorded. Secondly, sequences of two execution sets were formed. Within a group study, the first execution set was fixed and the second execution set was changed. However, the difference between the number of instructions present in the second execution set and the number of instructions present in the first execution set was constant throughout a group study. This difference was called a parallel variation (pv) number. The differences between the execution sets, per sequence, based on the instruction type and operands used, increased proportionally with the last number of the sequence’s label.

This section presents current measurement results of four group studies called par7, par8, par11, and par12 that were part of the general study. The following outlines the execution set composition and the sequences used in each group study. Each execution set had a label that identified the group and the execution set number. For example, par7.4 referred to the execution set number four that was part of the group study par7. The sequences used for each group study were also identified by a label that referred to the group study and the execution sets used in the
sequence. For example, par7.1.6 was a sequence pertaining to the group study par7 and it was formed by the execution sets par7.1 and par7.6.

**par7 execution sets:**
par7.1: [ add d0,d1,d2  mac d4,d5,d6 ]
par7.2: [ add d0,d1,d2  mac d4,d5,d6 ]
par7.3: [ add d0,d1,d2  suba r1,r0 ]
par7.4: [ mpy d6,d5,d7  mac d4,d5,d6 ]
par7.5: [ mpy d6,d5,d7  max d0,d4 ]
par7.6: [ add d0,d1,d2  move.w (r0),d6 ]
par7.7: [ mpy d6,d5,d7  move.l (r1)+,d4 ]

**par7 sequences:**
par7.1.2 = [ par7.1  par7.2 ];  par7.1.3 = [ par7.1  par7.3 ];
par7.1.4 = [ par7.1  par7.4 ];  par7.1.5 = [ par7.1  par7.5 ];
par7.1.6 = [ par7.1  par7.6 ];  par7.1.7 = [ par7.1  par7.7 ];

**par8 execution sets:**
par8.1: [ add d0,d1,d2  mac d4,d5,d6 ]
par8.2: [ add d0,d1,d2  mac d4,d5,d6  tfra r6,r7 ]
par8.3: [ add d0,d1,d2  suba r1,r0  tfra r6,r7 ]
par8.4: [ mpy d6,d5,d7  mac d4,d5,d6  eor d1,d5 ]
par8.5: [ mpy d6,d5,d7  max d0,d4  eor d1,d5 ]
par8.6: [ add d0,d1,d2  move.w (r0),d6  move.w (r1)+,d4 ]
par8.7: [ mpy d6,d5,d7  move.l (r0)+,d6  move.l (r1),d4 ]

**par8 sequences:**
par8.1.2 = [par8.1  par8.2];  par8.1.3 = [par8.1  par8.3];
par8.1.4 = [par8.1  par8.4];  par8.1.5 = [par8.1  par8.5];
par8.1.6 = [par8.1  par8.6];  par8.1.7 = [par8.1  par8.7];

**par11 execution sets:**
par11.1: [ add d0,d1,d2  mac d4,d5,d6 ]
par11.2: [ add d0,d1,d2  mac d4,d5,d6  add d0,d1,d4  mac d2,d6,d7  suba r1,r0 ]
par11.3: [ add d0,d1,d2  mac d4,d5,d6  max d0,d4  mpy d6,d5,d7  tfra r6,r7 ]
par11.4: [ mac d4,d5,d6  max d0,d4  mpy d6,d5,d7  eor d1,d5  tfra r6,r7 ]
par11.5: [ mpy d6,d5,d7  eor d1,d5  max d0,d4  suba r1,r0  tfra r6,r7 ]
par11.6: [ max d0,d4  mpy d6,d5,d7  eor d1,d5  tfra r6,r7  move.w (r0),d6 ]
par11.7: [ mpy d6,d5,d7  max d0,d4  eor d1,d5  move.w (r0),d6  move.l (r1)+,d2 ]

**par11 sequences:**
par11.1.2 = [ par11.1  par11.2 ];  par11.1.3 = [ par11.1  par11.3 ];
par11.1.4 = [ par11.1  par11.4 ];  par11.1.5 = [ par11.1  par11.5 ];
par11.1.6 = [ par11.1  par11.6 ];  par11.1.7 = [ par11.1  par11.7 ];
par12 execution sets:
par12.1: [ add d0,d1,d2 mac d4,d5,d6 ]
par12.2: [ add d0,d1,d2 mac d4,d5,d6 add d0,d1,d4 mac d2,d6,d7 suba r1,r0 tfra r6,r7 ]
par12.3: [ add d0,d1,d2 mac d4,d5,d6 max d0,d4 mpy d6,d5,d7 suba r1,r0 tfra r6,r7 ]
par12.4: [ mac d4,d5,d6 max d0,d4 mpy d6,d5,d7 eor d1,d5 suba r1 r0 tfra r6,r7 ]
par12.5: [ mpy d6,d5,d7 eor d1,d5 max d0,d4 not d0,d2 suba r1,r0 tfra r6,r7 ]
par12.6: [ max d0,d4 mpy d6,d5,d7 eor d1,d5 not d0,d3 tfra r6,r7 move.w (r0),d6 ]
par12.7: [ mpy d6,d5,d7 max d0,d4 eor d1,d5 not d0,d3 move.w (r0),d6 move.l (r1)+,d2 ]

par12 sequences:
par12.1.2 = [ par12.1 par12.2 ];  par12.1.3 = [ par12.1 par12.3 ];
par12.1.4 = [ par12.1 par12.4 ];  par12.1.5 = [ par12.1 par12.5 ];
par12.1.6 = [ par12.1 par12.6 ];  par12.1.7 = [ par12.1 par12.7 ];

The pv numbers for the above groups were in order: 0, 1, 3, and 4. Current measurements were performed for each execution set and sequence formed and their current traces and maximum values were recorded. An example of such measurements is presented in Figures 4.7 and 4.8 where the measurements of the group study par11 are presented. In these figures, exact prints of the oscilloscope screen with the measurements performed are shown. Similar measurements were performed for all of the groups analyzed.

The circuit state dependency analysis was based on comparisons between two currents:

- **Base Current**: The base current was the calculated current per sequence obtained by adding the two individual currents of the execution sets that formed the sequence.
- **Real Current**: The real current was the current measured per sequence.

Table 4.6 summarizes the max values recorded in the measurements 1 and measurements 2 for the groups considered in this section. The execution sets and the sequences used were identified in this table by their labels. The base currents calculated by using the maximum values of column 1 were recorded in column 5. The difference error and the relative error between the base currents and the real currents were calculated and presented in column 6 and column 7. All of these measurements were performed by the use of the AC current probe and all of the analysis was done on the dynamic component of the total current. When the DC component of the current consumption per instruction was considered, the relative errors of column 6 were reduced substantially. For example, the maximum relative error of 24.93% that was registered for the par8.1.4 sequence became 0.43% when a DC value of 74 mA was considered.
Figure 4.7. Current measurements for par11 execution sets

Figure 4.8. Current measurements for par11 sequences.
<table>
<thead>
<tr>
<th>Execution Set</th>
<th>Max AC Value/ES [mA]</th>
<th>Sequence</th>
<th>Real Current Max [mA]</th>
<th>Base Current Max [mA]</th>
<th>Difference Base-Real [mA]</th>
<th>Relative Error [%]</th>
</tr>
</thead>
<tbody>
<tr>
<td>par7.1</td>
<td>0.4422</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>par7.2</td>
<td>0.4355</td>
<td>par7.1.2</td>
<td>1.0410</td>
<td>0.8777</td>
<td>0.1633</td>
<td>15.68</td>
</tr>
<tr>
<td>par7.3</td>
<td>0.4682</td>
<td>par7.1.3</td>
<td>1.0050</td>
<td>0.9104</td>
<td>0.0946</td>
<td>9.41</td>
</tr>
<tr>
<td>par7.4</td>
<td>0.3381</td>
<td>par7.1.4</td>
<td>0.9571</td>
<td>0.7803</td>
<td>0.1768</td>
<td>18.47</td>
</tr>
<tr>
<td>par7.5</td>
<td>0.3639</td>
<td>par7.1.5</td>
<td>0.9112</td>
<td>0.8061</td>
<td>0.1051</td>
<td>11.53</td>
</tr>
<tr>
<td>par7.6</td>
<td>0.7114</td>
<td>par7.1.6</td>
<td>1.2250</td>
<td>1.1536</td>
<td>0.0714</td>
<td>5.83</td>
</tr>
<tr>
<td>par7.7</td>
<td>0.6600</td>
<td>par7.1.7</td>
<td>1.1720</td>
<td>1.1022</td>
<td>0.0698</td>
<td>5.95</td>
</tr>
<tr>
<td>average</td>
<td></td>
<td></td>
<td>1.0518</td>
<td>0.9384</td>
<td>0.1134</td>
<td>11.14</td>
</tr>
<tr>
<td>par8.1</td>
<td>0.4422</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>par8.2</td>
<td>0.6285</td>
<td>par8.1.2</td>
<td>1.3570</td>
<td>1.0707</td>
<td>0.2863</td>
<td>21.09</td>
</tr>
<tr>
<td>par8.3</td>
<td>0.7710</td>
<td>par8.1.3</td>
<td>1.4470</td>
<td>1.2132</td>
<td>0.2338</td>
<td>16.15</td>
</tr>
<tr>
<td>par8.4</td>
<td>0.5336</td>
<td>par8.1.4</td>
<td>1.3000</td>
<td>0.9758</td>
<td>0.3242</td>
<td>24.93</td>
</tr>
<tr>
<td>par8.5</td>
<td>0.5641</td>
<td>par8.1.5</td>
<td>1.2630</td>
<td>1.0063</td>
<td>0.2567</td>
<td>20.32</td>
</tr>
<tr>
<td>par8.6</td>
<td>1.1970</td>
<td>par8.1.6</td>
<td>1.8240</td>
<td>1.6392</td>
<td>0.1848</td>
<td>10.13</td>
</tr>
<tr>
<td>par8.7</td>
<td>1.1430</td>
<td>par8.1.7</td>
<td>1.7500</td>
<td>1.5852</td>
<td>0.1648</td>
<td>9.42</td>
</tr>
<tr>
<td>average</td>
<td></td>
<td></td>
<td>0.2417</td>
<td>0.1700</td>
<td></td>
<td></td>
</tr>
<tr>
<td>par11.1</td>
<td>0.4422</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>par11.2</td>
<td>1.3870</td>
<td>par11.1.2</td>
<td>2.2540</td>
<td>1.8293</td>
<td>0.4247</td>
<td>18.84</td>
</tr>
<tr>
<td>par11.3</td>
<td>1.4570</td>
<td>par11.1.3</td>
<td>2.3800</td>
<td>1.8992</td>
<td>0.4808</td>
<td>20.20</td>
</tr>
<tr>
<td>par11.4</td>
<td>1.4560</td>
<td>par11.1.4</td>
<td>2.3040</td>
<td>1.8982</td>
<td>0.4058</td>
<td>17.61</td>
</tr>
<tr>
<td>par11.5</td>
<td>1.3930</td>
<td>par11.1.5</td>
<td>2.0900</td>
<td>1.8352</td>
<td>0.2548</td>
<td>12.19</td>
</tr>
<tr>
<td>par11.6</td>
<td>1.7100</td>
<td>par11.1.6</td>
<td>2.4140</td>
<td>2.1522</td>
<td>0.2618</td>
<td>10.85</td>
</tr>
<tr>
<td>par11.7</td>
<td>1.9580</td>
<td>par11.1.7</td>
<td>2.7030</td>
<td>2.4002</td>
<td>0.3028</td>
<td>11.20</td>
</tr>
<tr>
<td>average</td>
<td></td>
<td></td>
<td>0.3551</td>
<td>0.1514</td>
<td></td>
<td></td>
</tr>
<tr>
<td>par12.1</td>
<td>0.4465</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>par12.2</td>
<td>1.8850</td>
<td>par12.1.2</td>
<td>2.3400</td>
<td>2.3315</td>
<td>0.0085</td>
<td>0.36</td>
</tr>
<tr>
<td>par12.3</td>
<td>1.8630</td>
<td>par12.1.3</td>
<td>2.3260</td>
<td>2.3095</td>
<td>0.0165</td>
<td>0.71</td>
</tr>
<tr>
<td>par12.4</td>
<td>1.8680</td>
<td>par12.1.4</td>
<td>2.4660</td>
<td>2.3145</td>
<td>0.1515</td>
<td>6.14</td>
</tr>
<tr>
<td>par12.5</td>
<td>1.6970</td>
<td>par12.1.5</td>
<td>2.2390</td>
<td>2.1435</td>
<td>0.0955</td>
<td>4.26</td>
</tr>
<tr>
<td>par12.6</td>
<td>1.9460</td>
<td>par12.1.6</td>
<td>2.5170</td>
<td>2.3925</td>
<td>0.1245</td>
<td>4.95</td>
</tr>
<tr>
<td>par12.7</td>
<td>2.3600</td>
<td>par12.1.7</td>
<td>3.0340</td>
<td>2.8065</td>
<td>0.2275</td>
<td>7.50</td>
</tr>
<tr>
<td>average</td>
<td></td>
<td></td>
<td>0.1040</td>
<td>3.99</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Table 4.6. Measurement analysis for parallelism dependency.
The results of Table 4.6 show an increase in the average of the difference between the real current and base current with the increase of the pv number. Nevertheless, when more AGU type instructions [56] were present (par12 group) in the execution sets, the base current was closer to the real current due to the fact that the AGU type instructions used had lower current consumptions in comparison with the ALU type and MOVE type instructions.

Based on this analysis it was concluded that the base current must be adjusted at the program level in order to properly account for the circuit state dependency. As indicated in Table 4.6, the error of approximation was not impacted in a positive way when similar instructions were executed in both execution sets of a sequence. Mainly, the approximating error was dependent on the parallel composition and on the parallel variation. In order to account for this dependency, the programs were divided in linear blocks. A linear block was different than the base block defined in Section 4.1 and was composed of clusters of instructions that had similar parallelism values. Averages of numbers of instructions per execution set and per clock cycle and averages of parallelism variation per linear block were used as parameters in the factorial dependency. Complete numerical examples of current models for various programs are presented in Chapter 5. As a result, the base currents of the programs were adjusted at the linear block level by multiplying the base current with constant factors derived from the parallelism composition and parallelism variation of the program.

### 4.4 The Instruction-Level Based Current Model: Gamma Functions

This section presents the abstract circuitry and the mathematical development of the instantaneous current model. The instantaneous current consumption model was considered a current dynamics macro-model since the variations of the processor’s current while executing a program application were modeled by the use of large pieces of current waveforms that approximated the current variations at the instruction level. The practical development of the current model was based on the superposition principles presented in Section 4.2. The current model assumed that the current variation $i_{sw}(t)$ (See Equation 4.1) per instruction had the following components:

- A control logic component that was due to the activity of the specific hardware logic activated in order to perform all of the functionality required by the instruction.
• A data dependency component that was due to the specific data values of the operands handled by the instruction.

• A circuit state dependency component that was due to the composition and sequencing of the execution sets executed prior and after the execution set considered.

• A memory dependency component that was due to the physical position occupied by the machine code in the memory of the computer.

Experimental measurements showed that the most influential components of the variable current \( i_{sw}(t) \) were the control logic component, the data dependency component, and the execution set composition and sequencing component. As a result, the model presented in this thesis focused on simulating these three components in three ways. First, the model considered the logic component by using superposition principles that were derived from the control logic behavior of the processor. Secondly, the model accounted for the data dependency impact on the dynamic current \( i_{sw}(t) \) by deriving the parameters of the atomic functions from averaged waveforms of a large number of current samples (400 samples) for each type of instruction operating on random and mixed data. Thirdly, the model accounted for the circuit state dependency component by the use of factorial adjustments as functions of the instruction parallelism and parallelism variation, applicable to linear blocks.

The current simulation macro-model developed the instantaneous current consumption at the program level in two major steps. First, a base current waveform was simulated by the use of the four current superposition principles (See Section 4.2). Secondly, the final current waveform was generated from the base current waveform by applying the factorial adjustments at the linear block level (developed in Chapter 5 based on the experimental evidence presented in Section 4.3). The following presents the development of the model and the equations of the current simulation.

Figure 4.9 summarizes the application of the superposition principles to the simulation process of the current consumption of the SC140 DSP processor while executing a program AL. In this figure, \( PA_i^k \) represents the load due to the pipeline switching activity of an individual instruction \( I_i \) that was part of an execution set \( ES_k \). In Figure 4.9, the model accommodated the maximum parallel capability of the SC140 processor and did not include any auxiliary effects.
The abstract current model of Figure 4.9 provided the first base current simulation formula applicable at the program level:

\[
i(t) = I_0 + \sum_{i=1}^{n} \sum_{k=1}^{6} SW(i,k,t) \cdot i_{sw}(PA_k^i,t)
\]  

The current Equation (4.2) used a step type switching function \( SW(i,k,t) \) that controlled the moment of time when the switching load \( PA_k^i \) was activated. The activation time depended on the sequencing and the composition of the executable code, the execution time (in clock cycles) of the instructions, and the pipeline hazards. The values of the switching functions were determined by tracing the executable code of the program. The \( I_0 \) current component was the measured NOP-level for the working frequency of the processor. In Figure 4.9, at the beginning of each clock cycle a new execution set was activated. The load introduced by individual instructions that were
active within an execution set acted as a parallel load within the set. The load introduced by the pipeline activity related to the entire execution set acted as a parallel load to the execution sets activated in the previous and future clock cycles.

The current dynamics model assumed a uniform current shape across the entire instruction set architecture. The only instruction dependent value was assumed to be the maximum current consumption per instruction. The real current waveforms of the dynamic component $i_{SW}(t)$ per instruction or execution set had a current variation shape like the one outlined in Figure 4.10 (See also Section 4.2.1 and Section 5.1). These current waveforms were approximated by atomic functions. Each instruction type had its own current characteristic and within the type, there was a significant current variation depending on data and the addressing modes used by the operands.

The model considered different atomic functions for each possible addressing mode used by a particular type of instruction. The atomic functions chosen covered the entire current range of the SC140 instruction set architecture. A good representative of the atomic functions was a continuous mathematical function called a gamma function. The gamma function used the maximum current value as an instruction dependent parameter. Mathematically, the gamma functions were generated by the following function [63]:

$$g(x; n, \lambda) = \frac{\lambda^x}{n!} e^{-\lambda x}; x \geq 0, \quad \lambda > 0;$$  \hspace{1cm} (4.3)
In Equation (4.3), \( n \) is the order of the gamma function and characterizes the shape of the waveform. Specific values for \( n \) and \( \lambda \) were determined to make the gamma functions as close as possible to the shape of the dynamic component \( i_{SW}(t) \) of the current measurements at the instruction level. Experimental measurements showed that the dynamic component of the current consumption for a general type instruction \( I_k \) that is part of the SC140 instruction set could be approximated across the entire working frequency range \([40\text{MHz}, 300\text{MHz}]\) by the following atomic function:

\[
i_k(t) = \begin{cases} 
K \cdot g(t:n,\lambda); & \text{for } 0 \leq t \leq \delta; \\
0 & \text{otherwise};
\end{cases}
\]

(4.4)

As a result, the atomic function of Equation (4.4) formed the basic building block of the current dynamics model. The current approximation of Equation (4.4) used a base function \( g(t:n,\lambda) \) with fixed parameter values \( n \) and \( \lambda \) at a given frequency. The instruction dependent parameter of the current approximation was \( K \). This parameter was a function of the peak dynamic current value per instruction. Also \( K \) could be expressed as a function of the RMS measurements. In this case, after \( n \) and \( \lambda \) are determined the RMS measurements needed for generating \( K \) could be collected by ammeters.

Since the oscilloscope collected and stored the current waveforms as sets of sampled points, other possible atomic functions were finite sets of sampled points derived from a measured current trace. For example, In Figure 4.10 it was assumed that the activity of the processor was coordinated by a clock signal of period \( \tau \), the distance between two sample points was \( \sigma \), and the total time variation of the current above the NOP-level was \( \delta \). Based on these assumptions, an example of an atomic function as a finite set of sample points representing a general type instruction \( I_k \) could be defined as follows:

\[
i_k(t) = \begin{cases} 
i_{SW}(k) = \{i_k(0), i_k(\sigma), i_k(2\sigma), \ldots, i_k(\delta - \sigma), i_k(\delta)\}; & \text{for } 0 \leq t \leq \delta; \\
0 & \text{otherwise.}
\end{cases}
\]

(4.5)

A model based on finite sets of sampled points required more computer memory and the collection and analysis of data was more complex in comparison with a model based on gamma
functions. As a result, this thesis presents results of current simulation based only on gamma functions (See Chapter 5).

It was assumed that the parameters of the atomic functions for all of the instruction types of the SC140 instruction set could be determined through experimental measurements and atomic functions could be generated at the instruction level. Based on these assumptions, a systematic approach that used three levels of abstract circuitry was developed in order to simulate instantaneous current waveforms at the program level. First, Figure 4.11 presents the abstract circuitry used to simulate current consumption at the execution set level. This circuitry used the parallel issue superposition principle to simulate the total current consumption of the processor at the execution set level. The circuitry of Figure 4.11 assumed that an execution set $ES_k$ was composed of at most 6 instructions that could be executed in parallel. These instructions were
denoted by I₁, I₂, I₃, I₄, I₅, and I₆ and their corresponding atomic functions by \( i_1(t) \), \( i_2(t) \), \( i_3(t) \), \( i_4(t) \), \( i_5(t) \) and \( i_6(t) \). The circuitry also used two auxiliary atomic functions, \( i_7(t) \) and \( i_8(t) \), which were executed in parallel with the execution set and were responsible for current dynamics due to prefix grouping (See Section 3.1 and 4.2.3) and looping instructions (LPMARKA, LPMARKB [47]). The simulated current at the execution set level was calculated by applying Kirchhoff’s current law [64] for an electric circuit to the abstract circuit presented in Figure 4.11. The current approximation at the execution set level was given by the \( f^{(i)}_{ES}(t) \) function that could use any atomic functions \( i_i(t) \) at the instruction level. The simulated current consumption at the execution set level was \( i^{(i)}_{ES}(t) \). The following shows the mathematical expressions of the variable current component at the execution set level.

\[
\begin{align*}
    I_{SW}(ES_k) &= \sum_{i=1}^{i=8} SW(i,k,t_k) \cdot i_i(t); \text{ for } t_k \leq t \leq t_k + \delta; \\
    SW(i,k,t_k) &= \begin{cases} 
    1 & \text{if instruction } I_i \in ES_k \\
    0 & \text{otherwise}
    \end{cases} \\
    f^{(i)}_{ES}(t) &= \sum_{i=1}^{i=8} SW(i,k,t_k) \cdot i_i(t); \text{ for } t_k \leq t \leq t_k + \delta; \\
    i^{(i)}_{ES}(t) &= f^{(i)}_{ES}(t) 
\end{align*}
\] (4.6)

Secondly, Figure 4.12 presents the abstract circuitry used to simulate the variable current consumption at the linear block level. This circuitry used the pipeline superposition principle and the factorial adjustment on parallelism (See Sections 4.2 and 4.3) to simulate the variable current component of the processor at the linear block level. The linear blocks within a program were composed of clusters of execution sets that executed sequentially. The principles of the linear block division within a program were outlined in Section 4.4 and were further detailed in Chapter 5. The circuitry of Figure 4.12 was the most detailed circuitry used in the current model. The circuitry assumed that a linear block \( LB_k \) was composed of \( (n+1) \) execution sets and these execution sets were fetched and started their execution at deterministic times \( t_0, t_1, \cdots, t_n \).
The variable component of the simulated current at the linear block level was calculated by applying Kirchhoff’s current law for an electrical circuit to the abstract circuitry of Figure 4.12:

\[
i_{ELB}^{(k)}(t) = \Lambda_{(k)}(parallelism, parallelism variation) \cdot \left( \sum_{p=0}^{n} SW(p) \cdot i_{ES}^{(p)}(t) \right);
\]

\[
SW(p) = \begin{cases} 
1 & \text{for } t \geq (t_p + \sum_{j=0}^{n} \Delta^{(j)}(nop)) \\
0 & \text{otherwise}
\end{cases}
\]

In Equation (4.8), \( i_{ES}^{(p)}(t) \) was defined by the Equation (4.6), \( \Lambda_{(k)} \) represented the factorial adjustment for the linear block \( LB_k \) that accounted for the circuit state dependency on parallelism and parallelism variation, and \( \Delta^{(j)}(nop) \) was a delay time expressed in number of NOP instructions inserted after the execution set \( ES^{(p)} \). The \( \Delta^{(i)}(nop) \) parameter was used only for execution sets with execution times greater than one clock cycle and when stalls due to memory conflicts were present.
Figure 4.13. Graphical representation of the current construction process at the linear block level.

Figure 4.13 presents the calculation process of the simulated current at the linear block level when current functions at the execution set level were used. The current simulation example used one discrete value per clock cycle from each active current function. A sample point \( k \) of an execution set function \( f_{ES} \) was denoted by \( S_{SP}^{(f_{ES})} \). For simplicity, an execution time of one clock cycle per execution set and no time delays between execution sets were assumed. A sample point of the simulated current at the linear block level was generated by summing up all of the corresponding current points active at that time. For example, in this graph the third sample point of the simulated current was generated as: 

\[
S^{(i)}_{SP} = S^{(f_{ES0})}_{SP} + S^{(f_{ES1})}_{SP} + S^{(f_{ES2})}_{SP}.
\]

Finally, Figure 4.14 presents the abstract circuitry of the simulated current at the program level. The figure assumed that the program was composed of \( (M+1) \) sequential linear blocks that started to execute at sequential times: \( t_0, t_1, \ldots, t_M \). It was also assumed that the simulated current for any linear block \( LB_k \) was \( i_{LB}^{(k)}(t) \) and was calculable by the use of Equation (4.8).
The simulated current at the program level was calculated by applying Kirchhoff's current law for an electrical circuit to the abstract circuitry of Figure 4.14:

\[
i(t) = I_0 + \sum_{k=0}^{M} SLB^{(k)} : i^{(k)}_{LB}(t);
\]

\[
SLB^{(k)} = \begin{cases} 
1 & \text{for } t \in [t_k, t_{k+1}) \\
0 & \text{otherwise}
\end{cases}
\]

In Equation (4.9), \( I_0 \) was the NOP-level current, \( i^{(k)}_{LB}(t) \) was the simulated current for the linear block \( LB_k \) (Equation (4.8)), and \( SLB^{(k)} \) was the switch that activated the linear block \( k \) of the program. Equation (4.9) was the final equation that was used to simulate the instantaneous current consumption of the SC140 DSP processor at the program level.
4.4.1 Dynamic Current Simulation Methodology

Following the basic modeling techniques presented in this chapter, a general procedural scheme for current simulation that targeted the SC140 DSP processor was developed. The block diagram of this procedural scheme is presented in Figure 4.15. An instruction reference data base (IRDB) of atomic functions for a large number of individual types of instructions executed at a reference frequency of 200MHz was generated. The IRDB stored for each instruction a set of current sample points, the maximum current value (Max), and the root-mean-square current value (RMS). The set points, the Max and the RMS values were representatives of the averaged mean filtered current waveform of the instruction operating on mixed/random operands. The averaged waveforms were used in order to eliminate the noise due to the measurement setup and to account for data dependency. Current samples at the instruction level operating on different data values were averaged in order to simplify the model and to reflect the global operand variation of real programs (See Section 4.2.3 and Section 5.1). The maximum values were used in a MATLAB type application to generate the individual gamma functions used in the current dynamics model. Using existing SC140 software tools [47], an equivalent assembly language representation was generated. Using the simulator tools [47], an execution set trace of the assembly language representation was produced for a given data input. The execution set trace was used to generate an execution trace matrix as an input to the MATLAB program application that simulated the abstract circuitries presented in Figures 4.11, 4.12, and 4.14. The execution trace matrix contained information related to the switching functions \( SW_i^{(k)} \), the starting execution times \( (t_1, t_2, \ldots, t_n) \), the \( \Delta_i^{(k)}(nop) \) delays, and the instruction identification within the IRDB. The linear block decomposition of the program was done by analyzing the distribution of the number of instructions per execution sets. Complete step-by-step examples of current model building for the SC140 processor with numerical values and error simulation analysis are presented in Chapter 5.

It is possible that the current simulation procedure presented in this chapter could be extended to other CMOS type processors. Note that the only processor dependent components of the current simulation procedure presented in this chapter were the atomic functions and the processor’s hardware architecture. The current superposition principles that reflected the SC140 hardware architecture could be transformed to accommodate different hardware architectures and the
atomic functions indicated could be matched to other current shapes, if needed. In fact, the sampled set functions could possibly be used as atomic functions for any processor that facilitates instantaneous current measurements at the instruction level. See discussion section in Chapter 7 for further details. The following chapter presents measurements, and modeling results for the SC140 DSP processor.

Figure 4.15 Current simulation methodology targeting the SC140 DSP processor.
5.1 Errors and Statistics of Instruction-Level Current Measurements

In this section, short definitions of the data analysis parameters and analysis of the instruction-level error measurements are presented. The data statistics and the error analysis targeted four types of sample points:

- IR: Column vector of n measured sample points of a current waveform w.
- IS: Column vector of n simulated sample points of a current waveform w.
- IRmIS: Column vector, defined as the difference between IR and IS.
- SP: m by n matrix of m sample points per current waveform w taken at n different times.

The following parameters are used [66]:

- Range (R): \[ R = \text{highest value} – \text{lowest value} \].
- Maximum and energy values:
\[ IR_{\text{max}} = \max(IR); \quad IS_{\text{max}} = \max(IS); \]  
\[
IR_{\text{energy}} = 1.898V \cdot \int_{T_{\text{start}}}^{T_{\text{end}+1.6\mu s}} IR(t) \, dt; \quad IS_{\text{energy}} = 1.898V \cdot \int_{T_{\text{start}}}^{T_{\text{end}+1.6\mu s}} IS(t) \, dt
\]

where 1.898V was the processor’s voltage supply, \( T_{\text{start}} \) was the start time of the waveform (program), \( T_{\text{end}} \) was the end time of the waveform (program), and the 1.6 \( \mu s \) time covered for the discharge time taken by the program after the execution ended.

- **Sample means** \((IR_{\text{mean}}, IS_{\text{mean}}, IRmIS_{\text{mean}})\):

\[
IR_{\text{mean}} = \frac{\sum IR}{n}; \quad IS_{\text{mean}} = \frac{\sum IS}{n}; \quad IRmIS_{\text{mean}} = \frac{\sum (IR - IS)}{n}
\]  

- The standard deviation \( s \) defined for the IRmIS column vector is:

\[
s = \sqrt{\frac{\sum (IRmIS - IRmIS_{\text{mean}})^2}{n-1}}
\]  

In the analysis presented in this thesis, the standard deviation \( s \) was used to compare how closely related two waveforms were and for this, the difference vector between the measured and simulated waveforms was considered.

- **Root mean square error** \((RMS_{err})\): Quantified the average magnitude of the error between the measured and the simulated waveforms.

\[
RMS_{err} = \sqrt{\frac{\sum_{i=1}^{n} (IR_i - IS_i)^2}{n}}; \quad \text{where } IR_i, IS_i : i^{th} \text{ elements of } IR \text{ and } IS
\]

- The square of the correlation \( CC^2 [%] \) was calculated from the covariance matrix \( C \) as follows:

\[
C = \text{cov}(IR, IS) = \begin{bmatrix} c_{11} & c_{12} \\ c_{21} & c_{22} \end{bmatrix}
\]

\[
CC^2 [%] = \frac{c_{12}^2}{c_{11} \cdot c_{22}} \cdot 100
\]
The square of the correlation as a percent represents the total variation of IR captured by IS. Perfect correlation \((CC^2\%) = 100\%\) meant that the variation in IR was completely accounted for by a linear relationship with IS. As the correlation coefficient \(CC^2\%)\) decreased, less variation was captured by the linear relationship.

- Signal to noise ratio (SNR):

\[
SNR = \frac{Mean}{s} = \frac{IR_{mean}}{s}
\]  

\((5.6)\)

<table>
<thead>
<tr>
<th>Instruction</th>
<th>Max Peak [mA]</th>
<th>(SNR_{mean})</th>
<th>Counts Within 1s [%]</th>
<th>Counts Within 2s [%]</th>
<th>Counts Within 3s [%]</th>
</tr>
</thead>
<tbody>
<tr>
<td>JMP label</td>
<td>1.5838</td>
<td>1.1530</td>
<td>68.21</td>
<td>96.7271</td>
<td>99.9625</td>
</tr>
<tr>
<td>NOP</td>
<td>0.6562</td>
<td>2.5800</td>
<td>67.96</td>
<td>96.7042</td>
<td>99.9688</td>
</tr>
<tr>
<td>ADD d0,d1,d2</td>
<td>0.9796</td>
<td>3.0192</td>
<td>67.48</td>
<td>97.0375</td>
<td>99.9792</td>
</tr>
<tr>
<td>MAC d14,d5,d12</td>
<td>1.0986</td>
<td>3.1464</td>
<td>67.91</td>
<td>96.8104</td>
<td>99.9771</td>
</tr>
<tr>
<td>EOR d1,d5</td>
<td>1.0712</td>
<td>2.8906</td>
<td>67.57</td>
<td>97.0604</td>
<td>99.9500</td>
</tr>
<tr>
<td>MOVE.2l d0:d1,(r0)</td>
<td>1.3732</td>
<td>3.6934</td>
<td>68.08</td>
<td>96.7687</td>
<td>99.9708</td>
</tr>
<tr>
<td>MOVEl (r0)+,d0</td>
<td>1.1840</td>
<td>3.3582</td>
<td>67.96</td>
<td>96.7708</td>
<td>99.9750</td>
</tr>
<tr>
<td>MOVEl (r0),d0</td>
<td>1.1322</td>
<td>3.2574</td>
<td>67.78</td>
<td>96.8333</td>
<td>99.9688</td>
</tr>
<tr>
<td>MOVEl (r0),d9</td>
<td>1.3946</td>
<td>3.6910</td>
<td>67.48</td>
<td>97.1146</td>
<td>99.9667</td>
</tr>
<tr>
<td>MOVE.w (r0)+,d0</td>
<td>1.1414</td>
<td>3.3803</td>
<td>68.10</td>
<td>96.7604</td>
<td>99.9729</td>
</tr>
<tr>
<td>DOEN1 #17</td>
<td>0.8606</td>
<td>2.8765</td>
<td>67.91</td>
<td>96.8771</td>
<td>99.9729</td>
</tr>
<tr>
<td>DOSETUP label</td>
<td>0.9766</td>
<td>3.0656</td>
<td>68.26</td>
<td>96.6479</td>
<td>99.9833</td>
</tr>
<tr>
<td>DECA r6</td>
<td>0.9522</td>
<td>2.8028</td>
<td>68.09</td>
<td>96.7562</td>
<td>99.9813</td>
</tr>
<tr>
<td>SUBA r1,r0</td>
<td>1.0346</td>
<td>2.9387</td>
<td>67.94</td>
<td>96.7479</td>
<td>99.9729</td>
</tr>
<tr>
<td>EOR #$aaaa,d2.l</td>
<td>1.2208</td>
<td>3.5903</td>
<td>67.90</td>
<td>96.7437</td>
<td>99.9792</td>
</tr>
<tr>
<td>OR #$aaaa,d2.l</td>
<td>1.2330</td>
<td>3.5368</td>
<td>67.93</td>
<td>96.7208</td>
<td>99.9750</td>
</tr>
</tbody>
</table>

Table 5.1. Instruction level data statistics and errors of measurement.
Table 5.1 reports current measurement analysis of the $i_{SW}(t)$ component (See Section 4.2.1) for 16 assembly language instructions. The table shows the maximum peak current variation (Max Peak), the mean value of the SNR vector ($SNR_{mean}$), plus counts of sample data lying within 1, 2, and 3 standard deviations from the mean (Counts Within 1s, Counts Within 2s, Counts Within 3s).
The results showed that 68% of the measured sampled points were within 1 standard deviation from the mean and more than 95% were within 2 standard deviations from the mean. All of these results indicate that the instruction measurements taken at different moments in time were stable and the spread of the sample data was low.

Figure 5.1 plots the current measurements collected for some of the instructions presented in Table 5.1. Each plot superimposed all of the individual waveforms collected per instruction (16 in all) by the use of the AC current probe, plus the IRmean vector (red waveform). Figure 5.2 presents the signal to noise ratio histogram for all of the instructions considered in Table 5.1. The histogram shows the distribution of the sample points within 10 equally spaced bins. As seen from this histogram the majority of the points were grouped in the second bin with SNR values greater than 2. This indicated that the current signal was strong in comparison with the measurement noise.
Figure 5.3. Example 1 of the data collection at the instruction level.

Figure 5.4. Example 2 of the data collection at the instruction level.
At the instruction level, for each measurement used in the current model, 400 individual waveforms were collected and averaged. The averaged waveform was filtered by a built-in centered (mean) filter in order to eliminate the noise and obtain consistent measurements throughout the entire instruction set. Figures 5.3 and 5.4 present examples of single, averaged, and filtered current waveforms for two individual instructions. Each figure plots with green is the single current waveform, with blue is the averaged waveform taken over 400 single collections, and with red is the mean filtered waveform. As seen from these plots, the filtered waveform was centered to the averaged waveform and eliminated the noise from the averaged waveform.

In conclusion, the filtered waveform eliminated most of the measurement noise and it could be considered as representative waveform for the necessary current measurements at the instruction level. The mean filtered and averaged waveforms were called measured waveforms. The mean filtered waveforms at the instruction and the program level were used throughout the remaining chapters of this thesis, unless otherwise stated. At the program level the averaged waveforms were used in order to eliminate the measurement noise due to interference, power supply variation, and measurement setup. Each sample of the averaged waveform was a measurement of the program executing in identical conditions and data environment. At the instruction level the averaged waveforms were used to eliminate noise and also to compensate for data and register name current dependency. Each sample of the averaged waveform was a measurement of the instruction executing in the same program conditions but operating on different data. This was to compensate for the overall operand and register name variation in real programs. Table B.1 of Appendix B presents AC maximum values (ACmax) obtained by using the filtered waveforms, and DC average values (Iavg) obtained by using the meter measurements.

5.2 Examples of Building Gamma Approximation Functions at Various Frequencies

In this section, two versions of a short assembly language application are analyzed at three different frequencies: 100MHz, 200MHz, and 300MHz. A complete step-by-step modeling approach for individual gamma approximating functions is presented. The approximating constants for the gamma functions were derived experimentally from the instruction
measurements used in these programs and were used for all other gamma functions throughout the chapters containing detailing results.

The assembly language application (AL) analyzed in this section is a short cryptographic application called powerco.asm. The application contained 7 identical hardware loops and executed for 167 clock cycles at run time. The complete listing of the powerco.asm program is presented in Appendix B. The following execution sets are used to illustrate the building of the gamma approximation functions (See Equation 4.4):

```
eco024: clr d0
eco025: [ doen3 #number dosetup3 label ]
eco026: move.l #label,r1
eco027: move.l #label+number,r2
eco028: move.l #label,r3
eco029: [ move.2l (r2)+,d0:d1 move.2l (r3)+,d2:d3 ]
eco030: [ eor d2,d0 eor d3,d1]
eco031: move.2l d0:d1,(r1)
jmp: jmp label
jsr: jsr subroutine
```

The above execution sets and their instructions were further identified in Figures 5.5, 5.6, and 5.7 and Table 5.2 by the use of their labels. For example, eco025/11 referred to the instruction ‘doen3 #number’ of the execution set ‘eco025: [doen3 #number dosetup3 label]’. Single instructions per execution set were identified only by the execution set label.

In order to provide a consistent method for building the individual gamma, the following procedure was adopted:

- The representative filtered waveform with all its sample points $IR_i$ was considered for each individual instruction $i$ and the maximum value $\text{max}_i = \text{max}(IR_i)$ was recorded.

- The constant $n$ of Equation (4.4) was set to 1. For each frequency $f$ analyzed, a $\lambda_f$ constant was determined experimentally such that the shape of the individual gamma function was as close as possible to the shape of the real current measurement.
After the $\lambda_f$ had been chosen for each working frequency (100MHz, 200MHz, 300MHz) the amplification factor $K$ of the gamma function (See Equation 4.4) was calculated as: $K = A \cdot \max_i$; where $A = 1/\max(g(x:1,\lambda_f))$.

The function $g(x:1,\lambda_f)$ (See Equation 4.3) was fixed at a given frequency for the entire SC140 instruction set architecture. As a result, the final gamma function depended only on the maximum current values measured for each instruction type. There was more than one $\lambda$ value that could give a good shape approximation. Each $\lambda$ value gave a unique $A$ value at different frequencies. From measurement analysis performed on the above execution sets, the following experimental triplet values ($\lambda, A$, frequency) had been chosen:

- (0.0038, 1.3980, 100MHz), (0.0038, 1.3972, 200MHz), and (0.0057, 2.0970, 300MHz).

These values were considered to be characteristic across the entire instruction set for these three frequencies. Using these empirically calculated values, the simulated current waveforms versus real current measurements at the instruction and execution set levels are presented in Figures 5.5, 5.6, and 5.7. The currents represent the dynamic components, the blue waveforms are the measured currents and the red waveforms (dotted) are the simulated currents.

Table 5.2 summarizes the RMS errors ($RMS_{err}$), and the square of the correlation ($CC^2 [%]$) values between the simulated current and the measured current for the instructions and the execution sets presented in Figures 5.5, 5.6, and 5.7. The table shows that the $RMS_{err}$ values were small and the $CC^2 [%]$ values were over 95% for all the cases analyzed. These indicate a good approximation of the current dynamics and magnitude by the individual gamma functions versus the measured current waveforms. The averages of the maximum current values for the instructions presented in Table 5.2 were 0.5763mA at 100MHz, 0.5590mA at 200MHz, and 0.5316mA at 300MHz. The average RMS errors were less than 6.27% of the maximum value at 100MHz, less than 3.44% of the maximum value at 200MHz, and less than 4.12% of the maximum value at 300MHz.
Figure 5.5. Gamma approximating functions at 100Mhz. Real measurements versus simulated functions.

Figure 5.6. Gamma approximating functions at 200Mhz. Real measurements versus simulated functions
As seen from the Figures 5.5, 5.6 and 5.7, the RMS errors were mainly due to differences between the simulated current and measured current that existed at the beginning and at the end of the current waveform. The current values for these portions of the waveform were very small and their impact on the current dynamics model was minimal. The measured execution time for the application powerco.asm was 167 cycles and that was 1670ns at 100MHz, 835ns at 200MHz, and 557ns at 300MHz. The application powercoN.asm had the same functionality with the powerco.asm application. The difference between these two applications was that powercoN.asm included an extra block of 40 NOPs that increased the execution time with 40 clock cycles. This block introduced a visible current variation in the powercoN.asm’s current trace and had the ability to reduce the peak current values in comparison with the powerco.asm peak values with up to 7.8% for the frequencies considered. At all frequencies analyzed, the data statistics for these programs were taken over a period of time equal with the effective execution time of the program plus 1600ns. The 1600 ns time was added to completely cover the discharging time of the processor’s capacitance (See Section 4.2.1). Figure 5.8 presents three current waveforms for powerco.asm and powercoN.asm at 100MHz, 200MHz, and 300MHz. The blue waveforms
represent the total currents measured by the DC current probe (IR vector), the red waveforms represent the total simulated current (IS vector), and the green waveforms represent the difference waveform between the total measured current and the total simulated current (IRmIS vector).

Tables 5.3 and 5.4 summarize the data statistics for these two applications (powerco.asm and powercoN.asm) at 100MHz, 200MHz, and 300MHz. Table 5.3 presents the maximum (Max) and mean (Mean) current values and Table 5.4 presents the relative errors for the maximum and mean values of the simulated waveforms versus measured waveforms. Table 5.4 also presents the standard deviation $s$ of the difference waveform ($IR-IS$) measured in mA. The small values obtained for the standard deviation $s$ indicated that the simulated waveform followed closely the measured waveform.

<table>
<thead>
<tr>
<th>Execution Set/Instr.</th>
<th>100 MHz $RMS_{err}$ [mA]</th>
<th>100 MHz $CC^2$ [%]</th>
<th>200 MHz $RMS_{err}$ [mA]</th>
<th>200 MHz $CC^2$ [%]</th>
<th>300 MHz $RMS_{err}$ [mA]</th>
<th>300 MHz $CC^2$ [%]</th>
</tr>
</thead>
<tbody>
<tr>
<td>eco24</td>
<td>0.0179</td>
<td>96.11</td>
<td>0.0143</td>
<td>97.41</td>
<td>0.0085</td>
<td>99.37</td>
</tr>
<tr>
<td>eco25/I1</td>
<td>0.0109</td>
<td>93.84</td>
<td>0.0066</td>
<td>97.97</td>
<td>0.0065</td>
<td>98.71</td>
</tr>
<tr>
<td>eco25/I2</td>
<td>0.0232</td>
<td>95.48</td>
<td>0.0133</td>
<td>98.60</td>
<td>0.0089</td>
<td>99.38</td>
</tr>
<tr>
<td>eco25</td>
<td>0.0364</td>
<td>96.35</td>
<td>0.0213</td>
<td>98.67</td>
<td>0.0140</td>
<td>99.42</td>
</tr>
<tr>
<td>eco26</td>
<td>0.0265</td>
<td>95.88</td>
<td>0.0158</td>
<td>98.50</td>
<td>0.0111</td>
<td>99.28</td>
</tr>
<tr>
<td>eco27</td>
<td>0.0227</td>
<td>96.27</td>
<td>0.0147</td>
<td>98.39</td>
<td>0.0142</td>
<td>99.05</td>
</tr>
<tr>
<td>eco28</td>
<td>0.0262</td>
<td>95.76</td>
<td>0.0153</td>
<td>98.51</td>
<td>0.0154</td>
<td>99.03</td>
</tr>
<tr>
<td>eco29/I1</td>
<td>0.0389</td>
<td>98.18</td>
<td>0.0189</td>
<td>99.57</td>
<td>0.0212</td>
<td>99.13</td>
</tr>
<tr>
<td>eco29/I2</td>
<td>0.0402</td>
<td>98.05</td>
<td>0.0184</td>
<td>99.55</td>
<td>0.0235</td>
<td>99.24</td>
</tr>
<tr>
<td>eco29</td>
<td>0.1037</td>
<td>97.77</td>
<td>0.0514</td>
<td>99.42</td>
<td>0.0793</td>
<td>99.01</td>
</tr>
<tr>
<td>eco30/I1</td>
<td>0.0161</td>
<td>96.38</td>
<td>0.0105</td>
<td>98.34</td>
<td>0.0079</td>
<td>98.85</td>
</tr>
<tr>
<td>eco30/I2</td>
<td>0.0120</td>
<td>94.46</td>
<td>0.0108</td>
<td>95.60</td>
<td>0.0086</td>
<td>97.30</td>
</tr>
<tr>
<td>eco30</td>
<td>0.0267</td>
<td>96.35</td>
<td>0.0181</td>
<td>98.20</td>
<td>0.0149</td>
<td>99.13</td>
</tr>
<tr>
<td>eco31</td>
<td>0.0430</td>
<td>98.57</td>
<td>0.0176</td>
<td>99.71</td>
<td>0.0295</td>
<td>99.16</td>
</tr>
<tr>
<td>jmp</td>
<td>0.0561</td>
<td>98.02</td>
<td>0.0244</td>
<td>99.56</td>
<td>0.0319</td>
<td>99.27</td>
</tr>
<tr>
<td>jsr</td>
<td>0.0776</td>
<td>98.30</td>
<td>0.0353</td>
<td>99.56</td>
<td>0.0551</td>
<td>99.13</td>
</tr>
<tr>
<td>mean</td>
<td><strong>0.0361</strong></td>
<td><strong>96.61</strong></td>
<td><strong>0.0192</strong></td>
<td><strong>98.60</strong></td>
<td><strong>0.0219</strong></td>
<td><strong>99.03</strong></td>
</tr>
</tbody>
</table>

Table 5.2. Gamma approximating functions. Errors of simulation.
Figure 5.8. Current simulation at program level at 100MHz, 200MHz, and 300MHz.

<table>
<thead>
<tr>
<th>Program/Trace</th>
<th>100 MHz</th>
<th>200 MHz</th>
<th>300 MHz</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Max [mA]</td>
<td>Mean [mA]</td>
<td>Max [mA]</td>
</tr>
<tr>
<td>powerco.asm/measured</td>
<td>83.2190</td>
<td>60.4232</td>
<td>137.7500</td>
</tr>
<tr>
<td>powerco.asm/simulated</td>
<td>81.4811</td>
<td>59.7857</td>
<td>137.4798</td>
</tr>
<tr>
<td>powercoN.asm/measured</td>
<td>76.9270</td>
<td>57.3267</td>
<td>127.1240</td>
</tr>
<tr>
<td>powercoN.asm/simulated</td>
<td>76.9063</td>
<td>56.6446</td>
<td>128.0719</td>
</tr>
</tbody>
</table>

Table 5.3. Max, Mean values at 3 different frequencies. Simulated and measured waveforms.
Table 5.4. Relative errors for Max, Mean values (simulated versus measured) plus the standard variation values of the difference waveform (IR-IS).

Figure 5.9. Regression line for current ratio to frequency ratio dependency.

Figure 5.8 shows a linear type dependency between the current traces of a program and the clock frequencies (See Equation 2.1 of Chapter 2 and Figure 4.3 of Chapter 4). For these frequency plots, an experimental regression line equation [66] was determined. In general, for a given set of points \( \{X, Y\} \) where \( X \) was a frequency ratio vector and \( Y \) was a current ratio vector, the regression line equation could be defined by the following relations [66]:

<table>
<thead>
<tr>
<th>Program</th>
<th>100 MHz</th>
<th>200 MHz</th>
<th>300 MHz</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Error of Max</td>
<td>Error of Mean</td>
<td>s [mA]</td>
</tr>
<tr>
<td>powerco</td>
<td>2.08%</td>
<td>1.06%</td>
<td>1.3309</td>
</tr>
<tr>
<td>powercoN</td>
<td>0.02%</td>
<td>1.19%</td>
<td>1.0908</td>
</tr>
</tbody>
</table>
\[ \hat{y} = a + bx; \quad a = \frac{(\sum y)(\sum x^2) - (\sum x)(\sum xy)}{m(\sum x^2) - (\sum x)^2}; \quad b = \frac{m(\sum xy) - (\sum x)(\sum y)}{m(\sum x^2) - (\sum x)^2}; \quad (5.8) \]

where, \((x, y)\) was an individual pair point from the set points \(\{X, Y\}\) and \(n\) was the total number of points. The Equation (5.8) could be used to estimate current plots of a program at various frequencies after a current simulation was performed at a working frequency. In order to determine the parameters of the regression line equation, 10 sample points per program were considered. A sample point k at different frequencies (100MHz, 200MHz, 300MHz) was taken at times \(t_{100}^{(k)}, t_{200}^{(k)}\) and \(t_{300}^{(k)}\) (in seconds) such that the number of clock cycles from the start points \(t_{s100}, t_{s200}, \text{ and } t_{s300}\) of the program until the representative frequency times were equal:

\[ (t_{100}^{(k)} - t_{s100}) \cdot 100MHz = (t_{200}^{(k)} - t_{s200}) \cdot 200MHz = (t_{300}^{(k)} - t_{s300}) \cdot 300MHz \quad (5.9) \]

For example, the current ratio \(i(t_{200}^{(k)}) / i(t_{100}^{(k)})\) (See y axis of Figure 5.9) could be plotted at the x axis of 2, since \((f_2 = 200MHz) / (f_1 = 100MHz) = 2\). The current ratio to frequency ratio dependency was program dependent. Nevertheless, the regression line equation could be used to obtain a good approximation for the dependency multiplication factor. From the sample points analyzed for the powerco.asm and powercoN.asm, the following regression line was obtained:

\[ \hat{y} = 0.3934 + 0.6383 \cdot x \quad (5.10) \]

A plot of the regression line and its determining points is presented in Figure 5.9. The regression line’s Equation (5.10) could be used to approximate the processor’s current consumption (SC140) at different frequencies than the analyzed frequency in the following way:

- Suppose that a current trace was obtained by means of current measurements or simulation at a working frequency \(f_1\). If the current trace was represented by n sample points \(CT_{f1} = \{(t_0, i_0), (t_1, i_1), \ldots, (t_{n-1}, i_{n-1})\}\), then the new current trace at a different working frequency \(f_2\) would be:

\[ CT_{f2} = \{(t_0 \cdot \hat{f}_2 / f_2, \hat{y}(f_2 / f_1) \cdot i_0), (t_1 \cdot \hat{f}_2 / f_2, \hat{y}(f_2 / f_1) \cdot i_1), \ldots, (t_{n-1} \cdot \hat{f}_2 / f_2, \hat{y}(f_2 / f_1) \cdot i_{n-1})\} \]
5.3 Simulation Results at the Program Level and Building the Model

In this section, simulated current waveforms obtained by applying the superposition macro-modeling approach introduced in Chapter 4 are presented. Results for the current dynamics model are presented when gamma functions are used as atomic functions. The modeling approach that used gamma functions as atomic functions was also named a gamma model.

All of the instructions of the SC140 instruction set, with few exceptions, executed in one clock cycle. Nevertheless, due to memory access restrictions within the core’s memory and parallel grouping, there were execution sets within the programs that executed in 2 clock cycles instead of 1 clock cycle (See Section 3.1). The real execution time per execution set was used in all the simulations. The current simulation model calculated two simulated current waveforms: the base current and the final current.

Experimental results showed that the measured waveforms had a complex circuit state dependency which had to be modeled. The circuit state impact on the simulated current was dependent mainly on the parallel activity (See Section 4.4) and on the data handled by the program. In order to determine a general formula that could be used to approximate the circuit state dependency for the SC140 DSP processor, the following model parameters per linear block \(k\) \((LB_k)\) were considered:

1. \(pb_1^{(k)}\): Average number of instructions per clock cycle.
2. \(pb_2^{(k)}\): Average number of instructions per execution set.
3. \(pb_3^{(k)}\): Average of parallel variation as an average number per execution set of the differences between the numbers of instructions per consecutive execution sets.

An example of calculating these parameters is provided in Figure 5.10. In this figure, a small linear block \(LB_k\) of 5 execution sets (ES) and its execution scenario were presented. The execution sets executed sequentially and the number of instructions per ES in their sequential order were: 4, 3, 5, 2, 2. The execution time in clock cycles for these ES was: 2, 1, 1, 1, 1. The instruction distribution, the execution times in clock cycles per ES, and the numerical calculations are all shown in this figure.

96
Figure 5.10. Example of model parameter calculation for a short linear bloc.

All of the analysis presented in this section was performed at a working frequency of 200MHz. In this analysis, four independent assembly language programs were used: powerco.asm, bubble.asm, viterbi.asm, polymul.asm. The analysis also included two extra programs powercoN.asm and bubvit.asm, which were derived from the powerco.asm, bubble.asm, and viterbi.asm. The powerco.asm and powercoN.asm programs were the same program versions used in Sections 4.2.1 and 5.2. The bubvit.asm was a combination of the viterbi.asm, bubble.asm and NOP blocks. The powerco.asm and polymul.asm were cryptographic applications. The polymul.asm was a very long and complex application with nested loops, control instructions and all types of moves and memory addressing. The viterbi.asm was a DSP specific application with high level of parallelism and special DSP oriented instructions. The bubble.asm was a general type application with nested hardware loops and control instructions. The complete listing for all these programs is presented in Appendix B. All of the maximum values used in the current simulation of powerco.asm, bubble.asm and viterbi.asm applications are presented in the Table B.1 of Appendix B.
The model parameters were calculated for all linear blocks that were formed within these programs. For the analysis presented in this section, the linear blocks were formed based on the parallelism distribution, hardware loop boundaries, unconditional block repetitions, and execution time of the block. An efficient length for a linear block execution was determined to be at least 160 clock cycles (800ns at 200MHz) in order to allow for a complete capacitance charge to take place. Large nested hardware loops or unconditional blocks that had small parallelism variation across the encompassed code were considered a linear block. Any continuous code that was long enough and had a high level of parallelism was considered as a separate linear block. The linear blocks used in this analysis were denoted by Bi. The block separation for all the Bi blocks was marked in the listing of the programs presented in Appendix B by two comment lines ‘; BEGIN Bi’ and ‘; END Bi’.

<table>
<thead>
<tr>
<th>Linear Block (B_k): program</th>
<th>pb_1^{(k)}</th>
<th>pb_2^{(k)}</th>
<th>pb_3^{(k)}</th>
<th>Representative Point of the Linear Block: {time, I_{sim}^{(k)}, I_{real}^{(k)}}</th>
<th>Clock Cycles per Program</th>
</tr>
</thead>
<tbody>
<tr>
<td>B1: bubble.asm</td>
<td>0.6304</td>
<td>1.6240</td>
<td>0.6640</td>
<td>{1.615, 39.9565, 58.4530}</td>
<td>750</td>
</tr>
<tr>
<td>B3: viterbi.asm</td>
<td>3.1566</td>
<td>5.6139</td>
<td>0.4304</td>
<td>{1.410, 177.5900, 129.4710}</td>
<td>282</td>
</tr>
<tr>
<td>B4: powerco.asm</td>
<td>1.1138</td>
<td>1.6174</td>
<td>0.8609</td>
<td>{0.860, 65.6635, 65.1910}</td>
<td>167</td>
</tr>
<tr>
<td>B5: powercoN.asm</td>
<td>1.0923</td>
<td>1.5778</td>
<td>0.8667</td>
<td>{1.410, 52.9954, 54.3250}</td>
<td>207</td>
</tr>
<tr>
<td>B6: bubvit.asm</td>
<td>3.1566</td>
<td>5.6139</td>
<td>0.4304</td>
<td>{1.410, 177.5900, 129.4710}</td>
<td>1044</td>
</tr>
<tr>
<td>B7: bubvit.asm</td>
<td>0.6090</td>
<td>1.5000</td>
<td>0.6544</td>
<td>{3.085, 41.6728, 62.0320}</td>
<td></td>
</tr>
<tr>
<td>B8: bubvit.asm</td>
<td>1.1854</td>
<td>1.4897</td>
<td>1.0501</td>
<td>{5.115, 53.2017, 66.2340}</td>
<td></td>
</tr>
<tr>
<td>B9: polymul..asm</td>
<td>1.3363</td>
<td>1.7948</td>
<td>0.9064</td>
<td>{6.490, 86.1469, 78.6140}</td>
<td>4264</td>
</tr>
<tr>
<td>B10: polymul.asm</td>
<td>1.1318</td>
<td>1.5195</td>
<td>1.0108</td>
<td>{12.305, 67.1178, 79.0466}</td>
<td></td>
</tr>
<tr>
<td>B11: polymul.asm</td>
<td>2.8333</td>
<td>3.8163</td>
<td>1.1837</td>
<td>{21.335, 96.3413, 93.7090}</td>
<td></td>
</tr>
</tbody>
</table>

Table 5.5. Model parameters for circuit state dependency.

Table 5.5 presents the total execution times per programs in clock cycles, the model parameters \(pb_1^{(k)}, pb_2^{(k)}\) and \(pb_3^{(k)}\) calculated for all the linear blocks formed within the analyzed programs, and the representative point of the block. These linear blocks included a large variety of instruction
types and parallelism composition. This variety made it possible to determine and verify a general expression for the circuit state dependency on parallelism of the SC140 DSP processor.

The modeling goal was to find a parameter dependency such that the lambda ($\Lambda$) block factors could account for the circuit state dependency (See Sections 4.3 and 4.4). The circuit state dependency of the model was derived from experimental measurements and simulations performed on small basic blocks and it was validated on different programs of various lengths.

Based on the maximum values obtained for individual types of instructions that were part of a program, a current base simulation trace $I_{S_b}$ was formed. This current base trace used the gamma function constants determined in Section 5.2 and a value of 1 for all of the lambda block factors. The final current simulation trace $I_{S_f}$ was formed by fitting the current base simulation $I_{S_b}$ to the measured waveform $I_R$. The curve fitting had to be performed on all the blocks that were part of the simulated program. The solution to the circuit state dependency on the model parameters could be found by solving linear equations [14] with numerical values derived from current base traces and measured traces. The equation to be solved for each linear block $B_k$ by this modeling approach is:

\[
(\sum_{i=1}^{3} p b_i^{(k)} \cdot x_i) \cdot I_{\text{sim}}^{(k)} = I_{\text{real}}^{(k)}
\]

\[
(\sum_{i=1}^{3} p b_i^{(k)} \cdot x_i) = I_{\text{real}}^{(k)} / I_{\text{sim}}^{(k)}
\]

where $I_{\text{real}}^{(k)}$ is the measured current value for a representative point of the block $B_k$, $I_{\text{sim}}^{(k)}$ is the simulated base current value for the same point in time, and $p b_i^{(k)}$ is the parameter $i$ ($i=1,2,3$) of the $B_k$ block. The points that represented each block were called representative points and they were the maximum values of the portion of the dynamic curve, where the block $B_k$ was executed. For each block, the multiplication factor that accounted for the circuit state dependency was:

\[
\Lambda^{(B_k)} = (\sum_{i=1}^{3} p b_i^{(k)} \cdot x_i)
\]
Having three dependency parameters per linear block, \( ph_1^{(k)} \), \( ph_2^{(k)} \), and \( ph_3^{(k)} \), a general procedure for determining the modeling solution for the SC140 DSP processor was to consider at least three representative linear blocks (short blocks) and solve the matrix Equation (5.12) for these combination of blocks. The solution obtained provided the \( x_i \) parameters that could be used for the calculation of all other lambda blocks \( \Lambda^{B_k} \) that were not part of the solution matrix.

Figure 5.11 presents a block diagram to the general solution used to determine the circuit state dependency of the SC140 processor. The first three linear blocks were part of two assembly language programs called bubble.asm and viterbi.asm. These two programs were called the ‘model input programs’ since they were used to obtain the modeling solution. The linear block \( B_1 \) was characterized by a low number of instructions per clock cycle due to an unconditional loop created by the use of the assembly language instruction ‘BT label’. The linear block \( B_2 \) was the second block of the bubble.asm program. This block was a typical type block with a medium number of instructions per execution set, and clock cycle that could be encountered in any general type program. The block had two nested hardware loops that contained memory access instructions and conditional type instructions (IFT and IFA). The linear block \( B_3 \) was characterized by a large number of instructions per execution set and clock cycle, combined with a low parallelism.
variation. Combining the data points for these three representative blocks, the general solution of the circuit state dependency was obtained by forming the following matrixes:

\[
P_B = \begin{bmatrix} p_{b_1}^{(1)}, p_{b_2}^{(1)}, p_{b_3}^{(1)} \\ p_{b_1}^{(2)}, p_{b_2}^{(2)}, p_{b_3}^{(2)} \\ p_{b_1}^{(3)}, p_{b_2}^{(3)}, p_{b_3}^{(3)} \end{bmatrix} \quad ; \quad X' = [x_1, x_2, x_3] \quad ; \quad IR' = \begin{bmatrix} I_{\text{real}}^{(1)} \\ I_{\text{real}}^{(2)} \\ I_{\text{real}}^{(3)} \\ I_{\text{sim}}^{(1)} \\ I_{\text{sim}}^{(2)} \\ I_{\text{sim}}^{(3)} \end{bmatrix}
\] (5.14)

The general model solution was given by solving the following matrix equation:

\[
P_B \times X = IR
\] (5.15)

The numerical solution (using MATLAB and data presented in Table 5.5) was:

\[
X' = [-1.5004, 0.8558, 1.5343]
\] (5.16)

The numerical solution X was used to calculate all other lambda block values \( \Lambda^{(B_k)} \) after the block parameters \( p_{b_i}^{(k)}; i = 1, 2, 3 \) were known. The numerical values of the vector solution X indicated that the current base trace overestimated the current for programs with high number of clock cycles per execution set and underestimated the current for programs with high level of parallelism that have a number of clock cycles per execution set less than 1.5.

The individual solution for each linear block gave the best lambda block value, which was called the ideal lambda. The lambda block values calculated based on the Solution (5.16) gave the calculated lambdas. The ideal and calculated lambda block values for the three representative blocks \( B_1, B_2, B_3 \) were identical and were equal with: \( \Lambda^{(B_1)}_{\text{bubble}} = 0.9736 \), \( \Lambda^{(B_2)}_{\text{bubble}} = 0.9736 \), \( \Lambda^{(B_3)}_{\text{viterbi}} = 0.9736 \). Using the matrix Solution (5.16) and the \( p_{b_i} \) values presented in Table 5.6, the following calculated lambda block values were obtained: \( \Lambda^{(B_4)}_{\text{powerco}} = 1.0342 \), \( \Lambda^{(B_5)}_{\text{powercoN}} = 1.0414 \), \( \Lambda^{(B_6)}_{\text{bubvit}} = 0.7290 \), \( \Lambda^{(B_5)}_{\text{bubvit}} = 1.3743 \), \( \Lambda^{(B_6)}_{\text{bubvit}} = 1.1078 \), \( \Lambda^{(B_7)}_{\text{polymul}} = 0.9220 \), \( \Lambda^{(B_8)}_{\text{polymul}} = 1.1534 \), \( \Lambda^{(B_9)}_{\text{polyco}} = 0.8315 \). These lambda block values accounted for the circuit state dependency on parallelism of the SC140 DSP processor as a function of the \( p_{b_i}^{(k)} \) parameters.
The following figures present various current plots for the programs analyzed. The blue waveforms are the measured currents and the red waveforms are the simulated currents. The current simulation methodology presented in Chapter 4 was used to derive all these plots. The simulation used gamma functions with the gamma constants at 200MHz that are presented in Section 5.2, together with the maximum values of Table B.1 of Appendix B.

Figure 5.12. Current base waveforms versus measured waveforms of model input programs.

Figure 5.13. Final current simulation waveform versus measured waveform of model input programs.
Figure 5.12 presents the simulated base waveforms (lambda block values were 1) versus the measured current waveforms, and Figure 5.13 presents the final simulated current (calculated lambda block values) versus the measured current waveforms for the input programs to the model. The base current simulation was underestimated for the bubble.asm program and overestimated for the viterbi.asm program. The block division was visible in both of these plots. The block B2 of bubble.asm was a complex block with nested loops, conditional instructions and memory moves. The internal loop had a variable length, and execution sets with IFT conditional instructions were present in the loop. These, together with the relative close positioning of the LPMARKB instructions [47] for the nested loops, made it difficult to obtain a current trace with a close related shape to the measured trace. A better simulated shape could be obtained by splitting the basic block B2 in two smaller blocks [21] but this was found to complicate the circuit state dependency solution. The viterbi.asm program did not have any conditional instructions with the exception of a main hardware loop that encompassed the entire code. As a result, the current simulation was very close to the measured waveform.

Figures 5.14 and 5.15 present the simulated base current, and the final simulated current versus the measured current for the test programs powerco.asm, powercoN.asm, bubvit.asm, polymul.asm. The powerco.asm, powercoN.asm, and polymul.asm were totally independent of the bubble.asm and viterbi.asm programs. The only program that was dependent on the model input programs was bubvit.asm. This program was introduced to indicate the capability of the model to reproduce the block succession dependency. The poweco.asm and powercoN.asm were short programs and the current simulation worked very well in these cases. In the case of bubvit.asm, the simulated current was underestimated for the B7 and B8 blocks in both situations (base current simulation and final current simulation). The polymul.asm program was a very long and complex program. Its code was divided in three linear blocks: B9, B10, and B11. The B9 block included an initialization block plus an unconditional loop that included short hardware loops and a variable length hardware loop. The unconditional loop executed three times and the hardware loop within this unconditional loop executed one, three, and seven times, respectively. The B10 block had an unconditional loop that included all kinds of hardware loops and a large variety of instruction types. The main unconditional loop repeated for seven times and it had equal execution times. The last block, B11, was a highly parallel block without any hardware loops or conditional instructions.
The difficulty of simulating this program was mainly due to its length, the complex data dependency, various types of memory addressing, nested loops and the short hardware loops that used the instruction LPMARKA. Nevertheless, as seen from the figures, the current simulation shape and magnitude proved to be satisfactory as related to the measured waveform.

Figure 5.14. Current base waveforms versus measured waveforms of the test programs.
Table 5.6 presents a summary of current measurements performed by using the simulated and measured waveforms of Figures 5.13 and 5.15. The results show that the simulated waveforms could be used to measure mean, maximum, energy and other current measurements at the
program level and the modeling technique introduced in this thesis could be useful for any kind of power or energy optimization analysis that targets the software composition.

Table 5.7 summarizes the relative errors, the standard deviation of the difference waveform (measured-simulated), the RMS errors and the $CC^2$ [%] coefficient of the simulated waveforms versus the measured waveforms. The combined relative errors between the measurements of mean, maximum, and energy performed on the simulated waveforms versus the measured waveforms were under 1.4%. Overall, the input programs proved to show smaller combined errors than the test programs and that was due to the fact that their calculated lambdas were identical with the ideal lambdas. The RMS error (2.17mA) of the combined input programs (row 8 of the Table 5.7) was less than 1.06% of the maximum value (205.73mA) and the $CC^2$ [%] for the combination was over 99%. The RMS error (5.89mA) of the combined test programs (row 9 of Table 5.7) was less than 2.87% of the maximum value (205.84mA) and the $CC^2$ [%] for the combination was over 96%. As a result, the overall final simulated current had a very close magnitude and shape to the measured current.

<table>
<thead>
<tr>
<th>PROGRAM</th>
<th>ISmean [mA]</th>
<th>IRmean [mA]</th>
<th>ISmax [mA]</th>
<th>IRmax [mA]</th>
<th>ISenergy [$\mu J$]</th>
<th>IRenergy [$\mu J$]</th>
</tr>
</thead>
<tbody>
<tr>
<td>bubble.asm</td>
<td>115.9789</td>
<td>115.3227</td>
<td>136.6977</td>
<td>137.2767</td>
<td>1.1799</td>
<td>1.1732</td>
</tr>
<tr>
<td>viterbi.asm</td>
<td>131.2085</td>
<td>132.8342</td>
<td>203.4834</td>
<td>205.7348</td>
<td>0.7633</td>
<td>0.7727</td>
</tr>
<tr>
<td>powerco.asm</td>
<td>100.0556</td>
<td>100.8914</td>
<td>139.1908</td>
<td>140.5288</td>
<td>0.4681</td>
<td>0.4720</td>
</tr>
<tr>
<td>powercoN.asm</td>
<td>98.9478</td>
<td>98.9118</td>
<td>128.3318</td>
<td>129.9058</td>
<td>0.4996</td>
<td>0.4994</td>
</tr>
<tr>
<td>bubvit.asm</td>
<td>135.0605</td>
<td>134.1919</td>
<td>203.4834</td>
<td>205.8381</td>
<td>1.7457</td>
<td>1.7345</td>
</tr>
<tr>
<td>polymul.asm</td>
<td>139.4821</td>
<td>136.9806</td>
<td>168.5484</td>
<td>168.6994</td>
<td>6.6978</td>
<td>6.5777</td>
</tr>
<tr>
<td>bubble + viterbi</td>
<td>121.5194</td>
<td>121.6933</td>
<td>203.4834</td>
<td>205.7348</td>
<td>1.9432</td>
<td>1.9460</td>
</tr>
<tr>
<td>powerco + powercoN</td>
<td>133.1679</td>
<td>131.3619</td>
<td>203.4834</td>
<td>205.8381</td>
<td>9.4112</td>
<td>9.2836</td>
</tr>
<tr>
<td>bubvit + polymul</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Table 5.6. Measurements of current mean, maximum, and derived energy from the simulated and measured waveforms.
<table>
<thead>
<tr>
<th>PROGRAM</th>
<th>Relative Error [% of Mean Values]</th>
<th>Relative Error [% of Max Values]</th>
<th>Relative Error [% of Energy Values]</th>
<th>RMSerr [mA]</th>
<th>CC² [%]</th>
</tr>
</thead>
<tbody>
<tr>
<td>bubble.asm</td>
<td>-0.5663</td>
<td>0.4218</td>
<td>-0.5663</td>
<td>1.4284</td>
<td>1.5713</td>
</tr>
<tr>
<td>viterbi.asm</td>
<td>1.2370</td>
<td>1.0943</td>
<td>1.2370</td>
<td>2.4469</td>
<td>2.9361</td>
</tr>
<tr>
<td>powerco.asm</td>
<td>0.8347</td>
<td>0.9521</td>
<td>0.8347</td>
<td>1.7399</td>
<td>1.3243</td>
</tr>
<tr>
<td>powercoN.asm</td>
<td>-0.0230</td>
<td>1.2116</td>
<td>-0.0230</td>
<td>1.3936</td>
<td>1.1689</td>
</tr>
<tr>
<td>bubvit.asm</td>
<td>-0.6578</td>
<td>1.1440</td>
<td>-0.6578</td>
<td>1.9655</td>
<td>2.1749</td>
</tr>
<tr>
<td>polymul.asm</td>
<td>-1.8258</td>
<td>0.0895</td>
<td>-1.8258</td>
<td>7.1280</td>
<td>7.0298</td>
</tr>
<tr>
<td>bubble + viterbi</td>
<td>0.1498</td>
<td>1.0943</td>
<td>0.1498</td>
<td>2.1632</td>
<td>2.1697</td>
</tr>
<tr>
<td>powerco + powercoN</td>
<td>-1.3753</td>
<td>1.1440</td>
<td>-1.3753</td>
<td>6.1646</td>
<td>5.8870</td>
</tr>
<tr>
<td>bubvit + polymul</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Table 5.7. Error values, simulated waveforms versus measured (filtered) waveforms.

Figure 5.16 presents the simulated waveforms obtained by using the ideal lambda block values versus the averaged waveforms. The plots also present the EXE signal that shows the effective execution time of the program. As it can be seen, there was a current delay response, for all the programs, due to the charging time of the processor’s capacitance. The averaged waveforms showed a higher level of noise in comparison with the filtered waveforms. Nevertheless, the final simulated waveforms followed closely the magnitude and the slope of the measured current. The comparison measurement results between the simulated and measured waveforms of Figure 5.16 showed similar level of errors with the errors presented in Table 5.5.

Figures 5.17 and 5.18 present enlarged plots of the waveform analysis for the bubvit.asm program and the polymul.asm program, respectively. These plots present the measured (averaged) current waveform obtained by using the DC current probe in blue, the final simulated current waveform obtained by using the calculated lambda block values in red, the EXE signal measured by the measuring station in black, and the difference waveform that was obtained by subtracting the simulated current from the measured current plus a shift value in green.
Figure 5.16. Final current simulation waveform with ideal lambda block values versus measured averaged waveforms.
Figure 5.17. Simulation waveform based on calculated lambda blocks versus averaged waveform for bubvit.asm.

The bubvit.asm program presented in Figure 5.17 was a combination program that included high parallelism blocks and conditional blocks. The standard deviation of the difference waveform was 2.146 mA, which was less than 1.60% of the measured current’s mean (134.18) and the range of the difference waveform was under 12 mA, which was less than 8.77% of the mean. This program was a short program and the amount of control instructions used was reduced. Also, the data variation within the program was not that significant.

The polymul.asm program presented in Figure 5.18 was in fact a subroutine of a larger program called polymulNIST.asm that implemented the elliptic point multiplication algorithm (See Section 2.31). The execution time of the polymul.asm program was 4264 clock cycles. In this case, the standard deviation of the difference waveform was 26.08 mA and the range of the difference waveform was 87.02. These large values for the standard variation and the range were due to poor simulation results for the first 10% of the current waveform. When this part of the waveform was
eliminated, the standard variation of the difference waveform was 2.790mA, which was less than 2.04% of the measured current's mean (136.98mA) and the range of the difference waveform was 17.04mA, which was less than 12.44% of the mean.

5.4 Comparisons with Static Instruction Level Model

This section compares a static model of the SC140 DSP processor utilizing the energy model introduced by Tiwari et al. [6] with the proposed dynamic model introduced in this thesis. For the rest of this section, the dynamic model introduced in the thesis is called a gamma model, and the energy model introduced by Tiwari et al. [6] is called a static model.
Due to its parallel capability, the SC140 DSP processor could execute up to 6 instructions in parallel. The static model did not accommodate processors with parallel capabilities. As a result, the static model had to be modified in order to be applicable to the SC140 DSP processor. Since the processor’s instruction set was large, the possible number of execution sets of instructions that could be combined in parallel was proportional with the sum of all possible combinations of 1, 2, 3, 4, 5 and 6 instructions per execution set. The data collection for all possible combinations was impractical. Due to this fact and meter measurement difficulties, a modified static model was introduced. According to this modified model, the data collection was needed only for individual instructions, based on instruction type and addressing modes, which were part of the execution set. The average current consumption per execution set was derived by applying the parallel superposition principle for static analysis. This principle was derived from the parallel superposition principle introduced in Section 4.2.4 of Chapter 4. Based on this principle, the average current consumption per execution set minus the NOP-level was considered to be equal with the sum of all average currents per individual instructions that composed the set, minus their static NOP-level. With the aid of this principle, the average current consumption per execution set was calculated based on the current averages of the composing instructions. For example, if an execution set ES had the following composition: \( ES = [I_1, I_2, I_3, I_4, I_5, I_6] \), the average current consumption for the execution set ES could be calculated by the following current relation:

\[
I(ES) = [I(I_1) + I(I_2) + I(I_3) + I(I_4) + I(I_5) + I(I_6) - 6 \times I(NOP)] + I(NOP)
\]

(5.17)

where \( I(In) \) (\( n \in \{1,2,3,4,5,6\} \)) was the average current measurement per instruction obtained by using the meter and the looping measuring method [6], and \( I(NOP) \) was 66.18mA.

The following presents some key comparison aspects between the dynamic gamma model and the static model:

- The gamma model and the static model are both instruction based models. However, the static model estimates average DC current consumptions per programs while the gamma model estimates instantaneous current consumptions per programs.
• The applications of the static model are limited to applications that use average current estimates while the dynamic model can be used in applications that use both, average and instantaneous current estimates.

• The gamma model uses maximum current values that require the use of an oscilloscope while the static model uses average current measurements obtained by the use of an ammeter.

• The gamma model is capable of conveying dynamic and static information related to the current, power, and energy consumption of a processor that are related to the software application. The static model applies simple average measurements at the instruction level by the use of a digital ammeter to construct a more complex model capable of providing average power and average energy estimates at the software level.

• For programs that had a short execution time, the meter measurement could not be used since a visual current reading could not be obtained. To overcome this difficulty, the static method required that the instructions and the programs must be executed in an infinite loop [6]. This method is called in this thesis a looping method. In this thesis, an interrupt-based measurement technique eliminated this difficulty and current measurements could be taken for instructions and programs of any execution length. In fact, the measurement method used in this thesis could perform single shot measurements that captured current consumptions for one-time running programs.

• The current measurement at the program level introduced in [6] was limited to programs with execution times much smaller than 100ms. This limitation was due to the limitations of the meter used. Similarly, the EXE method and the Wait method introduced in this thesis were limited to programs under 50ms length due to the limitations of the scope. Nevertheless, by using the EXE method the current measurements for programs of practical length could be performed by dividing the measurements in additional portions of less than 50ms each.

Significant measurement differences have been observed between the meter looping method and the WAIT or EXE methods used in this thesis. For example, the NOP-level current at 200MHz
when measured with the oscilloscope was within the [72mA-74mA] interval and when measured with the meter was within the [65mA-67mA] interval. Greater measurement differences between the scope and the meter were encountered for various instructions and programs. In general, it was noted that most of the instruction measurements performed with the meter were greater than the oscilloscope’s measurements. These differences were mainly due to the looping method used and the limited frequency bandwidth response of the meter versus the high frequency bandwidth response of the oscilloscope and the AC current probe. Due to the cumulative capacitance of the processor, the looping measurement method had the tendency to give higher average readings in comparison with the oscilloscope’s measurements, since the charge generated by the application did not have enough time to discharge to its idle value (start value) before the application started to execute again. The situation was worse for individual instruction and short programs when the loop repeated at short intervals of time. It was also noted that the current trace of a looped program had the tendency to follow the peak current values. As a result, the one time execution current measurement (Wait method) was considered to give better results of current measurements for applications that are followed by idle states.

The static model assigned a fixed energy cost value called base energy cost for each instruction. The base energy cost of a program was the sum of the base energy costs of all executed instructions. Gamma model used mathematical functions to approximate the variable component of the current at the instruction level. These gamma functions were built for each instruction based on the maximum current values per instruction. The basic current waveform per program was obtained by applying superposition principles to all of the instructions executed by the program. The base energy costs per program of the static model did not account for the inter-instruction effects due to circuit state and resource constraints. These effects were modeled separately and added to the base energy costs to generate the total energy cost. The gamma model used factorial adjusting parameters to alter the base model and generate the final simulated current. The factorial parameters accounted for the circuit state dependencies on parallelism.

For the particular processor core used in this research a meter measurement difficulty was encountered and two other alternative measurement techniques were used. First, for individual instructions that had current consumptions greater than 120mA, the instructions were interleaved
with NOP instructions in the base program loop in order to reduce the average current value under 120mA. The current measurement in these cases was determined from comparison measurements with the other instructions. Secondly, for programs that had average current consumption greater than 120mA, the scope and the DC current probe were used rather than the meter. These alternative methods were needed since the meter method as introduced in [6] gave false readings for the SC140 processor every time the current measurement was above 120mA. The SC140 processor core includes 256Kx16-bit of on-chip SRAM memory [47]. Thus, the current measurement performed at the processor’s power supply pin included the current consumption of the CPU plus the chip’s memory. Due to the low power supply voltage (less than 2V), the voltage fluctuations introduced by the serial ammeter had negative impact on the measurements. For the SC140 processor, as soon as instructions or a group of instructions that consumed more than 120mA were measured, a false reading was obtained. The voltage drop caused by the current on the meter’s measuring resistance (1Ω) was high enough to produce voltage fluctuations and cause erratic execution and false measurements, by completely or partially erasing the content of the internal SRAM memory that held the instructions or program under measurement.

Figures 5.19, 5.20, 5.21, 5.22, and 5.23 present measurements and simulation analysis using the static method for the SC140 DSP processor. These figures present cycle-by-cycle simulated current values (top) and measured waveforms of current obtained by the Wait method and looping method (bottom). The average value for all of the instruction types used in the analyzed programs were the ones presented in the Table B.1 of Appendix B. The average current consumption per clock cycle is plotted in red in the time order of the execution sets, as the base current cost per program was calculated. The average current waveform per program measured by the oscilloscope is also plotted in blue along with the above base current cost development. Also, the EXE signal for each program analyzed is shown by a green waveform. In all of the examples presented, clearly the static model did not have the capability of capturing the real current variation of the processor. However, the figures were presented in order to illustrate the energy calculation differences. At the bottom of each figure, a plot of the average current measurement per program when measured with the wait method in blue and looping method in black is presented.
The bottom of Figure 5.19 compares the measured current waveforms for the powerco.asm program. In this case, the wait method of measurement produced a smaller current value than the looping method because the execution time of the application was short. The looping method gave more time for the program charge to accumulate. In Figure 5.20 again, the wait method of measurement produced a smaller current value than the looping method due to the short execution time of the program. Also in this case, the small NOP block (40 NOPs) introduced a continuous saw-type shape to the current waveform measured through the looping method. The currents of the repetitive loops followed the peak current generated by the last part of the program.
In Figure 5.21, the looping method of measurement again follows the peak current of the bubble.asm program giving higher current measurements than the wait method. Due to nested loops and parallelism variation within the loops (with execution sets that have high current consumptions (MOVEs) followed by execution sets with low current consumption), the simulation of the average current for the second part of the program was poor.

In Figure 5.22, the viterbi.asm waveforms show that the wait method and the looping method produced similar values of average currents per program. The average current simulation overestimated the measured average due to the high level of parallelism encountered in this program.
Figure 5.21. Static simulation versus measured current. Current waveforms measured by looping and wait methods for bubble.asm

In Figure 5.23, the bulvit.asm waveforms are a combination of bubble.asm, viterbi.asm and NOP blocks. The analysis of this program was performed to show the current variation when large basic blocks with high current consumption are followed by large blocks with smaller current consumption.

Table 5.8 summarizes the average current and the energy consumption measurements for all the programs analyzed in Figures 5.19, 5.20, 5.21, 5.22, and 5.23. The measurements covered only the execution time of the program measured by the EXE signal in order to coincide with the base current costs. The ‘Simulated Current’ column shows the simulated measurements per program while the ‘Wait Method’ and ‘Looping Method’ show the measured values. The table also presents three relative errors: simulated values related to the wait method values, simulated values related to the looping method values, and the wait method values related to the looping method values.
Figure 5.22. Static simulation versus measured current. Current waveforms measured by looping and wait methods for viterbi.asm.

The ‘Wait Method’ to the ‘Looping Method’ errors varied from about 6% to 45% with the observation that for larger programs the error was always smaller (under 7.2%). Due to the measurement differences between these two methods, the error between the simulated values and the wait method values were significant. Nevertheless, the range of the error was somewhat small (-18% to 18%) when the simulated values were compared with the looping method values. As indicated in [6], the base current cost must be adjusted in order to compensate for the circuit state dependency. It is shown in Section 5.3 that the current state dependency of the SC140 DSP processor has a variable component that is mainly due to the processor’s parallel capabilities. The error results presented in Table 5.8 also confirm that a constant adjusting value would not properly compensate the circuit state dependency. The relative errors showed negative and positive values.
A current value of 15mA (similar value with the value indicated in [6]) for the circuit state dependency on instruction sequencing and parallelism was found to substantially improve the simulated results for programs with a low level of parallelism (average number of instructions per execution sets less than 3). Nevertheless, programs with a high level of parallelism (average number of instructions per execution set greater than 3) must subtract a value of 15mA in order to improve the simulation. As a result, a constant current value adjustment for the circuit state dependency [6] on instruction sequencing and parallelism was not found to be supported by the static current simulation method for the SC140 processor.
<table>
<thead>
<tr>
<th>Measurements</th>
<th>Simulated Current</th>
<th>Wait Method</th>
<th>Looping Method</th>
<th>Error Simulated to Wait</th>
<th>Error Simulated to Looping</th>
<th>Error Looping to Wait</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Execution Time [ns]</strong></td>
<td>830</td>
<td>830</td>
<td>830</td>
<td>--</td>
<td>--</td>
<td>--</td>
</tr>
<tr>
<td><strong>Average Current [mA]</strong></td>
<td>137.837</td>
<td>108.247</td>
<td>158.021</td>
<td>27.33%</td>
<td>-12.77%</td>
<td>45.98%</td>
</tr>
<tr>
<td><strong>Energy Consumption [μJ]</strong></td>
<td>0.2185</td>
<td>0.1710</td>
<td>0.2489</td>
<td>27.78%</td>
<td>-12.21%</td>
<td>45.55%</td>
</tr>
</tbody>
</table>

**STATIC SIMULATION, MEASUREMENTS: powerconops.asm**

| Execution Time [ns] | 1030 | 1030 | 1030 | -- | -- | -- |
| Average Current [mA] | 125.501 | 107.937 | 140.837 | 16.27% | -9.47% | 30.48% |
| Energy Consumption [μJ] | 0.2465 | 0.2115 | 0.2753 | 16.55% | -10.46% | 30.31% |

**STATIC SIMULATION, MEASUREMENTS: bubble.asm**

| Execution Time [ns] | 3735 | 3735 | 3735 | -- | -- | -- |
| Average Current [mA] | 109.984 | 124.432 | 133.351 | -11.61% | -17.52% | 7.17% |
| Energy Consumption [μJ] | 0.7807 | 0.8827 | 0.9460 | -11.56% | -17.47% | 6.69% |

**STATIC SIMULATION, MEASUREMENTS: viterbi.asm**

| Execution Time [ns] | 1400 | 1400 | 1400 | -- | -- | -- |
| Average Current [mA] | 246.903 | 149.087 | 209.952 | 65.61% | 17.60% | 40.82% |
| Energy Consumption [μJ] | 0.6584 | 0.3969 | 0.5579 | 65.88% | 18.01% | 40.56% |

**STATIC SIMULATION, MEASUREMENTS: bubvit.asm**

| Execution Time [ns] | 5205 | 5205 | 5205 | -- | -- | -- |
| Average Current [mA] | 146.591 | 146.316 | 155.721 | 0.19% | -5.86% | 6.43% |
| Energy Consumption [μJ] | 1.4496 | 1.4462 | 1.5384 | 0.24% | -5.77% | 6.38% |

Table 5.8. Static simulation results and relative errors.
Chapter 6

APPLICATIONS TO SECURITY

Within this chapter, some immediate applications of the current measurement technique introduced in this thesis to security and reliability are presented. Low current consumption instructions were used in programs to reduce peak current values and smoothen the current trace of the program in order to increase the reliability and the security of a system. As indicated in [62], filtered current waveforms increased the number of samples needed for performing power analysis attacks on smartcards and offer good resistance to simple power analysis attacks. Smoother current waveforms have similar characteristics with filtered current waveforms. Current waveforms were collected for cryptographic applications and energy analysis was conducted to analyze the security of the implementations against power analysis attacks. It was found that the smoothened waveforms produced smaller values for the security indicator but not enough to defeat SPA. Finally, the current measurements at the instruction level were used to simulate the current consumption of the SC140 processor when additional security hardware was considered.

6.1 Current Waveform Smoothening Technique

This section introduces a methodology and its current dynamics measurement results used to analyze and eliminate peak current values within a current trace that were due to program constructs. In this section, the electrical impact of a program was simplified and it was assumed to behave like a simple RC type circuit. Each program was assumed to be capable of delivering a charge to the capacitor at a rate determined by the instruction types, the composition of the execution sets, and the parallelism variation within the program. The current consumption of a program was smoothened by inserting non-functional type instructions within the program. A non-functional type instruction was an instruction that did not alter the functionality of the program, nor consume additional registers. Examples of non-functional instructions (NFI) for the SC140 processor are NOP; OR dn,dn; and ADD #0,dn. The purpose of introducing such instructions within the code was to extend the execution time of blocks that produced high
currents and thus reduce the amount of charge per unit time delivered by these blocks. The insertion of NFIs were also capable of bringing the charging rates levels to the discharging rates levels of the processor during the block execution. The NFIs could be also grouped into an execution set in order to increase the current consumption of a portion of the program when the corresponding current values were too low. Based on the modeling technique introduced in Chapter 4 and Chapter 5, it was possible to determine a current consumption equivalence per execution set that was measured in number of ALU instructions. The SC140 ALU instructions had close maximum values and for this analysis, they were considered to be part of the same current consumption class. The instructions that used memory addressing, and the control instructions that executed a jump had significantly higher current values than the ALU type instructions.

The current value achieved by the program was controlled by the number of NFIs introduced within the program. These NFIs created a source free discharge path for the processor’s capacitor and did not allow the current to reach high peak values. The discharging time of the processor depended on the program that executed and on the initial peak current value to be controlled. This discharging time was assumed to be constant for a program and was called a program discharging time. For each program, the discharging time was calculated by performing two consecutive measurements of two versions of the program.

Next, a method of determining the number of NFIs that must be distributed within the program is presented. The NFI instruction used in the program examples presented in this section was NOP if otherwise specified. The voltage across the capacitor for a source free discharge cycle in a simple RC type circuit, like the one presented in Figure 6.1, can be calculated by using the following formula [64]:

\[ v_C(t) = V_0 e^{-t/\tau_d} \]

where \( \tau_d = (R_C + R_d) \cdot C \), and \( V_0 \) is the initial voltage value \( (6.1) \)

The value \( \tau_d \) is called the discharge time constant. Knowing the voltage function of the circuit, the current function of the circuit can be determined as follows:

\[ i(t) = -\frac{v_C}{R_C + R_d} = \frac{-V_0}{R_C + R_d} e^{t/\tau_d} = I_{d_0} \cdot e^{t/\tau_d} \]

\( (6.2) \)
Reducing the program execution on the SC140 processor to a simple RC circuit made it possible to approximate the total number of NFIs that must be introduced throughout a program in order to reduce a certain measurable peak current value to a lower current value. In the electrical circuit of Figure 6.1, the voltage supply $E$, was known but $I_0, C, R_C, R_d$ were practically impossible to determine. The discharging time $\tau_d$ of the processor executing a certain program was approximated by performing two measurements of the program’s current in two different situations:

1. Measurement 1: This measurement determined the peak current value $I_{\text{max}_1}$ and the program execution time $T_1$ of the original unaltered program.

2. Measurement 2: This measurement determined the peak current value $I_{\text{max}_2}$ and the execution time $T_2$ for an altered version of the program. The altered version of the program preserved the original functionality but used NFIs throughout the program such that the average number of ALU instructions per execution sets was made uniform to a fixed value $u_r$. This transformation process was called a uniformization process.
Table 6.1 ALU equivalence chart for calculating the time discharge constant per program.

Table 6.1 was used to perform the process of ALU uniformization for the altered version of a program used in Measurement 2. After the program uniformization process, for any two consecutive execution sets, the average number of ALUs was equal to a fixed uniformization value \( u_f \). Considering any execution set ES that had an ALU equivalence of \( k \) ALU units, the number \( n \) of NFIs that were introduced after this execution set was calculated based on the following relation:

\[
  n = u_f (k + 1)
\]  

(6.3)

Using the current Equation (6.2) and the experimental values \( I_{max1} \), \( T_1 \), \( I_{max2} \), and \( T_2 \), the value of the discharging time \( \tau_d \) was approximated based on the following relation:

\[
  \tau_d = \frac{T_2 - T_1}{\ln \frac{I_{max1}}{I_{max2}}}
\]  

(6.4)

If the current value \( I_{max2} \) obtained by applying the equivalence chart of Table 6.1 was unsatisfactory, further current adjustments were made by using the calculated discharging time value. Given a new current value \( I_{new} < I_{max1} \), the extra amount of time needed to extend the program execution in order to introduce a free discharge cycle of the processor’s capacitor to this new current value was calculated by the following equation:
The above method was applied with satisfactory results to smoothen the current consumption for two program applications. First, the results of this method were applied to the program application called bubvit.asm. All of the measured waveforms used in this section were averaged waveforms.

Figure 6.2 presents the current waveforms and the measurements obtained for the program version called Case 0 when bubvit.asm was unchanged and for an altered version called Case 1 when bubvit.asm was uniformized. In this case, the uniformization process used Table 6.1 and a uniformization value $u_r = 2$. In the case of Measurement 1, the program’s current waveform had a high current hump followed by a low flat portion. The purpose of the smoothening procedure was to bring the high current hump to the same level as the flat level. The current hump was caused by a program block that had high parallelism constructs. The average number of instructions per execution set within this block was greater than four. Also, many of the execution sets used memory operands. The last block of the program had an average of approximately two
instructions per execution set and had little memory addressing usage. Since the low current portion had an average of two instructions per execution set the uniformization value was set at 2 for the entire program. The two measurement results and the maximum current values (Imax1=0.2036 A and Imax2=0.1086 A) are shown in Figure 6.2. The discharging time of the bubvit.asm program given by Equation (6.2), based on these maximum current values, was:

\[ \tau_d (\text{bubvit.asm}) = \frac{7.75 \mu s - 1.45 \mu s}{ln \frac{0.2036 A}{0.1086 A}} \approx 10.055 \mu s \]  

(6.6)

The \( I_{\text{max}} \) value obtained by the fixed uniformization procedure based on Table 6.1 was too low and the current waveform was not at the desired level. The new current value sought was the average current consumption of the second block of the program and that was \( I_{\text{new}} = 0.1269 A \). By Equation (6.5), the extra time needed to expend the execution time of the unaltered program was:

\[ t_{\text{extra}} = 10.055 \mu s \cdot ln \frac{0.2036 A}{0.1270 A} \approx 4.745 \mu s \]  

(6.7)

Based on the above value, in order to bring the initial peak current value caused by the first block of the program to an average current equal with the average current of the second block, the initial version of the program needed NFI instructions distributed throughout the program such that at execution time, the new version executed a total number of 949 NFIs.

The new version of the program that used 949 NFIs was called \textit{Case 2} version. The number of NFIs needed was calculated by the following relation:

\[ N_{\text{case2}} = \frac{t_{\text{extra}}}{T_{\text{clock}}} = \frac{4.745 \mu s}{0.005 \mu s} = 949 \]  

(6.8)

The \textit{Case 1} version used a total number of NOPs equal with:

\[ N_{\text{case1}} = \frac{T_{\text{execution-case1}} - T_{\text{execution-case0}}}{T_{\text{clock}}} = \frac{7.77 \mu s - 1.45 \mu s}{0.005 \mu s} = 1264 \]  

(6.9)
ALU EQUIVALENCE TABLE FOR SC140 DSP PROCESSOR

<table>
<thead>
<tr>
<th>Source Code Transformation</th>
<th>Instructions</th>
<th>Equivalence in ALU Units</th>
</tr>
</thead>
<tbody>
<tr>
<td>Rule 1</td>
<td>All ALU Instructions</td>
<td>1 ALU unit</td>
</tr>
<tr>
<td>Rule 2</td>
<td>NOP</td>
<td>0 ALU units</td>
</tr>
<tr>
<td>Rule 3</td>
<td>All control instructions that perform a jump</td>
<td>3 ALU units</td>
</tr>
<tr>
<td>Rule 4</td>
<td>loopstart</td>
<td>neph;</td>
</tr>
<tr>
<td></td>
<td>{loop body}</td>
<td>{loop body}</td>
</tr>
<tr>
<td></td>
<td>loopend</td>
<td>loopend</td>
</tr>
<tr>
<td>Rule 5</td>
<td>All MOVEs to and from memory</td>
<td>3 ALU units</td>
</tr>
<tr>
<td>Rule 6</td>
<td>All VSL instructions</td>
<td>4 ALU units</td>
</tr>
<tr>
<td>Rule 7</td>
<td>All Delay Slot instructions</td>
<td>Non Delay Slot instructions</td>
</tr>
</tbody>
</table>

Table 6.2. General ALU equivalence chart for smoothening current in the SC140 processor.

Figure 6.3. Current plots. Case 0 versus Case 2 for bubvit.asm
A new correspondence table was created to best fit the instruction distribution and their individual current consumptions such that after the uniformization process the number of extra NFIs executed by bubvit.asm was to be 949. Table 6.2 presents the new ALU equivalence that generated 944 NFIs after uniformization within the bubvit.asm program. Figure 6.3 presents the current plot for Case 1 version versus Case 2 version. As seen from the measurements, the average current value for block 1 was closer in the Case 2 version to the average current value of block 2 than in the Case 1 version. Table 6.3 summarizes the important data characteristics for the three current waveforms generated by the three analyzed versions of the bubvit.asm program. The mean, standard variation (s), and range (Pk-Pk) measurements were taken after the start time of the program in order to eliminate the first rising portion of the trace. As seen from the data analysis presented in Table 6.3, the equivalence of Table 6.2 applied at the program source assembly level gave acceptable smoothening results, the Pk-Pk variation was reduced from 69.00mA to 27.90mA, and the standard variation was reduced from 21.802mA to 6.294mA. This reduction was obtained at an increase in the energy consumption of the program. However, for versions Case 0 and Case2, the total energy increase ratio was 1.51 versus the Pk-Pk decrease ratio of 2.47.

<table>
<thead>
<tr>
<th>Program Version</th>
<th>Program Start Time [\textmu s]</th>
<th>Block 1 End Time [\textmu s]</th>
<th>Program End Time [\textmu s]</th>
<th>Energy Block 1 [\textmu J]</th>
<th>Energy Program [\textmu J]</th>
<th>Mean [10.5 \textmu s , End] [mA]</th>
<th>s [10.5 \textmu s , End] [mA]</th>
<th>Pk-Pk [10.5 \textmu s, End] [mA]</th>
</tr>
</thead>
<tbody>
<tr>
<td>Case 0</td>
<td>9.905</td>
<td>11.355</td>
<td>15.115</td>
<td>0.4167</td>
<td>1.461</td>
<td>153.5</td>
<td>21.802</td>
<td>69.0</td>
</tr>
<tr>
<td>Case 2</td>
<td>9.905</td>
<td>16.100</td>
<td>19.860</td>
<td>1.318</td>
<td>2.213</td>
<td>118.9</td>
<td>6.294</td>
<td>27.9</td>
</tr>
</tbody>
</table>

Table 6.3. Data analysis for smoothening technique applied to bubvit.asm program.

It was found that the code transformation generated by the equivalence Table 6.2 gave good results for a larger class of programs. Figure 6.4 presents smoothening results obtained by applying the smoothening technique based on the equivalence Table 6.2 to a large assembly language program called polymulNIST.asm. The polymulNIST.asm implemented a cryptographic
Figure 6.4. Current smoothening technique based on Table 6.2 applied to the polymuNIST.asm application that was prone to power analysis attacks. Figure 6.4 demonstrates four current measurements performed on this application for a particular set of data inputs. The following is the current trace description of Figure 6.4:

- **M1**: This represented the current consumption for the unaltered program.

- **M2**: This represented the current consumption for an altered version that applied the source code transformations of Table 6.2 (with the exception of Rule 7) such that the uniformization factor $u_V = 2$.

- **M4**: This represented the current consumption for an altered version that applied the source code transformations of Table 6.2 (with the exception of Rule 7) such that the uniformization factor $u_V = 2$. Besides the NOP instructions, this version used the OR $d_n, d_n$ in order to bring the execution sets that had only one ALU instruction equivalence per execution set to 2 ALU equivalence.
M7: This represented the current consumption for an altered version that applied the same source code transformations used for the M2 trace plus Rule 7 wherever possible. The initial source code made use of the following delay-slot instructions: rtsd, jmpd, and btd. These instructions were converted to the non-delay slot instructions rts, jmp, and bt by placing the instruction from the delay slot before these instructions.

Table 6.4 summarizes the data analysis for the traces presented in Figure 6.4. The period of time used for the measurements presented in this table is indicated by the start time values and the end time values of columns 2 and 3. The smallest Pk-Pk variation was given by the current trace M7 that applied all the rules of the Table 6.2. The 4.74 ratio reduction of the M1’s Pk-Pk value versus M7’s Pk-Pk value was obtained with a ratio of 1.74 of the energies.

<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>M1</td>
<td>25</td>
<td>681</td>
<td>209.9</td>
<td>0.1689</td>
<td>18.846</td>
<td>67.75</td>
</tr>
<tr>
<td>M2</td>
<td>25</td>
<td>1569</td>
<td>363.2</td>
<td>0.1240</td>
<td>1.5903</td>
<td>20.28</td>
</tr>
<tr>
<td>M4</td>
<td>25</td>
<td>1545</td>
<td>359.9</td>
<td>0.1247</td>
<td>1.9072</td>
<td>16.80</td>
</tr>
<tr>
<td>M7</td>
<td>25</td>
<td>1569</td>
<td>365.5</td>
<td>0.1247</td>
<td>1.9787</td>
<td>14.30</td>
</tr>
</tbody>
</table>

Table 6.4. Data analysis for smoothening technique applied to polymulNIST.asm program.

The smoothening technique presented in this section was capable of slightly improving the security of cryptographic applications as shown in Section 6.2. Also by reducing the peak current values the overall average power per application was reduced and thus an improved reliability in relation to chip heating could be considered.

6.2 Energy Analysis for Secure Implementations

This section presents a methodology and results for analyzing the security against simple power analysis attacks. The analysis used an SC140 assembly language implementation polymulNIST.asm of the scalar multiplication algorithm of the form \( kP \), with \( P \) a fixed point on a known elliptic curve and \( k \) a secret key, introduced in Section 2.3.1 of Chapter 2.
The scalar multiplication, point addition, point subtraction, and point negation implementations followed the algorithms presented in Table 2.1 of Chapter 2. The coordinates of the elliptic curve points were represented in polynomial basis and the basic polynomial field operations (addition, reduction, multiplication, inversion) were implemented using SC140 assembly language optimized routines [65]. The executable code used a NIST elliptic curve over the $GF(2^{163})$ field with a fixed NIST base point and the following irreducible polynomial [65]: $x^{163} + x^7 + x^6 + x^3 + 1$.

Entire current traces using $k=163$ bits were impossible to correlate from one routine to another due to the large time intervals between the sample points. For large sampling intervals, the waveform aliasing was possible [58] and many block dependent characteristics were lost. For example, within a current trace of the elliptic curve implementation, the average time duration per one bit of $k$ at a frequency of 200MHz was approximately 130µs. For $k=163$ bits, the average execution time for $kP$ (polymulNIST.asm) was approximately 21.5ms. In this case a current collection at 200MHz for $k=163$ bits needed a sample frame of 25ms. At a maximum sample point rate of 4000 points [58], the oscilloscope sampled current points at every 6.25µs and thus skipped approximately 1,250 instructions. At this rate, many characteristics specific to the block composition were lost.

As an application to the current dynamics measurement approach introduced in this thesis, a fast way of assessing the security of the system was developed using energy consumption measurements and analysis with smaller keys ($k$ less than 6 bits). The small key values were used in order to reduce the amount of data collected. However, the experimental procedures applied to small key values are unchanged when large key values are used. The idea of the energy consumption analysis was to counterattack the randomness of the current traces and to overcome the similarities between the executable algorithmic blocks. The energy analysis technique presented was based on the simple observation that different algorithmic operations could not be efficiently implemented through exactly the same line of coding and instruction sequencing. For example, when the start point and the end point of the execution time for two particular algorithmic operations, say point addition and point subtraction, were identified, a different energy consumptions per block was expected (See Figure 4.2). As a result, an easy way to tell if an implementation was vulnerable to power analysis attacks was to calculate energy consumptions for
random traces and compare them to known traces.

The non-adjacent form (NAF) of an integer $k$ (or signed binary expansion, using 0, 1, -1, where no two consecutive coefficients are nonzero [61]) was used in the scalar multiplication implementation. Based on the NAF definition, the bit that followed a "1" or a "-1" bit was always "0". As a result, the first two most significant bits of the unknown random $NAF(k)$ representations were known to be "10". The process of analyzing the random trace $RT$ (or trace under attack) for $NAF(k)$ and of predicting the bits of $k$ was performed in sequential stages. The execution time and the energy consumption per bit analyzed at each stage were functions of the previous bits analyzed. The time intervals per bit $t_{Bi}$ were divided into smaller time intervals in order to use the energy distribution values along the trace for accurate bit predictions.

Figure 6.5 presents all possible situations and prediction outcomes that could take place at Stage i+1 when the predicted bit at Stage i was 0, 1, or -1. When the predicted bit at Stage i was determined to be 1 or -1, the predicted bit at Stage i+1 was automatically set to 0. In Figure 6.5, at Stage i the predicted NAF bits were $(10P_2 P_3 \cdots P_i)$. For the analysis technique at Stage i+1, it was assumed that reference current traces could be generated for the following NAF bit
Figure 6.6. Theoretical traces and timing setup for energy analysis methodology of a random trace RT.

representations: \((10P_2P_3\cdots P_i00)\), \((10P_2P_3\cdots P_i1)\), and \((10P_2P_3\cdots P_i-1)\). The total energy consumption and the energy distribution values per bit analyzed for the reference traces was compared with the energy consumption and the energy distribution values of the corresponding bit position from the random trace. The bit of the reference trace that had the best energy distribution matches with the energy distributions of the random trace was chosen as the next predicted bit for Stage \(i+1\).

The bit analysis strategy presented in Figure 6.5 together with the traces and data information shown in Figure 6.6 were developed into a bit prediction technique that was called "energy analysis for secure implementation" or EASI technique. Using the EASI technique, a cryptographic implementation could be declared resistant to SPA if the EASI analysis resulted in early miss-prediction for a large number of target traces.
for all the traces. The algorithmic steps for the EASI technique are presented in Table 6.5. The ability of the EASI technique in accurately predicting the $NAF(k)$ bit combination for the SC140 implementation was related to the sampling interval length of the oscilloscope. As a result, the analysis presented in this section used successive smaller portions of current traces in order to reduce the sample interval. It was found that the EASI technique proved to be successful in predicting key bits values when the sampling interval used was under 6.5ns.

Table 6.5. EASI technique algorithm.

for all the traces. The algorithmic steps for the EASI technique are presented in Table 6.5. The ability of the EASI technique in accurately predicting the $NAF(k)$ bit combination for the SC140 implementation was related to the sampling interval length of the oscilloscope. As a result, the analysis presented in this section used successive smaller portions of current traces in order to reduce the sample interval. It was found that the EASI technique proved to be successful in predicting key bits values when the sampling interval used was under 6.5ns.

Table 6.5. EASI technique algorithm.

for all the traces. The algorithmic steps for the EASI technique are presented in Table 6.5. The ability of the EASI technique in accurately predicting the $NAF(k)$ bit combination for the SC140 implementation was related to the sampling interval length of the oscilloscope. As a result, the analysis presented in this section used successive smaller portions of current traces in order to reduce the sample interval. It was found that the EASI technique proved to be successful in predicting key bits values when the sampling interval used was under 6.5ns.

Table 6.5. EASI technique algorithm.

for all the traces. The algorithmic steps for the EASI technique are presented in Table 6.5. The ability of the EASI technique in accurately predicting the $NAF(k)$ bit combination for the SC140 implementation was related to the sampling interval length of the oscilloscope. As a result, the analysis presented in this section used successive smaller portions of current traces in order to reduce the sample interval. It was found that the EASI technique proved to be successful in predicting key bits values when the sampling interval used was under 6.5ns.
Sections 6.2.1 and 6.2.2 present analysis of current traces for random values of $k$ that were smaller than 6 bits. These traces were assumed to simulate smaller portions of a longer trace and the results in these cases were easily verified. In all of the experimental results that follow, the processor was running at 200MHz frequency and the oscilloscope's sampling interval was 5\text{ns}.

The bit prediction per stage was done automatically by a MATLAB SIMULINK prediction module. The block diagram of the simulator for one stage is presented in Figure 6.7. The stage simulator module had as inputs the reference and the random traces collected by the sampling scope, the start times and the end times of the bit analyzed for the reference and random traces. The module calculated and recorded the energy values at successive intervals of 10\text{\mu}s. These energy values were used in the predictive process. When the algorithm took a different route than
the one assumed in the reference trace, the energy values of the reference trace showed small variations in comparison with the random trace. The block simulator presented in Figure 6.7 used current waveforms collected for the polymulNIST.asm program as application example. The current reference traces (t08.mat, t05.mat, t03.mat) were inputted in MATLAB format (5ns sampling interval) to the ‘Energy per Bit2 Reference Traces’ block and the random trace (tran2.mat) was inputted to the ‘Energy per Bit2 Random Trace’ block. These energy blocks used a continuous-time integrator to calculate the energy consumption at 10 $\mu$s intervals. The start times for each trace, the voltage supply (1.898 V), the attenuation factor (0.2), and the DC current component (0.074 A) were constants to these blocks. The simulator used the ‘Trace Alignment Energy Distribution Differences’ block to align the traces and calculate the difference traces for the stage. The difference traces (diff08, diff05, diff03) were used to calculate the bit prediction indicators. The indicators were used in the ‘Bit Prediction’ block to determine for this stage the reference trace that most resembled with the random trace. In the case shown in Figure 6.7, the Stage 2 simulator chose the reference trace ‘8’, that was t08.mat, as indicated in the ‘Trace Selected’ block.

6.2.1 EASI Results for polymulNIST.asm Implementation

This section presents detailing results for the EASI technique applicable to the polymulNIST.asm program executing with small key values. In this analysis, the MATLAB traces tran2.mat, tran3.mat, tran5.mat were the current traces of the Stage 2, Stage 3, and Stage 5 analysis pertaining to the random trace tran.mat. The reference traces used in this analysis were: t08.mat, t05.mat, and t03.mat for Stage 2; t16.mat, t09.mat, and t07.mat for Stage 3; and t56.mat, t29.mat, and t27.mat for Stage 5 analysis. The number of the reference trace was the decimal expression of the key value used in the polymuNIST.asm program when the trace was collected. For example t27.mat trace was the current trace at Stage 5 analysis that was obtained for a key value $k=27$. The NAF representation of this key was $k=100-10-1$. All of the start and end times for the random and reference traces plus the NAF representations for the reference traces are indicated in the table results per stage. The stair graph plots of the energy differences for the Stage 2, Stage 3 and Stage 5 of the EASI algorithm applied to the polymulNIST.asm program are presented in Figures 6.8, 6.9,
and 6.10. Variances of sets of 21 energy difference values were used as an indicator for predicting the stage bit. Table 6.6 summarizes the energy values per intervals of all the traces used in Stage 2 analysis. These energy values were stored in the following vectors: tran2, t08, t05, and t03. The difference vectors, diff08=tran2-t08, diff05=tran2-t05, and diff03=tran2-t03 were used in the bit prediction process. In this table, the variances of these differences were used as a bit prediction indicator. The variance value var(diff08) had the lowest value and as a result, the current reference trace t08.mat was the one that most resembled the random trace tran.mat and bit2=0 was the prediction at Stage 2. The outcome result was also visible from Figure 6.8 where Diff08 stair plot (in green) showed the lowest variation.

Table 6.7 summarizes the energy values per interval of all the traces used in Stage 3 analysis. These energy values were stored in the following vectors: tran3, t16, t09, and t07. The difference vectors, diff16=tran3-t16, diff09=tran3-t09, and diff07=tran3-t07 were used in the bit prediction process. The variance value var(diff07) had the lowest value and as a result, the current reference trace t07.mat was the one that most resembled the random trace. Bit3=-1 was the predicted bit at Stage 3. The outcome result at Stage 3 was also visible from Figure 6.9 where Diff07 stair plot (in red) showed the lowest variation. Since a bit of -1 was predicted at this stage, the bit for Stage 4 was automatically set to bit4=0.

Finally, Table 6.8 summarizes the energy values of all the traces used in Stage 5 analysis. These energy values were stored in the following vectors: tran5, t56, t29, and t27. The difference vectors used in the prediction process at this state were diff56=tran5-t56, diff29=tran3-t29, and diff27=tran5-t27. The variance value var(diff29) was shown in Figure 6.10 and was calculated in this table to have the lowest value and as a result, the current reference trace t29.mat was selected and bit5=1 was predicted.

The overall prediction was k=10-101 as the NAF representation for the random k. The prediction was valid and as a final result, the polymulNIST.asm implementation was declared unsafe against simple power analysis attacks.
Figure 6.8. Stage 2 energy stair graph for the polymulNIST.asm program.

<table>
<thead>
<tr>
<th>[Tstart, Tend], [µs]</th>
<th>Energy at: $t_n = T_{start} + n \cdot 10 \mu s$ ; $n=1,2,\ldots,20$, and $t_{21} = T_{end}$ [µJ]</th>
<th>Var/Bit</th>
</tr>
</thead>
<tbody>
<tr>
<td>t08 = T(1000), [113.800, 304.465]</td>
<td>0 1.7269 3.2593 4.7281 6.1454 7.5554 8.6202</td>
<td>var(diff08)=3.8594e-006</td>
</tr>
<tr>
<td>t05 = T(101), [113.345, 303.670]</td>
<td>0 1.7266 3.2585 4.7273 6.1445 7.5545 8.6197</td>
<td>var(diff05)=2.1663e-004</td>
</tr>
<tr>
<td></td>
<td>20.3968 21.5804 22.8076 24.1867 25.5984 27.0004 27.0549</td>
<td></td>
</tr>
<tr>
<td>t03=T(10^-1), [113.400, 303.960]</td>
<td>0 1.7265 3.2583 4.7270 6.1440 7.5541 8.6192</td>
<td>var(diff03)=3.0659e-004</td>
</tr>
<tr>
<td>tran2 [114.910, xxx.xxx]</td>
<td>0 1.7254 3.2556 4.7234 6.1399 7.5497 8.6160</td>
<td>⇒ Bit 2 = 0</td>
</tr>
<tr>
<td></td>
<td>20.3759 21.5508 22.7949 24.1616 25.5891 27.0217</td>
<td></td>
</tr>
<tr>
<td>t21: energy ⇒</td>
<td>305.575: 27.1365 305.235: 27.0826 305.470: 27.1208</td>
<td></td>
</tr>
</tbody>
</table>

Table 6.6. EASI technique for polymulNIST.asm Stage 2, time and energy analyses, and bit 2 prediction.
Figure 6.9. Stage 3 energy stair graph for the polymulNIST.asm program.

![Stage 3 Energy Distribution Plot](image)

Table 6.7. EASI technique for polymulNIST.asm. Stage 3, time and energy analyses, and bit 3 and bit 4 predictions.

<table>
<thead>
<tr>
<th>Tstart, Tend, [(10^\mu s)]</th>
<th>Energy at: (t_n = T_{\text{start}} + n \cdot 10^{\mu s}); (n=1,2,...,20), and (t_{21} = T_{\text{end}}) [(\mu J)]</th>
<th>Var / Bit</th>
</tr>
</thead>
<tbody>
<tr>
<td>(t_{16} = T(10000), [209.955, 403.035])</td>
<td>0</td>
<td>1.7199</td>
</tr>
<tr>
<td>(t_{09} = T(1001), [209.490, 399.300])</td>
<td>0</td>
<td>1.7199</td>
</tr>
<tr>
<td>(t_{07} = T(100-1), [209.540, 399.650])</td>
<td>0</td>
<td>1.7195</td>
</tr>
<tr>
<td>(\text{tran3}, [210.560, \text{xxx.xxx}])</td>
<td>0</td>
<td>1.7184</td>
</tr>
<tr>
<td>(t_{21}:) energy (\Rightarrow)</td>
<td>403.640: 27.5709</td>
<td>400.370: 26.9770</td>
</tr>
</tbody>
</table>
Figure 6.10. Stage 5 energy stair graph for the polymulNIST.asm program.

<table>
<thead>
<tr>
<th>[Tstart,Tend] [μs]</th>
<th>Energy at: tn = Tstart + n \cdot 10 \mu s \ ; n=1,2,\ldots,20, and t21 = Tend [μJ]</th>
<th>Var/Bit</th>
</tr>
</thead>
</table>

Table 6.8. EASI technique for polymulNIST.asm. Stage 5, time and energy analyses, and bit 5 prediction.
Figures 6.11 presents an example of the AC current collection for two current traces used in Stage 2 analysis. The waveform plots present a portion of the currents captured by the AC current probe (magnified by a factor of 5). The waveforms are over an interval of 40 µs starting with $T_{\text{start}}+140\mu s$ for each trace (see Table 6.6). The current variation within these traces showed small differences from trace-to-trace that were visible at these plotting ratios. Nevertheless, the differences were hard to correlate to the specific algorithmic block and when the sample interval was increased, these differences were not visible anymore.

6.2.2 EASI Results for polymulNISTsmooth.asm Implementation

The smoothening technique introduced in Section 6.1 was applied to the polymulNIST.asm implementation and the results of the EASI analysis are presented in this section. The new application obtained by applying the smoothening technique of Section 6.1 (Table 6.2) was called
Figure 6.12. Stage 2 energy stair graph for the polymulNISTsmooth.asm program.

polymulNISTsmooth.asm. In this analysis, the MATLAB traces tran2sm.mat, tran3sm.mat, tran5sm.mat were the current traces of the Stage 2, Stage 3, and Stage 5 analysis pertaining to the random trace transm.mat. The reference traces used in this analysis were: t08sm.mat, t05sm.mat, and t03sm.mat for Stage 2; t16sm.mat, t09sm.mat, and t07sm.mat for Stage 3; and t56sm.mat, t29sm.mat, and t27sm.mat for Stage 5 analysis. Like in Section 6.2.1 the number of the reference trace was the decimal expression of the key value used in the polymulNISTsmooth.asm program when the trace was collected.
Table 6.9. EASI technique for polymulNISTsmooth.asm. Stage 2, time and energy analyses, and bit 2 prediction.

<table>
<thead>
<tr>
<th>Trace: [Tstart, Tend] [\mu s]</th>
<th>Energy Values at: ( t_n = T_{\text{start}} + n \cdot 10 \mu s ); ( n = 1, 2, \ldots, 44 ), and ( t_{45} = T_{\text{end}} ) [\mu J]</th>
<th>Variance Values/Bit Predicted</th>
</tr>
</thead>
<tbody>
<tr>
<td>t08sm: T(1000) [248.610, 687.260]</td>
<td>0 1.4448 2.8819 4.3063 5.7377 7.1555 8.5806</td>
<td>var(diff08) = 1.6500-006</td>
</tr>
<tr>
<td></td>
<td>29.9079 31.3109 32.7637 34.1900 35.6247 37.0501 38.4681</td>
<td></td>
</tr>
<tr>
<td></td>
<td>39.8954 41.3414 42.7387 44.1592 45.5807 47.0084 48.4276</td>
<td></td>
</tr>
<tr>
<td></td>
<td>49.8698 51.2947 52.6943 54.1100 55.5179 56.9830 58.3976</td>
<td></td>
</tr>
<tr>
<td></td>
<td>59.8041 61.2180 62.4320</td>
<td></td>
</tr>
<tr>
<td>t05sm: T(101) [247.730, 686.210]</td>
<td>0 1.4449 2.8820 4.3063 5.7379 7.1558 8.5809</td>
<td>var(diff05) = 2.9224-005</td>
</tr>
<tr>
<td></td>
<td>29.9090 31.3104 32.7538 34.1887 35.6176 37.0440 38.4687</td>
<td></td>
</tr>
<tr>
<td></td>
<td>39.8907 41.3171 42.7437 44.1648 45.5855 47.0096 48.4324</td>
<td></td>
</tr>
<tr>
<td></td>
<td>49.8551 51.3079 52.6965 54.1110 55.5182 56.9753 58.4047</td>
<td></td>
</tr>
<tr>
<td></td>
<td>59.8070 61.2252 62.4120</td>
<td></td>
</tr>
<tr>
<td>t03sm: T(10^-1) [247.795, 686.890]</td>
<td>0 1.4446 2.8816 4.3058 5.7371 7.1548 8.5798</td>
<td>var(diff03) = 3.0692e-005</td>
</tr>
<tr>
<td></td>
<td>29.9068 31.3094 32.7536 34.1864 35.6152 37.0419 38.4662</td>
<td></td>
</tr>
<tr>
<td></td>
<td>39.8866 41.3135 42.7403 44.1634 45.5813 47.0070 48.4312</td>
<td></td>
</tr>
<tr>
<td></td>
<td>49.8480 51.3004 52.6971 54.1080 55.5307 56.9682 58.4012</td>
<td></td>
</tr>
<tr>
<td></td>
<td>59.8044 61.2219 62.5056</td>
<td></td>
</tr>
<tr>
<td>tran2sm: [250.565, xxx.xxx]</td>
<td>0 1.4448 2.8817 4.3059 5.7371 7.1548 8.5797</td>
<td>( \Rightarrow ) Bit 2 = 0</td>
</tr>
<tr>
<td></td>
<td>10.0072 11.4297 12.8480 14.2681 15.6924 17.1109 18.5471</td>
<td></td>
</tr>
<tr>
<td></td>
<td>29.9056 31.3086 32.7613 34.1875 35.6222 37.0473 38.4651</td>
<td></td>
</tr>
<tr>
<td></td>
<td>39.8921 41.3107 42.7354 44.1555 45.5772 47.0047 48.4242</td>
<td></td>
</tr>
<tr>
<td></td>
<td>49.8658 51.2908 52.6906 54.1066 55.5145 56.9787 58.3941</td>
<td></td>
</tr>
<tr>
<td></td>
<td>59.8005 61.2145</td>
<td></td>
</tr>
</tbody>
</table>

143
EASI results for the Stage 2 and Stage 3 of the polymulNISTsmooth.asm implementation are presented in Figures 6.12 and 6.14, and Tables 6.9 and 6.10. The figures present the stair energy graphs and the tables summarize all of the energy calculations. The EASI technique applied to this new application was again able to predict the right bits at all stages, but the variance values were substantially reduced in all the stages analyzed. Due to longer time execution introduced by the smoothening technique, the number of intervals analyzed had increased from 21 to 45. Also, the energy consumption per bits had increased, due the longer time executions.
Trace:  
[Tstart, Tend]  
[μs]  

<table>
<thead>
<tr>
<th>Energy Values at: t_n = Tstart + n \cdot 10μs; n=1,2,…44, and t_45 = Tend</th>
<th>[μJ]</th>
<th>Variance Values/Bit Predicted</th>
</tr>
</thead>
<tbody>
<tr>
<td>t16sm: T(10000) [469.710, 914.780]</td>
<td></td>
<td>var(diff16) = 6.1207e-004</td>
</tr>
<tr>
<td>0</td>
<td>1.4508</td>
<td>2.8768</td>
</tr>
<tr>
<td>29.9073</td>
<td>31.3157</td>
<td>32.7639</td>
</tr>
<tr>
<td>39.8967</td>
<td>41.3243</td>
<td>42.7436</td>
</tr>
<tr>
<td>49.8583</td>
<td>51.3123</td>
<td>52.7128</td>
</tr>
<tr>
<td>59.8154</td>
<td>61.2292</td>
<td>62.3556</td>
</tr>
<tr>
<td>t09sm: T(1001) [468.840, 905.680]</td>
<td></td>
<td>var(diff09) = 9.6102e-006</td>
</tr>
<tr>
<td>0</td>
<td>1.4509</td>
<td>2.8773</td>
</tr>
<tr>
<td>29.9098</td>
<td>31.3166</td>
<td>32.7615</td>
</tr>
<tr>
<td>39.8967</td>
<td>41.3180</td>
<td>42.7427</td>
</tr>
<tr>
<td>49.8762</td>
<td>51.3049</td>
<td>52.6999</td>
</tr>
<tr>
<td>59.8160</td>
<td>61.2351</td>
<td>62.1935</td>
</tr>
<tr>
<td>t07sm: T(100-1) [468.910, 906.370]</td>
<td></td>
<td>var(diff07) = 2.9575e-006</td>
</tr>
<tr>
<td>0</td>
<td>1.4507</td>
<td>2.8769</td>
</tr>
<tr>
<td>29.9091</td>
<td>31.3265</td>
<td>32.7627</td>
</tr>
<tr>
<td>39.8956</td>
<td>41.3175</td>
<td>42.7407</td>
</tr>
<tr>
<td>49.8705</td>
<td>51.3104</td>
<td>52.6977</td>
</tr>
<tr>
<td>59.8125</td>
<td>61.2304</td>
<td>62.2823</td>
</tr>
<tr>
<td>tran3sm: [470.750, xxx.xxx]</td>
<td></td>
<td>var(diff16) = 6.1207e-004</td>
</tr>
<tr>
<td>0</td>
<td>1.4511</td>
<td>2.8768</td>
</tr>
<tr>
<td>29.9038</td>
<td>31.3212</td>
<td>32.7575</td>
</tr>
<tr>
<td>39.8918</td>
<td>41.3143</td>
<td>42.7371</td>
</tr>
<tr>
<td>49.8670</td>
<td>51.3083</td>
<td>52.6952</td>
</tr>
<tr>
<td>59.8110</td>
<td>61.2290</td>
<td>62.2827</td>
</tr>
</tbody>
</table>

Table 6.10. EASI technique for polymulNISTsmooth.asm. Stage 3, time and energy analysis, and bit 3 and bit 4 predictions.
Table 6.9 shows energy values for the Stage 2 analysis. At this stage the energy values were stored in the following vectors: tran2sm, t08sm, t05sm, and t03sm. The difference vectors, diff08sm=tran2sm-t08sm, diff05sm=tran2sm-t05sm, and diff03sm=tran2sm-t03sm were used in the bit prediction process. The variance value var(diff08sm) had the lowest value and as a result, the current reference trace t08sm.mat was the one that most resembled the random trace and bit2=0 was the predicted bit at Stage 2. Stage 3 analysis is presented in Table 6.10. At Stage 3, the energy values were stored in the following vectors: tran3sm, t16sm, t09sm, and t07sm. The difference vectors that were used in this case in the bit prediction process were diff16sm=tran3sm-t16sm, diff09sm=tran3sm-t09sm, and diff07sm=tran3sm-t07sm. The variance value var(diff07) had the lowest value and the current reference trace t07sm.mat was the one that the most resembled the random trace. This resulted in a bit3=-1 prediction at Stage 3 and an automatic prediction bit4=0 for Stage 4.
Figure 6.15. Example of smoothened current traces (tran2sm.mat and t08sm.mat from Tstart+350us to Tend+390us) at Stage 2 EASI analysis.

The outcome results presented in Tables 6.9 and 6.10 were also visible in Figures 6.12 and 6.13. In Figure 6.12, the Diff08 stair plot (in green) showed the lowest variation and in Figure 6.13, the Diff07 stair plot (in red) showed the lowest variation.

The last stage of the EASI algorithm, Stage 5, was obvious since the reference trace t29sm was identical with the random trace and the variance of the vector that stored the energy differences between these traces was zero. The overall prediction was again k=100-101 as the NAF representation for the random k. The prediction was valid and as a final result the polymulNISTsmooth.asm implementation was also declared unsafe against simple power analysis attacks but safer than the polymulNIST.asm implementation since the EASI variance indicators showed smaller values in the case of the polymulNISTsmooth.asm analysis.
Figures 6.14, 6.15, and 6.16 present examples of the AC current plots for the current traces used in Stage 2 analysis of the polymuNISTsmooth.asm program. The plots presented portions of the currents captured by the AC current probe (magnified by a factor of 5). In Figure 6.14 the current plots are over the entire execution time of bit 2. In Figures 6.15 and 6.16, the plots are over an interval of $40\,\mu s$ starting with $T_{\text{start}}+350\,\mu s$ for each trace. The current variation within these traces showed visible differences from trace-to-trace, but in comparison with the currents of the polymuNIST.asm implementation, the current range of variation was much smaller (See also Figure 6.11).
In this section, a processor hardware extension for security is proposed. The purpose of this extension was to increase the security of the processor against power analysis attacks by dynamically controlling the current consumption of the processor. The hardware description tries to dynamically control the current drawn by the processor within an upper and lower bound set by the user. Hence a random or smooth current trace can be supported in theory.

A power analysis attack resistant (PAAR) architecture was proposed as an application to the smoothening technique in hardware. The software smoothening could not take into account the data variation during the program execution. This limitation could possibly be eliminated by the hardware architecture to be presented. In this section, the PAAR architecture was assumed to be connected to the SC140 processor.

Figure 6.17 presents the PAAR architecture and a hypothetical connection to the SC140 processor. The hardware security improvement against power analysis attacks was envisioned as an add-in block to an existing processor's architecture. However, principles of the PAAR block could be used possibly with other processors as well since the connection between the block and the processor were minimal. The PAAR is a closed loop type architecture that controls dynamically the current consumption of a processor within a programmed range. The PAAR block has software programmable capabilities and can be activated only when needed. After reset, the block's signals are inactive until the current level registers are programmed. The PAAR block has 6 control lines and one 8-bits output data bus. A current resistor $R_p$ is connected between the power supply line $V_{dd}$ and the power supply entry to the processor core. Two voltage signals, $V_1$ and $V_2$, are used to measure the voltage drop across the current resistor $R_p$. The voltage drop across the current resistor is proportional with the processor's current at all times. A differential amplifier $DA_1$ is used to bring the small signal value of the voltage drop to a comparable signal level $C_{LP}$ that is proportional with the instantaneous current consumption of the processor. The architecture assumes that two current level registers are loaded with the binary values $L_1$ and $L_2$. These values can be loaded every time a secure communication is executing. The binary values $L_1$ and $L_2$ are converted by digital to analog converters into the current levels $CL_1$ and $CL_2$. The current levels
CL1 and CL2 are comparable with the current level CL generated by the DA1 block. The current level CL1 represents the minimum current value allowed and the CL2 represented the maximum current level allowed. The current levels CL1 and CL2 are compared with the instantaneous current level CL by two comparator blocks C1, and C2, respectively. The comparator blocks C1 and C2 produce the digital output signals of the PAAR architecture to the processor core. The output signals are ‘Cut Current Signal’ (CCS) and ‘Increase Current Signal’ (ICS). The CCS signals that the processor’s instantaneous current is above the allowed maximum value and the ICS signals that the processor’s instantaneous current is below the allowed minimum value.

In the case of the SC140 processor, these signals would be connected to the PSEQ block. When the CCS signal would be active, the PSEQ would insert series of NOP instructions until the CCS signal would be triggered to its inactive state. When ICS signal would be active, the PSEQ would insert series of other NFI instructions (See Section 6.1) that have high current consumption levels until the ICS signal would be triggered to its inactive state. However, when the current consumption of the processor must be brought to a level that is less or equal with the minimum
current consumption registered for the program, the L1 register can be loaded to zero and the current control would be done only by the use of the L2 register.

By the use of the current simulator, a current projection of the processor while executing a security application was possible. Figure 6.13 presents the current projection of the SC140 processor while executing the polymul.asm routine. The current projection assumed that the PAAR block had a constant current consumption that was not significant to this simulation. Figure 6.20 plots with blue the measured current and with red the simulated current when the PAAR block was inactive. The figure also shows with magenta and black, the simulated currents when the PAAR block was active with the CL2 level at 70mA, and at 60mA, respectively. In these cases the CL1 level was zero.
Table 6.11 presents the execution times, the mean, max, Pk-Pk, energy, and standard deviation (s) values for the measured current and the simulated currents. The standard deviation was calculated over the time interval $T = [2500\text{ns}, 21300\text{ns}]$ for all of the current traces presented in Figure 6.18. As shown by the Pk-Pk and s values the current variation decreases when the PAAR block is activated. However, longer execution times and energy consumptions were recorded for the simulated currents when the PAAR block was activated. As a result, the hardware as the software smoothening technique showed an increase in the energy consumption and the execution time per program. Theoretically this architecture could assure maximum security against power analysis attacks by trading energy consumption and execution time for security.

Another way of reducing the peak current values and smoothen the current waveforms would be by dynamically changing the frequency of the processor while the program executes. The blocks that generate high current consumption peaks would be executed at a lower frequency as soon as the maximum current limit is passed. By using the gamma modeling technique, the results of this approach could be theoretically projected. Nevertheless, due to block succession, the current jumps when the frequency was changed could not be simulated. The modeling of transitional currents could not be captured by the gamma model and new adjustments are needed. The development board used did not allow dynamic frequency change and no experimental results were possible.
DISCUSSIONS AND CONCLUSIONS

This thesis introduces for the first time an instruction-level based model for instantaneous current consumption in a VLIW DSP processor (referred to as the dynamic current model). Current measurements and modeling methodologies at the instruction and program level were presented. Current simulation waveforms with calculation of average currents and energy consumptions were presented for a large class of programs. The current dynamics model captured 98% of the variation of the instantaneous current for 6 complex diverse applications, with an average RMS error of less than 2.7% of the average measured mean. The average energy estimates for the programs considered was within 1% from the measured values. The current dynamics of the simulated waveforms followed closely the measured waveforms. The square of the correlation and the RMS error between the simulated and measured waveforms were over 99%, and less than 2mA, for the short programs. For the long program (polymul.asm) these values were over 93% and under 7.1mA, respectively. The current measurements and methodologies were applied to analysis of security and reliability. Comparisons of the instantaneous current model with other current models, discussions, and limitations of the methodologies introduced, and future work are presented in the following sections of this chapter.

7.1 Comparisons to Previous Research

The major difference between the current dynamics model proposed in this thesis and the previous instruction-level research for processors is that previous power models are based on DC average current estimates at the instruction and program level, while the dynamic current model is based on instantaneous current simulation. As a result, a broader application space applies to the dynamic power model such as: verifying the design’s power constraints in embedded designs [1], [6]; software power optimization analysis [6]; and power-aware designs in communications [11]; reliability [44], [69], [70], [72] and security [15].
The modeling and analysis presented in [6], [7], [14], [24], and [26] have similarities with the dynamic power model in the sense that instruction level methodologies and physical current measurements are used. The power models presented in [25], [27], and [31] are also instruction based models but unlike the dynamic power model these models base their methodologies on simulated processor architectures and no physical current measurement are performed. Power models researched at lower levels than system level use simulation of functional units or small circuits [2], [23], [28], [29]. These models are efficient in the VLSI design process at the functional and circuit level. However, the power estimation of a processor at the instruction or program level becomes impractical [6] by using these models.

So far, the instruction based power models applicable to commercial processors use average current measurements and are derived from the research introduced in [6]. In Section 5.4 an extended comparison analysis between the methodologies introduced in [6] and the methodologies introduced in this thesis was presented. A comparison of previous researched power modeling methodologies and the current dynamics methodologies presented in this thesis is given below:

- Previous research performed average current measurements on infinitely looped instructions and programs [6], [7], [14], [24], [26], [27]. In this research the instructions and programs to be measured were not looped, instead an interrupt based technique that was capable of generating one time run measurements was used (See Section 3.2).

- Previous research used average current consumption measurements obtained by the use of a low frequency ammeter [6], [7], [14], [24] or by the use of an oscilloscope [26]. A rather sophisticated method with AD-converters and integrators was used in [37] to generate power per cycle measurements. This research used instantaneous current measurements collected by the use of a high frequency sampling oscilloscope (See Section 3.2).

- Previous research used average current measurements at the instruction level to build an average base current cost at the program level [6], [7], [24], [26]. This research used maximum current variation values to build a base current waveform at the program level (See Sections 4.4, 5.2 and 5.3).
• Unlike previous research that used experiments at one low working frequency (40MHz in [6], [7], [24], and [37] and 33MHz in [26]), this research used experiments at three different high frequencies: 100MHz, 200MHz, and 300MHz (See Chapter 5).

• Unlike previous research ([6], [26]) where the current models targeted processors with cache support, pipelining, and low level of parallelism (maximum 2 operations in parallel), the current dynamics model targeted a complex multiple issue processor core with high level of parallelism (maximum 6 operations in parallel) and a VLIW architecture but no cache support (See Section 3.1).

• In [23], the effects of input data switching activity on power are considered at module level. In [6], [7], [24], and [26], the current cost impact caused by register names, operand values, memory locations were taken into consideration by using averaged measurements. Similarly, this research averages these effects at the execution stage level (functional unit level) by considering mixed/random operand values (See Section 4.2.3).

• Previous research considered inter-instruction effects as per-instruction overheads in pair tables [7], [27] or additive constants derived from the processor’s architecture characteristics [6], [26]: circuit state (modeled by a fixed value at the processor level), resource constraints (modeled by a fixed value at the processor level), and the cache miss (modeled by experimental values derived from cache simulators). Similarly the current dynamics model considered architecture specific effects by using current superposition principles and circuit state dependency on parallelism and sequencing (See Chapter 4).

• Previous research used inter-instruction effects for successive instructions that are different, as an overhead cost per instruction related to the NOP instruction [27]. The current dynamics model implicitly takes into account these inter-instruction effects by using the total current superposition principle where a NOP-level and a variable component above NOP-level are considered (See Section 4.2.1).

• Similarly to statistical approaches in [14] this research also used linear regression for power scaling. However, in this research the least square-algorithm model was used in the current
simulation process to generate a numerical solution for the factorial adjustment as a dependency on parallelism and sequencing (See Section 5.3).

- In VLIW architectures, previous research separated the energy consumption into energy per pipe stages with the execution stage level considered separable and “functional additive [25]”. Similarly, this research used the pipelining superposition principle as a characteristic of the pipeline architecture and used stage separation. The parallel issue superposition principle was a characteristic of the functional separability at the execution stage of the parallel executing instructions (See Sections 4.2.2 and 4.2.4).

- Most of the previous research validated the instruction based power models with a few examples. In [6] and [7], one program was used that executed for approximately 72 clock cycle, and 66 clock cycles, respectively. The current dynamics model was validated on six diverse programs that execute for 167, 207, 282, 750, 1044, and 4264 clock cycles (See Chapter 5).

- The approximating errors of the energy estimates that were compared to physical measurements presented in the previous research were greater than the approximating energy estimates obtained by the current dynamics model. In [6], [7], [24] and [26] the energy estimates were within 3%, 10.5%, 4%, and 8%, respectively. Similar results were reported for the simulated models [25], [27], [31]. For example, in [27] the energy estimation error was within 10% for all of the power models presented. In this thesis energy estimates within 1.3% for the 5 short programs and within 1.9% for the long program were obtained (See Section 5.3).

- Unlike previous research to my knowledge, this research was capable of conveying instantaneous current consumption through simulated current waveforms.

In contrast to the energy models presented in [6], [7], [14], [24], and [26] the models introduced in [23] and [30] take into consideration the dynamic characteristic of the energy consumption at the clock-cycle. The model presented in [23] applies to the circuit level while the model presented in [30] applies to the processor level. However, the fundamentals of the model presented in [30] are different than those of the dynamic current model. In [30], average power estimates, at the clock
cycle level derived from data sheet power specifications, were used, rather than current waveforms used by this research. Direct current measurement techniques employed by [6], [7], [14], [24], [26], and [37] proved to be unstable for the target processor used in this thesis by creating voltage fluctuations on the power supply pin that were up to 15% of the power supply value (See Section 3.2). As a result, this thesis used inductive current measurement techniques that proved to be suitable for low voltage supply processor technology.

Besides the current modeling contributions, the research presented in this thesis has contribution to the area of security. In [67], smoothening of current traces was realized by using capacitors in order to increase the security of smart cards. In this research, assembly code transformations were used to achieve similar results by leveling down the high current peaks (See Section 6.1). Also, this technique could possibly increase the reliability of a microprocessor based system by decreasing the average power consumption, which is responsible for chip heating. Unlike the power analysis attacks introduced in [15], [16], [17] that use power analysis, the EASI technique uses energy distribution analysis over time intervals that include more instructions (See Section 6.2). As a result, the security indicators reflect intervals that include large number of instructions rather than single instructions. Techniques that introduce randomized execution into a conventional processor (random out of order issue and random register renaming) [68] in order to produce temporal misalignment of traces to defeat the DPA attack, would most likely have an insignificant effect on the EASI based techniques. Unlike the randomized methods of hardware architecture [68] that have the final effect of changing the power consumption pattern, the PAAR architecture introduced in this thesis targets a dynamically controlled current level, which theoretically would show no detectable current variation (See Section 6.3). A random sequence of instructions could be controlled by the data and control logic impact on current and generated by the PAAR architecture. As a result the PAAR architecture could possibly overcome any power analysis attack without employing any software based countermeasure techniques.

7.2 Summary and Discussions

This thesis developed a high level current measurement-based macro-model for simulation of instantaneous current consumption in a complex parallel issue VLIW DSP processor. Simple
elementary functions called gamma functions were used as atomic functions at the instruction level to approximate the dynamics of the switching activity of the processor core. Other elementary functions could be used as atomic functions. An example of such functions was the set of sample points generated by Equation (4.5) of Section 4.4. The approximating methodology at the instruction level could be generalized to other CMOS type processors and systems that provide capabilities for current consumption measurements at the instruction level. Instruction-level energy models can be integrated with the power-optimization compilers more easily than simulation base estimation since the instruction is the atomic unit used by code generators and no understanding of the underlying hardware is needed [27]. As a result, the current dynamics model, together with a complete data base of maximum current consumptions for all of the instructions within the processor’s instruction set, could be used along with a simulator to provide valuable current dynamics and energy consumption information at the software level. Maximum current measurements of current consumption at the instruction level were obtained by the use of an oscilloscope. However, after the shape parameters used by the gamma functions are determined simple RMS measurements provided by a meter could be used to approximate the instantaneous current consumption at the instruction level.

The thesis presented step-by-step procedures with numerical examples for generating the gamma functions at three working frequencies (100MHz, 200MHz, and 300MHz) and for developing a complete current simulation model for the SC140 processor. Experiments conducted on general type applications (bubble.asm), DSP specific applications (viterbi.asm), and specialized cryptographic applications (powerco.asm, polymul.asm) proved that the current simulated waveforms closely followed the measured waveforms. These applications varied in length, complexity and functionality and formed a good benchmark for the current dynamic model (See Section 5.3). Instructions that cover the entire instruction set architecture were present. Also, complex nested hardware loops and control sequencing were analyzed. The square of the correlation factor between the simulated waveforms and the measured waveforms was over 99% for the short programs (bubble.asm, viterbi.asm, powerco.asm, powercoN.asm, bubvit.asm) and over 93% for the long program (polymul.asm). The average RMS error for the short programs was 1.8351mA, which was less than 1.6% of the average value (116.4304mA) of the individual mean values (measured waveforms) and for the long program the RMS error was 6.5777mA, which was
less than 4.9% of the measured mean value. Current measurements and simulations at the program and instruction level were performed at various frequencies with specific examples. These measurements supported the power frequency relationship expressed by Equation 2.1 of Chapter 2 (See Section 5.2). An experimental regression line for current dependency to frequency was also presented. A static model for the SC140 processor was also introduced and was compared to the dynamic current model. Looping and interrupt based current measurements were also compared and analyzed. Due to the complexity of the SC140 processor, the performance of the static model proved to be poor when compared with the dynamic model developed in this thesis. As presented in this thesis the current dynamics model has a few limitations:

- The model does not currently support dynamic voltage scaling or power management. However, these techniques could be supported by the model if experimental measurements were possible.

- The simulated programs must be traced and modeled separately for each control path. The programs must be divided in linear blocks based on parallelism composition and factorial adjustments at the linear block level must be applied in order to obtain the final current waveform. However, for processors with smaller parallel issue capabilities the linear block division may not be required.

- The model does not dynamically support current variations generated by the changes in the data manipulated by the instruction’s operands. Nevertheless, average measurements were used in order to compensate globally for data variation impact on current consumption. Variations of current due to operand switching at the instruction level of up to 30% on the variable current component were less than 1% on the total current.

- Due to limited published knowledge of the underlying SC140 hardware support of loops and conditional control, the modeling of hardware loop specific instructions (LPMARKB, LPMARKA) and conditional type execution sets (IFT, IFA, IFF) was difficult. A possible solution would be a mixed model that uses lower level power estimates for the hardware loop module.
The experimental results presented in this thesis showed that the current dynamics model is suitable for analysis of energy and power, supports reliability and low power design of devices, and is crucial for the wireless communications industry. The thesis also applied the instantaneous current measurements and modeling of processors to security, which is a growing area of concern. The applications of the current dynamics model can be numerous. The current dynamic model can be used in embedded systems and system-on-chip (SOC) designs by modeling the current waveforms of the software component of an embedded system design or by analyzing worst case current waveforms generated by processor cores. In the designing of a high-performance microprocessor the power distribution network is considered early in the design process [72]. In the designing of the power grid, since the actual circuit implementation is not available, the peak current magnitude is estimated as 5x-8x the average projected current [72]. Using the current dynamics model a good estimate of the peak current can be performed for the software component of the embedded systems, and also a worst case value can be projected for a processor core in SOC designs. These estimated maximum currents can be used to optimize or verify the projected power grid in embedded and SOC designs. The estimated peak currents could also be used in electromigration reliability studies for providing peak values to the dc stress tests [72]. Another possible application of the current dynamics model could also be in evaluating the accuracy of the RTL power and energy models.

The crypto researchers could use the current dynamics model for a first hand study of the resistance of the cryptographic implementations against simple power analysis attacks. The model could assist in implementing cryptographic software that has reduced current consumption variations and possibly eliminate most of the visual current dependencies due to code and algorithm. The software smoothening technique could be used as a countermeasure at the program level against SPA. Simple code transformations based on maximum current measurements at the instruction level could enhance the security of the implementation against SPA that is due to algorithm and code implementations. The hardware smoothening technique could provide theoretical immunity against SPA and DPA since a dynamic current control is performed in real time at the clock speed of the processor. The PAAR architecture can be designed as a plug-in module and can be attached at an existing general type processor. However, the processor must be capable of responding to the current signals generated by PAAR. This can
be done through an external interrupt technique or the processor must be redesigned to support PAAR. This hardware technique is better than other hardware techniques proposed in [68] since it requires little or no change at the processor architecture level and it does not complicate the pipeline design. The PAAR module can also be designed for low-power support by having two states: sleep state and active state. Finally, the EASI technique presented in this thesis provides support for a simple and quick verification tool against SPA in cryptographic designs.

7.3 Future Work

Some important future research directions emerging from this research are:

1. Incorporate the dynamic current model as a CAD tool for support of software-hardware co-design in embedded and SOC designs.

2. Develop a dynamic current model that supports dynamic voltage scaling and power management techniques.

3. Investigate the generality of the modeling methodologies presented in this thesis to other processors.

4. Investigate a hardware implementation of the power attack resistant architecture presented in Section 6.3.

In conclusion, unlike the cycle-by-cycle or average power models, this thesis presented new measured-based modeling techniques and principles for instantaneous current and power consumption in a complex processor core. Besides the energy consumption analysis at the program level the instantaneous current model could be used to analyze current peaks and waveform shapes in security or reliability applications. The only processor dependent components in the simulation macro-model process were the atomic functions and the processor’s hardware architecture. However, the current superposition principles that reflected the SC140 hardware architecture could possibly be transformed to accommodate different hardware architectures and the atomic functions could be mathematically matched to other current shapes, if needed for other CMOS type processors. This model could form a strong base for developing valuable current, power, and energy estimating tools in embedded systems. Also, an important characteristic of the
research presented in this thesis is that the space of the applications of the instruction based models was expanded to security and reliability, thus creating many future research possibilities.
PUBLICATIONS


REFERENCES


Appendix A

EQUIPMENTS

The equipments used to support the current measurement setup for the SC140 DSP processor were:

• **The Fluke 867 Graphical Multimeter.** This meter has many features that are useful in performing low-current measurements [57]. The meter has selectable True-RMS or Average Sensing with a basic measurement accuracy of ± 0.05% to 0.2%. It can measure true-RMS AC with bandwidth to 300 kHz. It also has capability of waveform display from DC to 1 MHz. The meter has a serial communication port and an automated data measurement collection can be done by connecting it to a host system [57].

• **Tektronix TDS 8000 Digital Sampling Oscilloscope.** The TDS 8000 oscilloscope is a very complex and versatile tool [58]. This oscilloscope together with 2 inductive current probes, TCP202 DC current probe, and CT1 an AC current transformer [58] formed the main current dynamics measuring tools. The functionality of the sampling scope and its horizontal synchronization and acquisition setup had a major impact on the methodology used in this thesis to perform the current dynamics measurements. The oscilloscope samples the data, one sample point per trigger event, over repeated trigger events until a full record is stored and then assembled into the displayed waveform.

In Figure 3.3, the waveform record and its defining parameters for this sampling oscilloscope are presented. The instrument acquires points in order from left to right, with each point from a separate trigger event and delayed from that event by Horizontal delay + (Sample interval × (sample number −1)). When all the points in the waveform record have been sampled and digitized, the waveform record is in the acquisition memory and becomes available for display and use in math waveforms, storing, exporting, and elsewhere [58]. These features made it possible to perform complex data analysis on the current measurements obtained.
The typical signal bandwidth for the sampling module used (80E02) is 20 GHz. Other important terms used to define the waveform shown in Figure 3.3 were defined below:

- Time of first point = Horizontal position – (10 divs × Horizontal Scale in sec/div × Horizontal Reference / 100)
- Time Duration (seconds) = 10 divs (window size) × Horizontal Scale (sec/div)
- Time Duration (seconds) = Sample Interval (seconds/sample) × Record Length (samples/sec)
- Sample Interval (sec/sample) = Resolution (sec/sample) = 1 / Sample Rate (samples/sec)
- Maximum Record Length = Time Duration / Min Sample Interval

All the above equates were important in defining the measuring conditions and characteristics for the software application. The Horizontal Reference is indicated in percentage of window size. If its value is 0%, then the Horizontal delay is equal with the Horizontal Position. The elements controlled by the user are the external trigger, the horizontal position, the horizontal scale, and the record
length. All other elements are automatically calculated based on these setups. The horizontal scale is from 1ps to 5ms/div and the farthest reachable point is 50ms away from the triggering signal. As a result, direct measurements could be performed on software applications that had an execution time less than 50ms. In order to perform current, power and energy analysis on programs that had execution times longer than 50ms, the program applications had to be divided in independent executable blocks that executed for less than 50ms.

- **The Current Probes.** **CT1** is a 1GHz AC inductive current probe that can be installed temporarily or permanently in the device-under-test (DUT) [58]. The probe measures current through a conductor by developing a voltage proportional to the current. The sensitivity of the probe is $5mV / 1mA$, ±3%. The presence of DC current affects the performance of the probe [58]. **TCP202** is an AC/DC current probe that measures AC or DC current in a single conductor by induction. This current probe is suitable to measure instantaneous low current or low power that has a DC component and an AC component of up to 50MHz. In the current experiments performed, the processor ran at frequencies up to 300MHz, but the real current variation was much lower. The DC accuracy of the probe is ±3% and the noise increases over 20MHz. The advantage of the TCP202 probe was that it gave an unaltered instantaneous current measurement that included the AC and DC component.

- **Hewlett Packard 81130A.** The HP 81130A is a pulse/pattern generator with working frequencies of up to 660MHz [59]. The instrument can generate a large class of signals and signal patterns. Its most important feature which made the current dynamics method possible was that it can be used to generate long pattern signals of periods longer than 50ms. It also has a very stable frequency and as a result, it could provide good triggering signals to the sampling scope.
Appendix B

CURRENT MEASUREMENTS AND PROGRAMS

B.1 Current Measurement Table at the Instruction Level

Table B.1 summarizes the AC maximum values (ACmax) obtained by using the representative waveforms, and the average values (Iavg) obtained by using the meter measurements. It presents the current measurements for all of the instruction types encountered in the assembly language programs analyzed in this thesis: powerco.asm, powercoN.asm, bubble.asm, viterbi.asm, and bubvit.asm (See Section B.2). The measurements were performed for instructions that operated on random/mixed data values. Separate measurements were taken when the operands were present in different registers. These measurements were averaged and the values obtained were the ones reported in Table B.1.

<table>
<thead>
<tr>
<th>Instruction name</th>
<th>Iavg [mA]</th>
<th>AC Max [mA] =Imax –NOP level</th>
<th>Number of Cycles</th>
</tr>
</thead>
<tbody>
<tr>
<td>#</td>
<td>Meter Measurement</td>
<td>Scope Measurement</td>
<td></td>
</tr>
<tr>
<td>0  WAIT</td>
<td>20.75</td>
<td>--</td>
<td>--</td>
</tr>
<tr>
<td>1  2 WORD PREFIX ACTIVITY</td>
<td>-</td>
<td>0.1795</td>
<td>--</td>
</tr>
<tr>
<td>2  NOP</td>
<td>65.57</td>
<td>0.0000</td>
<td>1</td>
</tr>
<tr>
<td>3  DOENn #u6</td>
<td>68.63</td>
<td>0.1175</td>
<td>1</td>
</tr>
<tr>
<td>4  DOENn DR</td>
<td>82.57</td>
<td>0.3076</td>
<td>1</td>
</tr>
<tr>
<td>5  DOENSHn #u6</td>
<td>69.82</td>
<td>0.1482</td>
<td>1</td>
</tr>
<tr>
<td>6  DOSETUPn label</td>
<td>78.21</td>
<td>0.3050</td>
<td>1</td>
</tr>
<tr>
<td>7  MOVE.l #u16,Rn</td>
<td>87.89</td>
<td>0.3221</td>
<td>1</td>
</tr>
<tr>
<td>8  MOVE.l #u16,Bn</td>
<td>89.82</td>
<td>0.3124</td>
<td>1</td>
</tr>
<tr>
<td>9  MOVE.l #u32,MCTL</td>
<td>98.54</td>
<td>0.3396</td>
<td>1</td>
</tr>
<tr>
<td>10 MOVE.l Mn,Dn</td>
<td>73.34</td>
<td>0.2031</td>
<td>1</td>
</tr>
<tr>
<td>11 MOVE.l Dn,Rn</td>
<td>78.25</td>
<td>0.2941</td>
<td>1</td>
</tr>
<tr>
<td>12 MOVE.2l (Rn)+,Da:Db high bank</td>
<td>107.67</td>
<td>0.6385</td>
<td>1</td>
</tr>
<tr>
<td>MOVE.2l (Rn)+,Da:Db</td>
<td>107.67</td>
<td>0.8315</td>
<td>1</td>
</tr>
<tr>
<td>No.</td>
<td>Instruction</td>
<td>Time (ns)</td>
<td>Utilization (µs)</td>
</tr>
<tr>
<td>-----</td>
<td>------------------------------------------</td>
<td>-----------</td>
<td>------------------</td>
</tr>
<tr>
<td>13</td>
<td>MOVE.2l Da:Db,(Rn)</td>
<td>118.95</td>
<td>0.7311</td>
</tr>
<tr>
<td>14</td>
<td>MOVE.2l Da:Db,(Rn)+</td>
<td>118.73</td>
<td>0.7847 (from pair)</td>
</tr>
<tr>
<td>15</td>
<td>MOVE.1 (Rn),Da</td>
<td>81.50</td>
<td>0.5050</td>
</tr>
<tr>
<td>16</td>
<td>MOVE.2l (Rn)+Nx,Db</td>
<td>-</td>
<td>0.6658</td>
</tr>
<tr>
<td>17</td>
<td>MOVE.2l (Rn)+Nx,Db,Db</td>
<td>111.46</td>
<td>0.8453 (from pair)</td>
</tr>
<tr>
<td>18</td>
<td>CLR Dn</td>
<td>83.95</td>
<td>0.2709</td>
</tr>
<tr>
<td>19</td>
<td>[POP R6 POP R7]</td>
<td>106.52</td>
<td></td>
</tr>
<tr>
<td>20</td>
<td>ADDA #s16,rx,Rn</td>
<td>72.39</td>
<td>0.2682</td>
</tr>
<tr>
<td>21</td>
<td>IFA ADDA #u5,Rn</td>
<td>72.63</td>
<td></td>
</tr>
<tr>
<td>22</td>
<td>ADDA #u5,Rn</td>
<td>70.57</td>
<td>0.1419</td>
</tr>
<tr>
<td>23</td>
<td>TFR Da,Dn</td>
<td>68.75</td>
<td>0.1651</td>
</tr>
<tr>
<td>24</td>
<td>TFR Da,Dn</td>
<td>78.68</td>
<td>0.2139</td>
</tr>
<tr>
<td>25</td>
<td>TFR Da,Dn</td>
<td>99.40</td>
<td>0.3865</td>
</tr>
<tr>
<td>26</td>
<td>MOVE.2l Da:Db,Dn</td>
<td>74.81</td>
<td>0.1839</td>
</tr>
<tr>
<td>27</td>
<td>MOVE.2l Da:Db,Dn</td>
<td>77.31</td>
<td>0.2858</td>
</tr>
<tr>
<td>28</td>
<td>MOVE.2l Da:Db,Dn</td>
<td>73.59</td>
<td>0.2364</td>
</tr>
<tr>
<td>29</td>
<td>ADD Da,Db,Dn</td>
<td>104.80</td>
<td>0.5153</td>
</tr>
<tr>
<td>30</td>
<td>SUB Da,Db,Dn</td>
<td>74.57</td>
<td>0.1469</td>
</tr>
<tr>
<td>31</td>
<td>SUB Da,Db,Dn</td>
<td>87.24</td>
<td>0.2651</td>
</tr>
<tr>
<td>32</td>
<td>ADD Da,Db,Dn</td>
<td>106.90</td>
<td>0.4364</td>
</tr>
<tr>
<td>33</td>
<td>MOVE.2l Da:Db,(Rn)</td>
<td>98.53</td>
<td>0.4464</td>
</tr>
<tr>
<td>34</td>
<td>MOVE.2l Da:Db,(Rn)+</td>
<td>100.69</td>
<td>0.4464</td>
</tr>
<tr>
<td>35</td>
<td>TSTGT Dn</td>
<td>76.38</td>
<td>0.2751</td>
</tr>
<tr>
<td>36</td>
<td>BT label: T=1</td>
<td>116.16</td>
<td>0.9258</td>
</tr>
<tr>
<td>37</td>
<td>BT label: T=0</td>
<td>77.49</td>
<td>0.2543</td>
</tr>
<tr>
<td>38</td>
<td>JMP label</td>
<td>111.57</td>
<td>0.9229</td>
</tr>
<tr>
<td>39</td>
<td>LPMARK</td>
<td>111.57</td>
<td>0.8427</td>
</tr>
<tr>
<td>40</td>
<td>CMPGT Da,Dn</td>
<td>84.95</td>
<td>0.2265</td>
</tr>
<tr>
<td>41</td>
<td>IFT MOVE.1 Da,(Rn): T=1</td>
<td>94.09</td>
<td>0.6615</td>
</tr>
<tr>
<td>42</td>
<td>IFT MOVE.1 Da,(Rn): T=0</td>
<td>70.48</td>
<td>0.2716</td>
</tr>
<tr>
<td>43</td>
<td>[ IFA ADDA #&lt;4,R0</td>
<td>110.33</td>
<td>0.8698</td>
</tr>
</tbody>
</table>

173
<table>
<thead>
<tr>
<th></th>
<th>Instruction</th>
<th>AC Current</th>
<th>RMS Current</th>
<th>Duration</th>
</tr>
</thead>
<tbody>
<tr>
<td>44</td>
<td>[ IFA ADDA #&lt;4,R0 IFT MOVE.1 D5,(R1) ]: T=0</td>
<td>92.24</td>
<td>0.4872</td>
<td>1</td>
</tr>
<tr>
<td>45</td>
<td>INSERT #u6,#u6,Db,Dn</td>
<td>92.36</td>
<td>0.2992</td>
<td>1</td>
</tr>
<tr>
<td></td>
<td>high bank</td>
<td>-</td>
<td>0.4852</td>
<td>1</td>
</tr>
<tr>
<td>46</td>
<td>NEG Dn</td>
<td>97.43</td>
<td>0.2027</td>
<td>1</td>
</tr>
<tr>
<td></td>
<td>high bank register</td>
<td>97.43</td>
<td>0.3822</td>
<td>1</td>
</tr>
<tr>
<td>47</td>
<td>EOR Da,Db</td>
<td>81.50</td>
<td>0.2412</td>
<td>1</td>
</tr>
<tr>
<td></td>
<td>high bank</td>
<td></td>
<td>0.4128</td>
<td></td>
</tr>
<tr>
<td>48</td>
<td>[EOR Da,Dn EOR Db,Dn]</td>
<td>126.81</td>
<td>0.3832</td>
<td>1</td>
</tr>
<tr>
<td>49</td>
<td>ADD2 Da,Dn</td>
<td>84.28</td>
<td>0.3099</td>
<td>1</td>
</tr>
<tr>
<td>50</td>
<td>SUB2 Da,Dn</td>
<td>83.99</td>
<td>0.3059</td>
<td>1</td>
</tr>
<tr>
<td>51</td>
<td>ADD2 Da,Dn</td>
<td>105.80</td>
<td>0.4136</td>
<td>1</td>
</tr>
<tr>
<td>52</td>
<td>SUB2 Da,Dn</td>
<td>103.11</td>
<td>0.4208</td>
<td>1</td>
</tr>
<tr>
<td>53</td>
<td>SUBA Ra,Rn</td>
<td>74.85</td>
<td>0.1986</td>
<td>1</td>
</tr>
<tr>
<td>54</td>
<td>MAX2VIT D4,D2</td>
<td>86.76</td>
<td>0.3538</td>
<td>1</td>
</tr>
<tr>
<td>55</td>
<td>MAX2VIT D0,D6</td>
<td>86.21</td>
<td>0.3444</td>
<td>1</td>
</tr>
<tr>
<td>56</td>
<td>MAX2VIT D12,D10</td>
<td>101.91</td>
<td>0.5067</td>
<td>1</td>
</tr>
<tr>
<td>57</td>
<td>MAX2VIT D8,D14</td>
<td>110.23</td>
<td>0.5313</td>
<td>1</td>
</tr>
<tr>
<td>58</td>
<td>VSL.4w D2:D6:D1:D3,(Rn)+N0</td>
<td>111.24</td>
<td>0.7712</td>
<td>2</td>
</tr>
<tr>
<td>59</td>
<td>VSL.4f D2:D6:D1:D3,(Rn)+N0</td>
<td>111.63</td>
<td>0.7526</td>
<td>2</td>
</tr>
<tr>
<td>60</td>
<td>VSL.4w D10:D14:D9:D11,(Rn)+N0</td>
<td>120.72</td>
<td>1.0610</td>
<td>2</td>
</tr>
<tr>
<td>61</td>
<td>VSL.4f D10:D14:D9:D11,(Rn)+N0</td>
<td>120.95</td>
<td>1.0320</td>
<td>2</td>
</tr>
<tr>
<td>62</td>
<td>ADD Da,Db,Dn</td>
<td>104.77</td>
<td>0.4124</td>
<td>1</td>
</tr>
</tbody>
</table>

Table B.1. Instruction types. Maximum AC current values measured with the oscilloscope and RMS values measured with the Fluke meter.

B.2 Assembly Language Programs

This section presents the SC140 assembly language listing for the following programs: bubble.asm, viterbi.asm, powerco.asm, and polymul.asm. The listing also indicates the linear block delimitation for these programs as described in Section 5.3 of Chapter 5.

B.2.1 BUBBLE.ASM

```assembly
; section .data local
org p+$4000
...```
align 8
F__MemAllocArea
Dinit_bubble_list
del 44,26,95,4,8,88 ; offset = 0
del 96,34,7,35
ds 8 ; gap
Dsoft_stack
ds 4 ; offset = 48
ds 4 ; gap
Dexternal_aliased
ds 4 ; offset = 56
align 4
; endsec
; section .text local
org p:$6000
TextStart_bubble
bb_cs_offset__bubble equ 0 ; At _bubble sp = 0
bb_cs_offset_DW_2 equ 2 ; At DW_2 sp = 2
bb_cs_offset_DW_4 equ 22 ; At DW_4 sp = 22
bb_cs_offset_DW_33 equ 2 ; At DW_33 sp = 2
bb_cs_offset_DW_34 equ 0 ; At DW_34 sp = 0
;=========================================================
; Function _bubble
;
; Calling Convention: Standard
;
;Stack frame size: 80
;
;=========================================================
; global _bubble
;_bubble type func

; BEGIN B1
PUM:
_bubble:
push r6 push r7 ;Bub1
DW_2
adda #>80,sp,r6 ;Bub2
tfra r6,sp ;Bub3
DW_4
adda #>80,sp,r1 ;Bub4
move.w #<40,d0 ;Bub5
move.l #Dinit_bubble_list,r0 ;Bub6
nop ; Pointer update delay
L12
sub #1,d0 move.b (r0)+,d1 ;Bub8
tsgt d0 move.b d1,(r1)+ ;Bub9
br L12 ;Bub10

; END B1
; BEGIN B2
SUB:
doen2 #<0 ;Bub11
dosetup2 L11 ;Bub12
dosetup3 L10 ;Bub13
adda #>76,sp,r0 ;Bub14
adda #>80,sp,r1 ;Bub15
move.w #<9,d2 ;Bub16
move.w #<36,d0 ;Bub17
move.w #<36,d1 ;Bub18
loopstart2
L11
tfr d2,d3 ;Bub19
doen3 d3 ;Bub20
falign
loopstart3

}
B.2.2 VITERBI.ASM

;*******************************************************************************************************
;      Viterbi decoding for a convolution encoder.
;      The encoder input is binary.
;      The encoder output is a 2 bit symbol, generated according to the polynomials:
;          a. G0(D) = 1 + D^3 + D^4
;          b. G1(D) = 1 + D + D^3 + D^4
;      The signal corresponding to symbol '00' is antipodal to the signal of '11'.
;      The signal corresponding to symbol '01' is antipodal to the signal of '10'.
;      The signal corresponding to '00' is orthogonal to the signal of '01'.
;      The block length decoded - 12.
;*******************************************************************************************************

;Constants Used:
;    NO_OF_STATES : 16    Number of trellis states.
;    V             : 12    Number of viterbi stages.
;    PM_INIT      : $0000  "-32768"
;    ZERO_INIT    : 1000   "31768"
;    INPUT        : $4000  Matched filter output.
;    PM_RAM_TRELLIS_RAM : $6000  viterbi memory - 2*(NO_OF_STATES+NO_OF_TRELLIS)*2
;    OUTPUT       : $6000

;*******************************************************************************************************
; Cycle count:
; initialize 16+NO_OF_STATES cycles
; kernel 5+V*10 cycles
;
; Input/Output location:
; Input initial address: INPUT
; Output initial address: OUTPUT
;
; registers used:
; d0-d15,r0,r1,r2,r3,r4,r5,h2,b3,b4,b5,n0,n1,n3,m0,m2
; r0,r1 point to the input signal.
; r2,r3,r4,r5 point to the viterbi memory
;
; assumptions:
; All registers used could be written to.
; V must be smaller than 12.
; the viterbi memory (PM_RAM_TRELLIS_RAM) should be 8-word aligned.
;
******************************************************************************************
NO_OF_STATES equ 16
V equ 12
PM_INIT equ $0000
ZERO_INIT equ 1000
INPUT equ $4000
PM_RAM_TRELLIS_RAM equ $6000
OUTPUT equ $6000
org p:INPUT
dc -84,-65,107,-67,79,55,71,47
dc -116,55,-61,-64,-58,27,70,68
dc 78,-66,86,-64,-66,46,-46,-27
dc 27,-126,-62,-64,92,70,-70,66
org p:$3000
; BEGIN B3

PUM:
init:
Vit1: move.l #INPUT,r0
Vit2: move.l #INPUT+2,r1
Vit3: move.l #PM_RAM_TRELLIS_RAM,r2
Vit4: move.l #PM_RAM_TRELLIS_RAM+32,r3
Vit5: move.l #PM_RAM_TRELLIS_RAM+64,r4
Vit6: adda #8,r4,r5
Vit7: move.w #$0000,m2
Vit8: doensh0 #NO_OF_STATES-1
Vit10:_clr d1
Vit11: insert #16,#16,d0,d0
Vit12: move.2l d0,d1,(r2)+
Vit13: move.l #PM_RAM_TRELLIS_RAM,r2
Vit14: move.w #$0000,m2
Vit15: move.l #PM_RAM_TRELLIS_RAM,b2
Vit16: move.l #PM_RAM_TRELLIS_RAM,b3
Vit17: move.l #PM_RAM_TRELLIS_RAM,b4

177
Vit18:       move.l #PM_RAM_TRELLIS_RAM,b5
Vit19:       move.l #$00888800,MCTL
;
kernel:
;
; Butterfly Initialization
Vit20:       dosetup0 _main_loop   doen0 #V
Vit21:       move.w (r0)+n1,d15 move.w (r1)+n1,d13 ; Get soft decisions for
             ; first stage.
Vit22:       add d15,d13,d7 sub d13,d15,d13 ; Calculate BM for first stage.
clrd5
;
Vit23:       insert #16,#16,d13,d7 neg d13
sub d7,d5,d5
;
Vit24:       insert #16,#16,d13,d5
;
; Viterbi kernel

_kernel

_loopstart0

loopstart0
Vit25:       tfr d7,d4 tfr d7,d6
             insert #16,#16,d13,d5
             move.2l (r2)+,d0:d1 move.2l (r3)+,d2:d3 ; load path metrics and
             ; trellis for ACS 1 and 2.
             ; ACS 1 and 2 :
Vit26:       add d0,d6,d2 sub2 d6,d2
             add d2,d6,d6 ; path metric calculation(1,2)
             move.w (r0)+n1,d15 move.w (r1)+n1,d13 ; load path metrics and
             ; trellis for ACS 3 and 4.
             ; ACS 3 and 4 :
Vit27:       max2vit d4,d2 max2vit d0,d6 ; for each state choose the one
             ; with the bigger value(1,2).
             tfr d7,d12 tfr d7,d14
             move.2l (r2)+,d8,d9 move.2l (r3)+,d10,d11
             ; ACS 3 and 4 :
Vit28:       add d2,d12,d10 sub2 d14,d10 ; path metric calculation(3,4)
             add d2,d14,d14 ; path metric calculation(3,4)
             vsl.4w d2:d6:d1:d3,(r4)+n0 vsl.4f d2:d6:d1:d3,(r5)+n0 ; update path metric and
             ; trellis for ACS 1 and 2.
             ; ACS 5 and 6 :
Vit29:       max2vit d12,d10 max2vit d8,d14 ; for each state choose the one
             ; with the bigger value(3,4).
             tfr d5,d4 tfr d5,d6
             move.2l (r2)+,d0,d1 move.2l (r3)+,d2,d3 ; load path metrics and
             ; trellis for ACS 5 and 6.
             ; ACS 5 and 6 :
Vit30:       add d2,d12,d12 sub2 d6,d2 ; path metric calculation(5,6)
             add d2,d6,d6 ; path metric calculation(5,6)
             vsl.4w d10:d14:d9:d11,(r4)+n0 vsl.4f d10:d14:d9:d11,(r5)+n0 ; update path metric and
             ; trellis for ACS 3 and 4.
             ; ACS 7 and 8 :
Vit31:       max2vit d4,d2 max2vit d0,d6 ; for each state choose the one
             ; with the bigger value(5,6).
             tfr d5,d12 tfr d5,d14
             move.2l (r2)+n3,d8,d9 move.2l (r3)+n3,d10,d11 ; load path metrics and
             ; trellis for ACS 7 and 8.
             ; ACS 7 and 8 :
Vit32:       add d2,d12,d12 sub2 d14,d10 ; path metric calculation(7,8)
             add d2,d10,d14 ; path metric calculation(7,8)
             vsl.4w d2:d6:d1:d3,(r4)+n0 vsl.4f d2:d6:d1:d3,(r5)+n0 ; update path metric and
             ; trellis for ACS 5 and 6.
             ; ACS 7 and 8 :
Vit33:       max2vit d12,d10 max2vit d8,d14 ; for each state choose the one
             ; with the bigger value(7,8).
             add d13,d13,d7 sub d13,d15,d13 ; prepare soft decisions for
             ; next stage.
Vit34:       insert #16,#16,d13,d7 sub d7,d5,d5
neg d13

178
B.2.3 \textsc{powerco.asm}

\begin{verbatim}
\input equ $6000
\input1 equ $6028
\input2 equ $6050
\input3 equ $6078
\input4 equ $60a0
\input5 equ $60c8
\input6 equ $60f0
\input7 equ $6218
\output equ $6400
\output1 equ $6428
\output2 equ $6450
\output3 equ $6478
\output4 equ $64a0
\output5 equ $64c8
\output6 equ $64f0
\output7 equ $6518

org p:$INPUT
\dc $DFB5,$C31B,$1940,$88F5,$A0FD,$DFF9,$2942,$89A7
\dc $2C8,$4AE3,$BB77,$DDC5,$7D67,$29AB,$8F6E,$7F77
\dc $FB68,$A211,$535F,$1C10,$D187,$62B0,$1D55,$8DD9
\dc $3393,$2977,$7010,$F999,$48BD,$5224,$9140,$A9C7
\dc $4612,$F756,$A782,$BB2A,$46E2,$88DD,$F2A7,$5BA0
\dc $9E5B,$4E37,$912D,$C8A1,$562E,$7E34,$B2E3,$8296
\dc $9B08,$2193,$9750,$8D32,$8D88,$9CE1,$619C,$9D3F
\dc $2B7F,$E7FF,$5C8D,$F13A,$857A,$695C,$89DA,$1FF4
\dc $5039,$83D6,$3C33,$E1E1,$A57A,$7D0D,$8065,$355D
\dc $9C9D,$7DBF,$B091,$9C5A,$D3F1,$4088,$B175,$609D
\dc $1F85,$2D11,$6D0A,$E472,$9E97,$4B3F,$49F1,$33EA
\dc $9967,$D075,$CDF8,$9DF9,$F5CD,$46A5,$1F62,$293E
\dc $F7C7,$0F63,$3A70,$1F58,$A80F,$6F59,$8B40,$A516
\dc $9A4C,$2143,$7312,$272F,$D71D,$3C43,$3E1A
\dc $80F7,$C3CA,$C9A4,$4ABE,$D4B3,$9F8A,$C653,$6DB1
\dc $A6D5,$F188,$C2FF,$8BDB,$B53B,$3E03,$2587,$B975
\dc $23B4,$6BFD,$F6BB,$95A3,$21B2,$94B2,$7B28,$9AE5
\dc $E7FD,$5B52,$A2A0,$9B07,$981C,$2B2A,$D605,$5FB9
\dc $D668,$F4D0,$21B2,$9357,$BB73,$89F6,$C8D9,$D290
\dc $93FA,$8678,$D705,$8FE1,$747D,$A42C,$9F08,$F780
\dc $A4CC,$89A1,$4C15,$20C5,$5A94,$8CA6,$2FC5,$5641
\dc $9D85,$4E9D,$E96B,$B0E4,$B55E,$B0A0,$4702,$433B
\dc $E8AF,$4FB8,$458F,$63F7,$4E35,$A0A1,$1882,$64C6
\dc $0CA0,$2296,$D6D9,$A177,$60BF,$EC23,$DCC3,$3299
\dc $8201,$160A,$2B11,$15EC,$6C55,$5F69,$0B1F,$3B0D
\dc $D149,$920D,$B44E,$8F0C,$D77E,$4288,$4AE2,$6D68
\dc $81F2,$1E97,$F35A,$F194,$554F,$83C7A,$FCA4,$8B18
\dc $1A72,$22B0,$6E8B,$E9D7,$E8D0,$B297,$9D3D,$02AF
\dc $F5F3,$477F,$7F2A,$A3FD,$7512,$E337,$B930,$4AD6
\dc $8A92,$E43E,$8052,$7TB4,$B2EE,$B521,$C944,$E847
\dc $3CFA,$2823,$34C1,$154E,$01D0,$9C40,$60D3,$5768
\dc $3A3D,$BBF7,$88A9,$8A54,$8E68,$2B84,$3E63,$68B2
\dc $FDB1,$F1FE,$A7CF,$A55F,$F5FD,$7793,$FED7,$A3CC
\dc $0709,$BCFC,$3FB2,$8AB7,$8A55,$06BA,$6A89,$80F0
\dc $81B0,$B9C9,$FF45,$8D14,$33C9,$6232,$621F,$98D4

org p:$4000
\end{verbatim}
; BEGIN B4

PUM:
init:
align 16

Eco1: [ 
  doen3 #<3 ;[40] 
  dosetup3 L5 ;[40] 
] 
Eco2: move.l #OUTPUT,r1 ;[40] 
Eco3: move.l #INPUT+24,r2 ;[40] 
Eco4: move.l #INPUT,r3 ;[40] 
;| 
;| Block 1, Pred F0,LB1, Succ LB1,F2 
;| Freq 3, Nesting 1, VISITED 
;| Lc: n3,r0,r1,r2,r3 
;| 
  falign 
loopstart3 
L5 
Eco5: [ 
  move.2l (r2)+,d0:d1 ;[40] 
  move.2l (r3)+,d2:d3 ;[40] 
] 
Eco6: eor d2,d0 eor d3,d1 ;[40] 
Eco7: move.2l d0:d1,(r1) ;[41] 
loopend3 
;| 
;| Block 1, Pred F0,LB1, Succ LB1,F2 
;| Freq 3, Nesting 1, VISITED 
;| Lc: n3,r0,r1,r2,r3 
;| 
Eco8: clr d0 ;[47] 
Eco9: [ 
  doen3 #<3 ;[40] 
  dosetup3 L501 ;[40] 
] 
Eco10: move.l #OUTPUT1,r1 ;[40] 
Eco11: move.l #INPUT1+24,r2 ;[40] 
Eco12: move.l #INPUT1,r3 ;[40] 
;| 
;| Block 2, Pred F1, Succ R:@NONE@ 
;| Freq 1, Nesting 0, VISITED 
;| Lc: d0 
;| 
;| Block 2, Pred F1, Succ R:@NONE@ 
;| Freq 1, Nesting 0, VISITED 
;| Lc: d0 
;| 
Eco13: clr d0 ;[47] 
Eco14: [ 
  doen3 #<3 ;[40] 
  dosetup3 L502 ;[40] 
] 
Eco15: move.l #OUTPUT2,r1 ;[40] 
Eco16: move.l #INPUT2+24,r2 ;[40] 
Eco17: move.l #INPUT2,r3 ;[40]
nopr  ; added by the compiler for alignment

181

Block 1, Pred F0, Succ LB1, F2
Freq 3, Nesting 1, VISITED
Lx: n3, r0, r1, r2, r3

falign
loopstart3
L502
[
  move.2l  (r2)+,d0:d1          ;[40]
  move.2l  (r3)+,d2:d3          ;[40]
  eor  d2,d0  eor  d3,d1          ;[40]
  move.2l  d0:d1,r1              ;[41]
  loopend3
]

Block 2, Pred F1, Succ R: @NONE@
Freq 1, Nesting 0, VISITED
Lx: d0

Eco25:  clr  d0                  ;[47]
[
  doen3  #<3                  ;[40]
  dosetup3 L503               ;[40]
  move.l  #OUTPUT3,r1         ;[40]
  move.l  #INPUT3+24,r2        ;[40]
  move.l  #INPUT3,r3          ;[40]
]

Block 1, Pred F0, Succ LB1, F2
Freq 3, Nesting 1, VISITED
Lx: n3, r0, r1, r2, r3

falign
loopstart3
L503
[
  move.2l  (r2)+,d0:d1          ;[40]
  move.2l  (r3)+,d2:d3          ;[40]
  eor  d2,d0  eor  d3,d1          ;[40]
  move.2l  d0:d1,r1              ;[41]
  loopend3
]

Block 2, Pred F1, Succ R: @NONE@
Freq 1, Nesting 0, VISITED
Lx: d0

clr  d0                  ;[47]
[
  doen3  #<3                  ;[40]
  dosetup3 L504               ;[40]
  move.l  #OUTPUT4,r1         ;[40]
  move.l  #INPUT4+24,r2        ;[40]
  move.l  #INPUT4,r3          ;[40]
]

Block 1, Pred F0, Succ LB1, F2
Freq 3, Nesting 1, VISITED
Lx: n3, r0, r1, r2, r3

falign
loopstart3
L504
[
  move.2l  (r2)+,d0:d1          ;[40]
  move.2l  (r3)+,d2:d3          ;[40]
]
doen3 #<3 ;[40]
dosetup3 L507 ;[40]
|
| Block 1, Pred F0,LB1, Succ LB1,F2
| Freq 3, Nesting 1, VISITED
|
| Lx: n3,r0,r1,r2,r3
|
| faalign
loopstart3 L507
|
| move.2l (r2)+,d0:d1 ;[40]
move.2l (r3)+,d2:d3 ;[40]
|
| eor d2,d0 eor d3,d1 ;[40]
move.2l d0:d1,(r1) ;[41]
loopend3
|
| Block 2, Pred F1, Succ R:@NONE@
| Freq 1, Nesting 0, VISITED
|
| Lx: d0
|
| clr d0 ;[47]
out
_dhalt:
|
; END B4

B.2.4 POLYMUL.ASM

; Binary Field Elliptic Multiplication Routine - file: NIST_polyelipticmul.asm
; Revision History: 01/08/29
; (Uses NIST values for px and py)
; Cycles: 3,190.280
; Inputs:
;  p.x: Address = $3000 Value = 3 f0eba162 86a2d57e a0991168 d4994637 e8343e36
;  p.y: Address = $3020 Value = 0 d51fbc6c 71a0094f a2cdd545 b11c5c0c 797324f1
;  a2: Address = $3060 Value = 0 0 0 0 0 1
;  a6: Address = $3080 Value = 2 0a601907 b8c953ca 1481eb10 512f7874 4a3205fd
;  k : Address = $3040 Value = 5 63323eab 10fc68f8 254d4d11 d2d518f2 9979dd24
; Outputs:
;  r.x: Address = $4ec8 Value = 4 b275d168 09023195 f0e09eab e8ddee0b f2e73d48
;  r.y: Address = $4ee0 Value = 4 c2144555 ee954b66 d8739c9c 940307e2 6ec3a365
; Notes: This program, when run multiple times will generate the correct
; output values but it is necessary to run it to the end of
; polyelipticmul function (denoted by label "finish") in order
; for the correct output values to be displayed (the output
; addresses are used for intermediate processing.)
; Also, the program structure closely resembles the code in
; polyelipticmul_verify (where the polyelipticmul function code is
; embedded within the main function of the program.)
; Addresses $3000-5000 used to store input values. Program set
; to start at address $6000.
; The k value can be changed. See web (http://optimal.vlsi/~asathian) for link
; to file for precomputed values for constant NIST x and y but varying k (shows
; required output for different values of k)

px  equ $3000
py  equ $3020
k  equ $3040
a2  equ $3060
a6  equ $3080

183
org p:sqtab

org p:px

org p:py

org p:a6

org p:a2

org p:k

org p:z1

org p:switch1

org p:z2

org p:px

org p:py

org p:a6

org p:a2

org p:k

org p:z1

org p:switch1

org p:z2
; dew 0,0,0,0,0,0,0,0,0,0,6; 6={1,0,-1,0} should have copy,sub,c
; dew 0,0,0,0,0,0,0,0,0,0,7; 7={1,0,0,-1} should have copy,c,sub
; dew 0,0,0,0,0,0,0,0,0,0,8; 8={1,0,0,0} should have copy,c,c
; dew 0,0,0,0,0,0,0,0,0,0,9; 9={1,0,0,1} should have copy,c,add
; dew 0,0,0,0,0,0,0,0,0,0,$a; 10={1,0,1,0} should have copy,add,c
; dew 0,0,0,0,0,0,0,0,0,0,$b; 11={1,0,-1,0}-1 should have copy,sub,c,sub
; dew 0,0,0,0,0,0,0,0,0,0,$10; 11={1,0,0,0}
; dew 0,0,0,0,0,0,0,0,0,0,$15; 21={1,0,1,0,1} should have copy,esum,sub,c,sub
; dew 0,0,0,0,0,0,0,0,0,0,$14; 20={1,0,1,0,0} should have copy,esum,add,c,c
; dew 0,0,0,0,0,0,0,0,0,0,$19; 25={1,0,1,0,1,1} should have copy,sub,c,esum
; dew 0,0,0,0,0,0,0,0,0,0,$18; 27={1,0,0,1,0,1}
; dew 0,0,0,0,0,0,0,0,0,0,$30; 56={1,0,0,0,0,0,0}
; dew 0,0,0,0,0,0,0,0,0,0,$1d; 29={1,0,0,-1,0,1} should have copy,esum,sub,c,esum
; dew 0,0,0,0,0,0,0,0,0,0,$1b; 27={1,0,0,-1,0,-1}
; dew 0,0,0,0,0,0,0,0,0,0,$12; 18={1,0,0,1,0,1}
; dew 0,0,0,0,0,0,0,0,0,0,$25; 37={1,0,0,0,1,0} should have copy,esum,esum,c
; org p:Dswitch1
; del 1,89,190,1,88 ; offset = 1872
; org p:0
; jmp $6000

REGD equ $440
org p:REGD
dcl $00000007,$0e781b3f,$fcaef2ee,$7a11afb7
dcl $038e9e4b,$2ccb9912,$f95de5dc,$00000001
del $0599732ed,$00000007,$0e781b3f
dcl $fcaef2ee,$7a11afb7,$038e9e4b,$2ccb9912
REGR:
dcl $5030,$5078,$5018,$5030,$4098,$4150,$4184,$5048
dcl $4e98,$50e8,$50ec,$50f0,$5064,$5068,$5070
polymulSPzone equ $4e60
org p:polymulSPzone
dcl $e0,$00e40001,0,0 ;4e60
del 0,0,0,31
dcl $2,$0a601907,$8e953ea,$1481eb10
dcl $5127874,$4a3205fd,$03,$0e90a162
cl $86a2d57e,$0991618,$d4994637,$e8343e36
dcl $903,$25f11dc,$c702dc31,$0254e42d
dcl $68581a3b,$01471ac7,$02,$0e579937
del $f964ab11,$6b546d13,$d7ab1a87,$0645dfb6
dcl $0f4,$714eb27b6,$c886d726,$e870db7
dcl $2b5f1412,$e8579c3b,$03,$0a58a9ae
dcl $5e849291,$e9861e51,$f4e83d55,$06a737f7
dcl $09,$1be9316,$e8f7f57,$8df5f885
dcl $14b4e9f2,$27c0e011,$b3,080e1b62
cl $86a2d57e,$0991618,$d4994637,$e8343e36
dcl $0d,$51f1b0e,$71a0094f,$1c2dd545
dcl $b11e5e0c,$797324f0,0
cl 0,0,0,0
cl $03,$8e343e36,$8d51f1b0e
cl $71a0094f,$1c2dd545,$b11e5e0c,$797324f1
cl 0,0,0,0
cl $01,$02,$0a601907
cl $b8e953ea,$1481eb10,$5127874,$4a3205fd
cl 0,0,0,0
cl $0e,$00e40000,$4e6,$4e68
cl $06b6,$0e0000,$035a,$4e68
cl $635c,$4e6b,$07,$792db2
cl $d6625e6,$33e2d6b,$8879d415c,$090c6937
cl $07,$e8781b3f,$fcaef2ee,$7a11afb7
cl $038e9e4b,$2ccb9912,$01,$7dc68407
cl $18b35c4,$b90f34be,$71a0094f,$1152590d
cl $8000289,$e6150df7,$88025400,$eb50dec5
cl $81263be1,$81ac3d6f,$54b0962e,$4dd287b5
Function poly_div_new

poly_div_new

Epd1: [ push     d6                         ;\[299\] push     d7                         ;\[299\] ]

Epd2: [ move.l   (r0),d0                    ;\[309\] adda     #>4,r0,r2                  ;\[310\] ]

Epd3: [ tfr      d0,d7                      ;\[317\] extractu #<29,#<3,d0,d11            ;\[318\] extractu #<4,#<28,d0,d5             ;\[319\] move.l   (r2),d1                    ;\[310\] ]

Epd4: [ tfr      d1,d9                      ;\[321\] extractu #<4,#<28,d1,d11            ;\[323\] tfr      d0,d10                     ;\[318\] tfr      d1,d14                     ;\[320\] ]

Epd5: [ asll     #<4,d9                     ;\[321\] asll     #<4,d7                     ;\[317\] asll     #<3,d10                    ;\[318\] asll     #<29,d14                   ;\[320\] push     r6                         ;\[299\] push     r7                         ;\[299\] ]

Epd6: [ move.w   #56,r6                     ;\[299\]; adda     #>8,r6,r3                  ;\[310\] ]

Epd7: [ move.l   (r3),d2                    ;\[310\] adda     #>12,r3,r4                  ;\[311\] ]

Epd8: [ tfr      d2,d15                     ;\[325\] move.l   (r4),d3                    ;\[311\] adda     r6,sp                       ;\[299\] ]

Epd9: [ asll     #<4,d15                    ;\[325\] adda     #>16,r5,r5                  ;\[311\] adda     #>20,r6,r6                  ;\[312\] ]

Epd10: [ ]
move.l (r6),d6  
move.l (r5),d4

Epd11:  
  eor d7, d6  
  eor d5, d4  
  tfr d10,d17  
  tfr d11,d5  
  adda #>24,r0,r7

Epd12:  
  eor d0, d6  
  asll #<3,d5  
  asll #<29,d7  
  move.l (r7),d8  
  adda #>28,r0,r8

Epd13:  
  eor d7, d8  
  eor d10,d6  
  tfr d2,d10  
  tfr d3,d7  
  move.l (r8),d13

Epd14:  
  eor d9, d8  
  asll #<3,d5  
  asll #<29,d7  
  move.l (r7),d8  
  adda #>28,r0,r8

Epd15:  
  eor d7, d8  
  eor d10,d6  
  eor d11,d6  
  extractu #<29,#<3,d1,d11

Epd16:  
  eor d5, d8  
  extractu #<4,#<28,d2,d12

Epd17:  
  extractu #<3,#<29,d1,d1  
  eor d2,d13  
  eor d11,d8  
  tfr d3,d11  
  move.w #56,r6

Epd18:  
  eor d12,d8  
  tfr d2,d12  
  asll #<4,d11  
  eor d1,d6  
  adda #>32,r0,r9

Epd19:  
  asll #<3,d12  
  asll #<3,d7  
  move.l (r9),d9  
  adda #>36,r0,r10

Epd20:  
  eor d12,d13  
  eor d10,d9  
  extractu #<29,#<3,d3,d10
  move.l (r10),d5
  suba r6,sp
DW_192

Epd21: [eor d11,d9 ;329]  
  eor d14,d13 ;326]  
  tfr d3,d14 ;328]  
  adda #>40,r0,r11 ;314]
]

Epd22: [eor d3,d9 ;329]  
  eor d15,d13 ;331]  
  extractu #<3,#<29,d0,d15 ;319]  
  asll #<29,d14 ;328]  
  move.l (r11),d12 ;314]
]

Epd23: [eor d15,d4 ;319]  
  eor d14,d5 ;328]  
  eor d7,d9 ;330]  
  extractu #<3,#<29,d2,d2 ;327] B2]
]

Epd24: [tfr d4,d0 ;333]  
  eor d10,d9 ;330]  
  extractu #<4,#<28,d4,d11 ;335]  
  tfr d4,d15 ;332]
]

Epd25: [asll #<4,d0 ;333]  
  asll #<29,d15 ;332]  
  eor d11,d9 ;333]  
  extractu #<3,#<29,d3,d3 ;331] B2]
]

Epd26c [eor d0,d5 ;333]  
  eor d15,d12 ;332]  
  eor d2,d8 ;327] B2]  
  tfr d4,d10 ;334] B2  
  move.w #<-8,d0 ;337]
]

Epd27: [and d6,d0 ;337]  
  eor d4,d5 ;333] B2  
  extractu #<3,#<20,d4,d2 ;335] B2  
  and #<7,d6,d7 ;341] B2]
]

Epd28: [tfr d0,d1 ;338]  
  eor d3,d13 ;331] B2]  
  adda #>4,r1,r12 ;343] B2]  
  adda #>8,r1,r13 ;344] B2]
]

Epd29: [asll #<4,d1 ;338]  
  eor d2,d9 ;335] B2]  
  extractu #<29,#<3,d4,d11 ;334] B2  
  adda #>12,r1,r14 ;344] B2]
]

Epd30: [eor d1,d12 ;338]  
  tfr d0,d1 ;338] B2]  
  extractu #<4,#<28,d0,d14 ;340] B2  
  adda #>16,r1,r15 ;345] B2]
]

Epd31: [asll #<3,d1 ;338]  
  extractu #<3,#<29,d0,d3 ;340]  
  extractu #<29,#<3,d0,d2 ;339]  
  extractu #<29,#<3,d0,d3 ;340]  
  extractu #<29,#<3,d0,d2 ;339]
move.l d7,(r1)  ;[343]
]
Epd32: [  
eor d1,d12  ;[338]
asl #<3,d10  ;[334]
move.l d8,(r12)  ;[343]
adda #<20,r1  ;[345]
]
Epd33: [  
eor d10,d5  ;[334]
eor d12,d12  ;[339]
movelo d13,(r13)  ;[344]
]
Epd34: [  
eor d11,d5  ;[334]
eor d2,d12  ;[339]
pop r6  ;[346]
pop r7  ;[346]
]
DW_206
Epd35: [  
eor d14,d5  ;[340]
movelo d9,(r14)  ;[344]
]
Epd36: [  
eor d3,d5  ;[340]
pop d6  ;[346]
pop d7  ;[346]
]
DW_208
Epd37: [  
movelo d5,(r15)  ;[345]
rsd  ;[346]
]
Epd38: movelo d12,(r1)  ;[345]
finish_polydivnew:
Epd39:  rts

; END B11
;*******************************************************************************
; Function poly_mul
;*******************************************************************************

; BEGIN B9

PUM:
Epm1: [  
clr d0  ;[200]
doen2 #<16  ;[200]
dosetup2 L336  ;[200]
]
Epm2: [  
push d6  ;[272]
push d7  ;[272]
]
DW_212
Epm3: [  
push r6  ;[272]
push r7  ;[272]
]

DW_213
Epm4: movew #104,r6  ;[272]
Epm5: movelo #lu_table,r2  ;[200]
Epm6: adda r6,sp
; Inline function: _poly_mul%1_pre_compute_table

DW_216
falign
loopstart2
L336
Epm7:  [*]
  doensh3 #<11 ;[0]
  tfra r2,r3 ;[200]
[
]
Epm8:  nop ;[0] L_D_3
loopstart3
L335
Epm9:  move.l d0,(r3)+ ;[203]
  loopend3
Epm10:  adda #>44,r0,r6 ;[198]
Epm11:  nop ;[0] L_L_2
  loopend2
Epm12:  doensh0 #<6 ;[0]
Epm13:  [*]
  move.l #lu_table+64,r2 ;[208]
  tfra r1,r3 ;[209]
[
]
loopstart3
L337
Epm14:  move.l (r3)+,r4 ;[212]
Epm15:  move.l r4,(r2)+ ;[212]
  loopend3
Epm16:  [*]
  move.w #<31,d3 ;[239]
  move.w #<1,d1 ;[220]
[
]
Epm17:  [*]
  move.w #<2,d2 ;[221]
  move.l d3,(sp-36) ;[239]
[
]
Epm18:  move.w #<44,d4 ;[239]
Epm19:  nop
Epm20:  nop
  falign
  nop
  nop
  nop
L100
; Inline function: _poly_mul%1_pre_compute_table%1Block1
;
Epm21:  [*]
  impysu d2,d4,d5 ;[228]
  doensh3 #<6 ;[0]
[
]
Epm22:  [*]
  imacus d2,d4,d5 ;[228]
  impyuu d2,d4,d6 ;[228]
  tfra r1,r5 ;[229]
[
]
Epm23:  [*]
  aslw d5,d5 ;[228]
  move.l #lu_table,r3 ;[228]
[
]
Epm24:  iadd d6,d5 ;[228]
Epm25:  sxt.l d5 ;[228]
Epm26:  move.l d5,r2 ;[228]
Epm27:  nop ;[0] AGU stall
Epm28:  adda r3,r2 ;[228]
Epm29:  adda #>20,x2,r4 ;[228]
  loopend3
L329
Epm30:  move.l (r5)+,r6 ;[232]
Epm31:  move.l r6,(r4)+ ;[232]
loopend3
Epm32: [ clr   d0          :238
  move.w #<1,d3          :239
  doen3 #10          :0] @II6
]
Epm33: [ sub  d3,d2,d6           :239
  dosetup3 L330           :0]
  adda  #>40,r2,r7          :237
]
Epm34: [ move.l (r7),d7        :242 0%=0
  move.l (sp-36),d3        :239
]
Epm35: [ tfr  d7,d8   :242 1%=0
  zxt.l  d3             :239
]
Epm36: [ asll d3,d6          ;239
  asll d1,d8               ;242 2%=0
]
Epm37: [ or  d8,d0           ;242 3%=0
  and  d6,d7               ;243 1%=0
]
Epm38: [ tfr  d7,d0        ;243 4%=0
  move.l d0,r4           ;242 4%=0
]
Epm39: [ zxt.l  d0          ;243 5%=0
  move.l +4,(r7)        ;244 5%=0
]
  falign
  loopstart3
L330
Epm40: [ lerr  d3,d0         ;244 6%=1
  move.l (r7),d7          ;242 0%=0
]
Epm41: [ tfr  d7,d8        ;242 1%=0
  and  d6,d7               ;243 1%=0
]
Epm42: asll d1,d8           ;242 2%=0
Epm43: or  d8,d0            ;242 3%=0
Epm44: [ tfr  d7,d0        ;243 4%=0
  move.l d0,r4           ;242 4%=0
]
Epm45: [ zxt.l  d0          ;243 5%=0
  move.l +4,(r7)        ;244 5%=0
]
  loopend3
Epm46: [ lerr  d3,d0         ;244 6%=1
  move.w #<1,d5          ;254 B12
  move1. #lu_table+44,r5 ;253 B12
]
Epm47: [ iadd  d2,d5          ;254 B12
  move.w #<1,d0          ;250
]
Epm48: [ cmpgt  d0,d2          ;250
  sub  d0,d2,d0           ;0] B12

192
loopstart3
L338
Epm73:  move.l  d0,(r5)+              ;[90]
  loopend3
Epm74:  [
  doen1  #<8              ;[106]
  dosetup1 L334             ;[106]
  ]
Epm75:  [
  dosetup2 L332             ;[9]
  move.w  #<7,d1             ;[95]
  ]
Epm76:  [
  adda  #<20,r0             ;[112]
  adda  #>-76,sp,r1           ;[125]
  ]
Epm77:  [
  adda  #>-56,sp,r2           ;[143]
  move.w  #<28,d2             ;[106]
  ]
Epm78:  move.l  #his_table,d8          ;[106]
  falign
  loopstart1
L334
Epm79:  [
  doen2  #<6             ;[9]
  dosetup3 L331            ;[133]
  ]
Epm80:  [
  move.w  #<15,d3             ;[106]
  tfra  r1,r3              ;[125]
  ]
Epm81:  asll  d2,d3             ;[106]
  falign
  loopstart2
L332
Epm82:  [
  move.l  (r0),d5            ;[112]
  doen3  #<6              ;[9]
  ]
Epm83:  [
  and  d3,d5              ;[112]
  tfra  r3,r5              ;[133]
  tfra  r3,r6              ;[9]
  ]
Epm84:  zxt.l  d5            ;[112]
Epm85:  lrr  d2,d5            ;[112]
Epm86:  [
  impysu  d5,d4,d6          ;[126]
  impyuu  d5,d4,d0          ;[126]
  ]
Epm87:  imacus  d5,d4,d6          ;[126]
Epm88:  ashw  d6,d6            ;[126]
Epm89:  iadd  d0,d6            ;[126]
Epm90:  sxt.l  d6            ;[126]
Epm91:  add  d8,d6,d9           ;[126]
Epm92:  move.l  d9,r4           ;[126]
Epm93:  nop                   ;[9]  AGU stall
Epm94:  adda  #<20,r4           ;[126]
Epm95:  nop                   ;[9]  falign
  loopstart3
L331
Epm96:  [
  move.l  (r4)+,d6          ;[133]
  move.l  (r6)+,d5          ;[133]
  ]
Epm97:  eor  d5,d6           ;[133]
Epm98: move.l d6,(r5)+              ;[133]
loopend3
Epm99: suba #<4,r3                   ;[108]
loopend2
Epm100: 
  tsteq d1                        ;[136]
  clr d0                         ;[144] B25
  tfra t2,r3                     ;[143] B25
]
Epm101: bt <L120 ;<L120            ;[136]
; Inline function: _poly_mul%2_poly_mul_window%2Block0
Epm102: [ 
  doen3 #<11                       ;[0]
  dosetup3 L333                     ;[0]
]
  falgn
loopstart3
L333
Epm103: move.l (r3),d3                ;[148]
Epm104: [ 
  tfr d3,d5                       ;[148]
  and #-268435456,d3,d6           ;[149]
]
Epm105: asll #<4,d5                   ;[148]
Epm106: or d5,d0                     ;[148]
Epm107: [ 
  extractu #<4,#<28,d6,d0          ;[150]
  move.l d0,r4                     ;[148]
]
Epm108: move.l r4,(r3)                ;[150]
loopend3
L244
; End inline _poly_mul%2_poly_mul_window%2Block0
;
L120
Epm109: [ 
  sub #1,d1                       ;[95]
  sub #<4,d2                      ;[95]
  adda #<24,r0                    ;[0]
]
Epm110: nop                         ;[0] L_L_1
Epm111: nop                         ;[0] L_C_1
loopendd
Epm112: jsrd poly_div_new            ;[279]
Epm113: [ 
  move.l (sp-132),r1              ;[279]
  adda #>-96,sp,r0                ;[279]
]
Epm114: move.w #104,r6              ;[280]
Epm115: nop                         ;[0] AGU stall
Epm116: subu r6,sp                   ;[280]
DW_323
Epm117: [ 
  pop r6                          ;[280]
  pop r7                          ;[280]
]
DW_324
Epm118: [ 
  pop d6                          ;[280]
  pop d7                          ;[280]
]
DW_325
finish_polymul:                      
  ;Epm119: rts                       ;[306]
Epm119 jmp APUM

; END B10
end