Evaluation of Software Energy Consumption on Microprocessors

By

Mitra M. Osqui

Bachelor of Science in Electrical Engineering with Highest Distinction and Honors
Purdue University, May 1998

SUBMITTED TO THE DEPARTMENT OF

ELECTRICAL ENGINEERING AND COMPUTER SCIENCE

IN PARTIAL FULFILLMENT OF THE REQUIREMENTS FOR THE

DEGREE OF

MASTER OF SCIENCE

IN

ELECTRICAL ENGINEERING AND COMPUTER SCIENCE

AT THE

MASSACHUSETTS INSTITUTE OF TECHNOLOGY

OCTOBER 2001

[February 2002]

Signature of Author: .....
Department of Electrical Engineering and Computer Science
August 2001

Certified by: .....
Anantha P. Chandrakasan
Associate Professor
Thesis Supervisor

Accepted by: .....
Arthur C. Smith
Chairman, Department Committee on Graduate Students

EVALUATION OF SOFTWARE ENERGY CONSUMPTION ON MICROPROCESORS

By

MITRA M. OSQUI

Submitted to the Department of
Electrical Engineering and Computer Science
In Partial Fulfillment of the Requirements for the Degree of
Master of Science in Electrical Engineering and Computer Science

ABSTRACT

In the area of wireless communications, energy consumption is the key design consideration. Significant effort has been placed in optimizing hardware for energy efficiency, while relatively less emphasis has been placed on software energy reduction. For overall energy efficiency reduction of system energy consumption in both hardware and software must be addressed.

One goal of this research is to evaluate the factors that affect software energy efficiency and identify techniques that can be employed to produce energy optimal software. In order to present a strong argument, two state-of-the-art low power processors were used for evaluation: the Intel StrongARM SA-1100 and the next generation Intel Xscale processor. A key step in analyzing the performance of software is to perform a comprehensive tabulation of the energy consumption per instruction, while taking into account the different modes of operation. This leads into a comprehensive energy profiling for the instruction set of the processors of interest.

With information on the energy consumption per instruction, we can evaluate the feasibility of energy efficient programming and use the results to gain greater insight into the power consumption of the two processors under consideration. Benchmark programs will be tested on both processors to illustrate the effectiveness of the energy profiling results. The next goal is to look at the leakage current and current consumed during idle modes of the processors and how that impacts the overall picture of energy consumption. Thus energy consumption will be explored for the two processors from both a dynamic and static energy consumption perspective.

Thesis Supervisor: Anantha P. Chandrakasan
Title: Associate Professor

I lovingly dedicate my work to my dear father, without whom I wouldn't be where I am today.

I will forever be thankful for absolutely everything he has done for me, which is endless. For all that he has taught me, for all that he has put within my reach, and for always being there for me, he truly was the best father I could have asked for.

I will always miss him very much.

Acknowledgments

I would like to thank God for all that I have been blessed with, and all that has been possible for me, and all that God continues to bless me with, I am truly grateful.

I would like to express my most sincere appreciation to Professor Dr. Anantha P. Chandrakasan; he is a great professor, advisor, and person. He has provided endless guidance, support, and patience throughout my whole research work here at MIT. He is beyond words knowledgeable and this work would not have been completed without his valuable insightful suggestions and guidance. I had the pleasure of meeting Dr. Chandrakasan for the first time when I took his class, 6.374 Analysis and Design of Digital Integrated Circuits, when I first came to MIT. I then had the opportunity to get acquainted with his laboratory and research group, by taking a couple of research courses from him. When the time came to begin work on my thesis, there was no doubt in my mind that I wanted Dr. Chandrakasan to be my thesis supervisor. I have enjoyed working with him and it has been a rewarding experience. I look forward to starting my doctoral research with Dr. Chandrakasan in the near future.

Now I would also like to thank my very good colleagues in 38-107. Amit Sinha, for all his help, valuable suggestions, and endless patience. I also want to thank Manish Bhardwaj, for also being very patient and always helpful. Rex Min for providing helpful suggestions by reading part of this thesis and for providing his Digital Camera for the photos in this work. Alice Wang for also offering suggestions by reading part of this work. Lastly, Nathan Ickes for his help with the Intel Xscale processor and its setup.

I want to thank Raytheon Systems Company, my employer, who has made my education here at MIT possible, by awarding me with a full-time fellowship.

I would also like to thank Intel for providing the Intel Xscale evaluation board, which has been essential to this thesis. This work is in part funded under the DARPA Power Aware Computing Communications Program.

Lastly I would like to thank my dear mother for her endless love, dedication, support and patience.

Mitra M. Osqui
MIT, August 2001

Table Contents

List of Figures

List of Tables

Chapter 1

Introduction

1.0 Low Power Trends

Power consumption is becoming a serious issue in wireless portable electronics, since they run on limited energy resources. Minimization of power dissipation is not limited to portable devices, as it is desirable to have desktop electronics that consume less energy and hence are cost effective [1]. As performance requirements grow, current microprocessors integrate millions of transistors, resulting in a dramatic power dissipation increase. Microprocessors today dissipate on the order of 10's of watts [2]. Clearly if the circuitry could be optimized to operate with lower supply voltages, energy would be reduced. There has already been significant effort in reducing energy consumption of hardware. The next step is to explore methods to minimize energy consumption through software. This is also very important, since the total energy consumed by the circuit or system is partly determined by the energy efficiency of the program running on processors. The focus of this research is to evaluate and present results of extensive analysis of software energy consumption.

1.1 Background

Previous research has been done to produce faster, more energy efficient code in high-level languages such as C programming [3-9]. In addition to utilizing the specific optimization options that are readily available to the processor, more sophisticated techniques that are more general can be applied by designing a smart compiler that can optimize code for energy efficiency; this can include the techniques suggested in [6], such as loop unrolling, software pipelining, and recursion elimination.

The goal of this research is to develop a very comprehensive instruction level model for two processors, Intel Xscale [10-17] and Intel StrongARM SA-1100 [10, 18-23], and identify potential opportunities for energy savings by evaluating the source of the energy consumption.

An estimation tool can be developed that will utilize the energy model of the processors to predict the energy consumed by a given program [24]. The energy consumption models are of great value alone since these processors are designed for high performance DSP and embedded applications. Analysis of the Xscale processor and comparisons with its predecessor (i.e. the StrongARM) will provide interesting insightful results; such as patterns of energy consumption of the two processors and energy efficiency of the processors. These results will also be helpful for better processor design and can be used to design a more energy optimal architecture for executing instructions.

To get a very complete energy consumption model for the processors in question, all sources of energy consumption must be explored. Both static and dynamic power consumption will be explored, since they are the primary sources of energy consumption. In CMOS circuits static power consumption can be attributed to reverse bias diode leakage and sub-threshold conduction, with sub-threshold conduction now being the dominating factor. Dynamic power consumption is due to switching currents that result from charging and discharging of parasitic capacitors. Dynamic power dissipation is usually the dominating factor contributing to about 90% of the total power dissipated [25]. Therefore, since dynamic power dominates, this will be explored in greater detail and we will produce a model that will minimize this type of energy consumption. Power is proportional to the square of the supply voltage, as given by the following equation [25-26]:

$$P_{dynamic} = \alpha_{0 \rightarrow 1} C_L V_{dd}^2 f_{clk}$$

Where C_L is the load capacitance, V_{dd} is the supply voltage, $\alpha_{0 \rightarrow 1}$ is the activity factor, and f_{clk} is the clock or switching frequency. The energy of a program can be calculated directly by the following equation:

$$E = V_{dd} \times I \times t_{run}$$

Where E is the energy consumed per program, I is the core current, and t_{run} is the run time. There are techniques that enable accurate and meaningful measurements of the energy consumption of the processor core [27-32], and these steps will be analyzed and outlined in detail in section 2.3.2. The Xscale processor uses the GNU compiler [33-37] and the StrongARM processor uses the ARM SDT v2.11 tool [38-40].

Upon completion of a thorough analysis and understanding of how energy is consumed during dynamic operation, the focus is placed on static power dissipation that can be explained with the following relation, where I_{subth} is the sub-threshold current:

$$P_{static} = I_{subth} V_{dd}$$

If the circuit is idling for a long time, it can dissipate a significant amount of energy. This is especially true for portable devices such as cellular phones, where the device spends significant time in the idle state. Both processors have the capability to be operated at a range of frequencies with corresponding minimum core voltages requirements; these different operation points are evaluated for leakage current levels and the use of the idle modes such as idle, drowsy, and sleep are explored to evaluate their effectiveness in preserving energy [1, 41].

1.2 Preface of Thesis

This work tries to evaluate and answer some questions related software energy consumption on the two chosen processors:

- How much energy do various instructions consume?
- How much variation exists in energy consumption across instructions? (i.e., is the amount of energy consumed per instruction dependent on the functionality of the particular instruction?)

- How can a program be more energy efficient with the knowledge of instruction energy profiling?

- What portion of total energy consumption is attributed to leakage currents?

Since dynamic power consumption dominates, this is explored in greater detail in chapters 2 and 3. Chapter 2 explores the instruction energy profiling for the Intel Xscale processor for all instructions. Then the acquired measurements are used to gain insight into how the processor core consumes energy for various instructions. The next step is to compare the results obtained for Xscale with the instruction energy profiling of the Intel StrongARM SA-1100 processor, which is presented in chapter 3. Based on the results of these two chapters the key point to be noted is that: ***The value of the energy consumed per instruction is directly dependent and proportional to the energy consumed per cycle for that instruction, where the energy per cycle is relatively constant for most instructions. Thus energy consumption per instruction is directly proportional to the cycles taken for the particular instruction to execute.***

The comparison of these two processors is taken a step further in chapter 4, by evaluating the performance and energy consumption of the two processors for a benchmark FIR filter program. The results and concepts presented in the earlier chapters are utilized to optimize the benchmark program and show the effectiveness of instruction energy profiling. Each processor operates at various frequencies and accompanying core voltages. The benchmark program is tested at the various possible frequencies for both processors to show the effect of clock frequency and accompanying core voltage change on the energy consumed per program. Lastly some performance-based comparisons are also made.

Chapter 4 also evaluates the static energy consumption of the Xscale processor based on the leakage energy model produced for the StrongARM processor [1, 41]. Again as was done for dynamic energy consumption, the static energy consumption of both processors is compared. Finally the idle modes are also evaluated for energy consumption. Chapter 5 summarizes the results.

Chapter 2

Instruction Level Energy Profiling of the Intel Xscale Processor

2.0 Introduction

The goal of this chapter is to get the instruction level energy profile for the Xscale processor. The 80200 evaluation platform (80200EVB) board is used, in which the core processor is the Intel 80200 processor (compliant with the ARM Architecture v5TE); it will be referred to as Xscale.

Each instruction is tested carefully to quantify the energy consumed for that particular instruction and form. An in depth analysis of the comprehensive results is presented in this chapter with illustrative plots. In particular, benchmark programs will be used in chapter 4 to illustrate the effectiveness of using instruction level optimization for energy efficient programming. There are some specific instructions that are only available on the Xscale; these instructions are specifically examined to determine the energy consumption with respect to the other instructions available on standard ARM processors.

A total of 63 instructions were profiled with 17 of them only specific to the Xscale processor or ARM version 5 and higher. Each instruction is accompanied by a series of different addressing mode forms; as a result a total of 327 experiments were done to obtain a very comprehensive energy profile for the instructions. A brief list of the 63 instructions is given in Table 2.1 below; further elaboration is reserved for section 2.4.

Table 2.1a: The ARM Standard and the Extended Instruction Set

ADC	ADD	AND	B	BIC	BL
CLZ	CMN	CMP	EOR	LDM(1)	LDM(2)
LDR	LDRB	LDRD	LDRBT	LDRH	LDRSB
LDRSH	LDRT	MLA	MOV	MRS	MSR
MUL	MVN	NOP	ORR	PLD	QADD
QDADD	QSUB	QDSUB	RSB	RSC	SBC
SMLAL	SMLA<x><y>	SMLAL<x><y>	SMLAW<y>	SMULL	SMUL<x><y>
SMULW<y>	STM(1)	STM(2)	STR	STRB	STRBT
STRD	STRH	STRT	SUB	SWP	SWPB
TEQ	TST	UMLAL	UMULL		
Table 2.1b: Xscale Specific Instruction Set					
MIA	MIA<x><y>	MIAPH	MAR	MRA	

2.1 Intel Xscale Processor

The Intel Xscale processor is a high performance processor that is loaded with an impressive list of features [10-17]. This processor is capable of running from 200 MHz to 733 MHz in increments of 66 MHz. The table below shows the clock frequencies and the corresponding core voltage that should be applied. The voltage values tabulated were determined by trial and error, and are the minimum required voltages to be applied at those corresponding clock frequencies. The value of the PLL configuration is used in the program to set the clock frequency [17, page 8-2]. Note that the two lower frequencies, 200 and 266 MHz, are inoperable on the 80200 board. Since the core clock must be set to at least 3 times the memory clock and the memory clock is fixed at 100 MHz [42].

Table 2.2: Frequency and voltage table for the Xscale core

PLL Clock Configuration	Corresponding Clock Frequency	Minimum Voltage
1	200 MHz	N/A
2	266 MHz	N/A
3	333 MHz	0.850 V
4	400 MHz	0.875 V
5	466 MHz	0.950 V
6	533 MHz	1.100 V
7	600 MHz	1.200 V
8	666 MHz	1.300 V
9	733 MHz	1.400 V

*Note that only a maximum of 2 V can be applied to the core.

2.2 The Setup of the 80200 Evaluation Platform

A very simplistic diagram of the overall system is shown below (Figure 2.1). In order to make the core of the processor accessible for current measurements, a slight modification is necessary, since there is no actual pin that allows for core access. The preparation steps for Xscale are outlined in appendix A1.

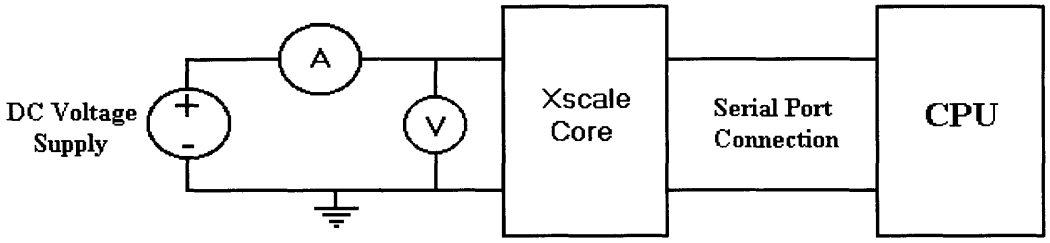

Figure 2.1: Diagram of Measurement Setup

The Keithley 2400 sourcemeter [43] is used to provide the DC source to the core and to measure the current drawn by the core. As Figure 2.2 shows the sourcemeter connections are directly made to the core. A multimeter is used to get a more accurate measure of the voltage applied to the core, the terminals of the voltmeter are connected across the Xscale core lead and ground. Voltage is supplied to the evaluation board through an adaptor that provides a 12 V constant DC voltage, pointed by an arrow to connection J9. Lastly, the connection to the computer is achieved through the serial port that is on the board as pointed to by another arrow in Figure 2.2.

2.3 Experimental Procedure

This section describes the procedure used to perform the measurements. First a brief discussion of the software used to communicate with the evaluation board and compiler is given, with reference to online manuals [33-37]. Then the actual methodology used in measuring the instruction level energy consumption will be discussed.

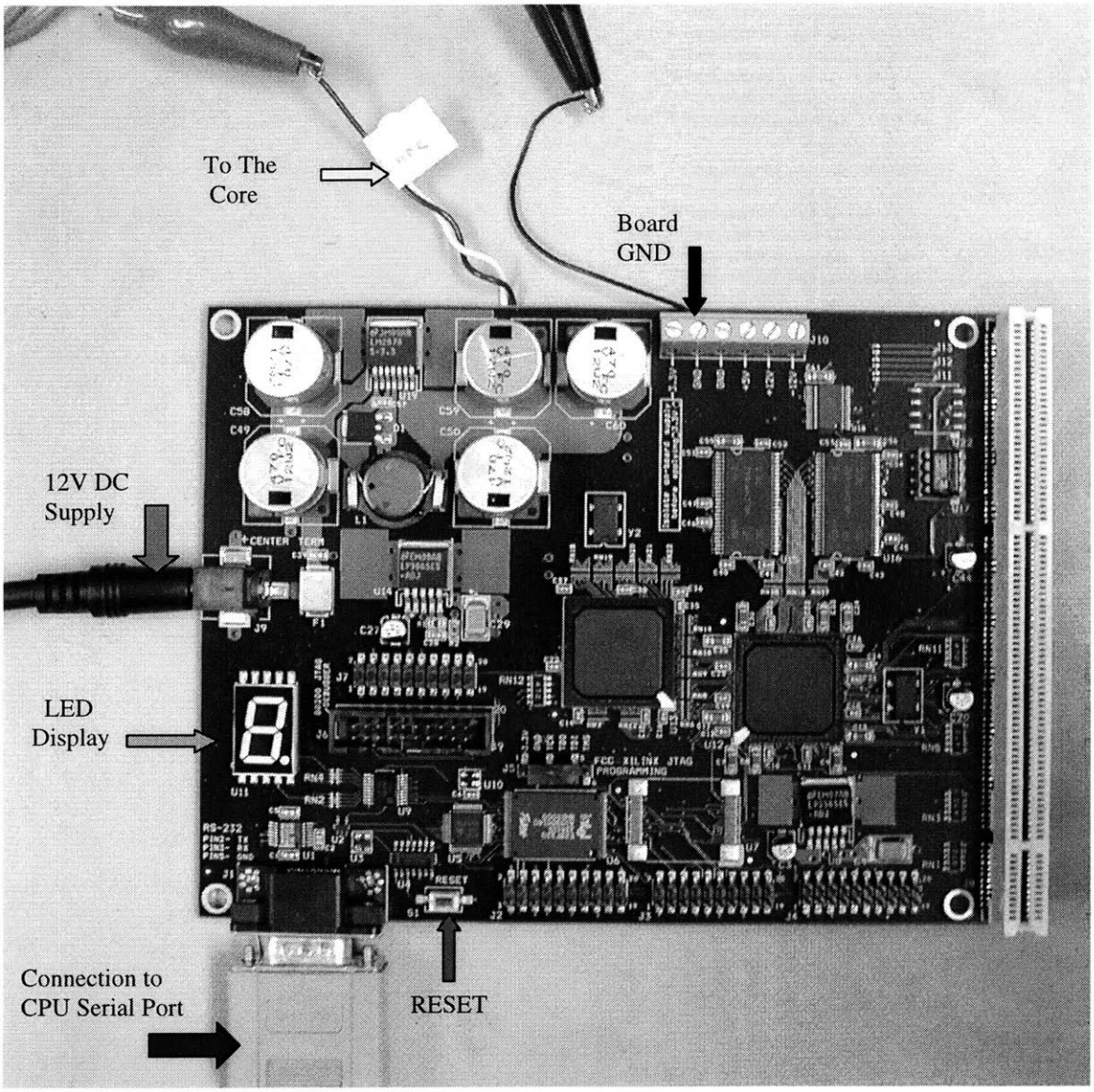

Figure 2.2: Close Up Photo of Setup

2.3.1 Computer Interface with Xscale Evaluation Board

Communication with Xscale is established through the serial port, where Cygwin [33] provides the software interface. The compiler used is the GNU compiler [34-37]; however, GNU does not operate in a windows operating system. Therefore, Cygwin is used to provide an environment to run the GNU compiler and debugger; it also provides an interface with the board. The commands for compiling and running the program on the board are outlined in appendix A2.

2.3.2 Methodology for Measuring Instruction Energy Consumption

The program to be run on Xscale is written in C with inline assembly, where the instruction to be profiled is inserted in the inline assembly segment. To get a reasonably accurate measure of the current, two major issues need to be considered. The first one is that a single instruction can't be tested alone; a loop must be put to get a relatively constant current reading. Also on average the run time of a single instruction is in the order of nanoseconds, which makes it impossible to get a current reading of a single instruction. Thus a large iteration number is needed for the loop, it was chosen to be 200 million loop iterations. The second issue is that several copies of the same instruction under test must be placed inside the loop rather than just one. This is done to compensate for the loop effects and thus to get a more accurate current and time reading. Since assembly code is embedded within the C code and the time measurements can only be done outside of the inserted assembly, the extra instructions placed to conduct the loop consume energy and time in addition to the instruction to be tested. As more instructions are placed within the loop the effect of the loop itself is minimized. Some tests were performed and it was concluded that in general 100 instructions within the loop was sufficient to compensate for the loop effects. Including more than 100 instructions gave only a marginal increase in accuracy and in many cases none.

Generally the testing for each instruction with a particular format lasted for approximately 40 seconds if the instruction took one cycle to compute. For instructions that are executed in greater cycle times, the run time increased accordingly. This is of course a reasonably long enough time for the current to stabilize. In many cases current reading stabilized in the first few seconds.

Below is a sample piece of the code used for the measurements, it is also used to actually measure the energy consumption for the add instruction in the immediate addressing mode form. The instructions that are in bold are instructions used to set up the loop, which are in effect the instructions that need to be compensated for. Another thing to note is that all the instructions and accompanying addressing modes and forms are placed in one main program and each one is tested individually by using the preprocessor directive

#ifdef and #endif to selectively choose the instruction to profile. Each single experiment block (like the one shown below) must be done individually, so lets say the experiment below is wished to be profiled, then at the beginning of the main code a #define add1 needs to be included to enable this segment. Then upon completion of this experiment, changing the directive to, for example, the following does the next experiment: #define add2.

```
#ifdef add1

 asm volatile
 ("
        ldr r0,=200000000   /*the loop is executed 200,000,000 times*/
        mov r1, #00
        mov r2, #00
1:
        add r1,r1,#01

   .rept 100              /*we are repeating the instruction to be profiled 100x*/
          add r2,r1,#01
   .endr

        cmp r1,r0
        bne 1b
  ");

#endif
```

The entire program that does the instruction level energy profiling is about 300 pages, clearly too long to be included in the appendix. Appendix A3 contains the main program without the 327 experiment segment blocks (that are like the sample for one block shown above). Note that the complete program in entirety in located in my MTL account at the following directory: ~mitra/MSThesis/Xscale/Instruction_Profiling/inst_e.c.

The next step to get the actual measurements while the program is being run on Xscale. Current and voltage readings are monitored and recorded and the following relations were used to calculate the parameters of interest [27, Chapter 5: Section 5.2.1]. The run time consumed for a single instruction (T_{inst}) is calculated using the following relation:

$$T_{inst} = \frac{T_{run}}{N_{inst} * N_{iter}}$$

Where T_{run} is the run time mentioned above, N_{inst} is the number of instructions inside the loop (100), and N_{iter} is the number of iterations (200 million). The cycles per instruction (CPI) is calculated as follows:

$$CPI = T_{inst} / T_{clk}$$

Where T_{clk} is the inverse of the clock frequency f (MHz). Finally the energy consumed per instruction (E_{inst}) and the energy consumed per cycle for each instruction (E_{cycle}) is calculated using the following relations respectively.

$$E_{inst} = (V_{dd} \times I \times CPI) / f$$

$$E_{cycle} = E_{inst} / CPI$$

Where I is the current measured with multimeter and V_{dd} is the voltage supplied to the Xscale Core. Note that CPI is used rather than run time for energy calculations since it provides greater level of accuracy. It is important to point out that as this chapter progresses with the discussion of the results of all the elaborate energy consumption profiling for the instructions, the goal is to determine the patterns of energy consumption for these instructions. Given that an elaborate table of instruction timings is available for Xscale [17, Chapter 14], the focus of the next section will be primarily on energy consumption patterns and not on timing patterns.

2.4 Results

We are finally ready to dive into the instruction level energy profiling. The Intel Xscale processor uses the ARM standard instruction set, the extended instruction set [44], and some Xscale specific instructions [17]. There are a total of 15 categories of instructions and a total of 327 instructions with various forms and addressing modes. Nine of those categories belong to the ARM standard instruction set, with 284 instructions including all addressing mode forms. All of these instructions in this set are available on Xscale and

most are available on most architecture versions [44, page A4-113]. This standard instruction set is extended to provide for enhanced DSP instructions. The extended instruction set is still part of the ARM instruction set; however, availability is limited to the ARM architecture versions v5TE (Xscale) and v5Texp [44, page A4-113]. There are 4 categories with 35 different experiments in the extended instruction set that are profiled. Lastly there are 2 categories with 8 different experiments in the Xscale specific instruction set that are profiled. The following sections present the results of the energy profiling for these instructions. Further details of syntax, addressing modes, functionality of each and every instructions is referred to the following references: [44, Chapters A3-A5], [44, ppA3-27 to A3-34 and Chapter A10], and [17, pp 2-4 to 2-8] for the ARM standard instruction set, ARM extended instruction set, and Xscale specific instructions respectively.

For all of these measurements and experiments the processor is set at a clock frequency of 533 MHz with accompanying core voltage of 1.1 V. Evaluating the effect of voltage scaling and frequency variations will be explored in chapter 4.

2.4.1 Data-Processing Instructions (Standard and Extended Instructions)

There are three individual categories of instructions that will be presented here: the 16 data-processing instructions (Table 2.3), the miscellaneous arithmetic (CLZ) instruction, and the 4 saturated addition and subtraction instructions (Table 2.4). Where the last category of instructions are from the extended instruction set. These three groups of instructions are explored together due to their close functionality.

First the standard data-processing instructions are explored, the result of the energy consumption profiling are summarized by the bar graph shown in Figure 2.3. Even though this graph is for the ADD instruction, it is to be noted that all the other 15 instructions in this category also consume the same amount of energy in each of the 11 addressing mode categories. Each of the bars represents the instruction with the different addressing modes labeled on the bar. The x-axis shows the cycles taken to execute the instruction and the y-axis shows the corresponding energy consumed for each instruction.

It is clear that shift by registers take 2 clock cycles while all other addressing modes take 1 cycle to execute.

Table 2.3: List of Data-Processing Instructions

Inst.	Description	Inst.	Description	Inst.	Description
ADC	Add with Carry	ADD	Add	AND	Logical AND
BIC	Logical Bit Clear	CMN	Compare Negative	CMP	Compare
EOR	Logical EOR	MOV	Move	MVN	Move Negative
ORR	Logical OR	RSB	Reverse Subtract	RSC	Reverse Subtract with Carry
SBC	Subtract with Carry	SUB	Subtract	TEQ	Test Equivalence
TST	Test				

It is also apparent that the 2-cycle addressing modes consume approximately twice the energy as the other forms. Figure 2.3 also shows the energy consumed per cycle, where we see that per cycle the 2 cycle addressing modes consume on the worst-case approximately 12.8% more energy compared to the 1 cycle addressing modes. The worst-case variation in energy per cycle for 1 cycle and 2 cycle instructions is 2.7% and 3.4% respectively.

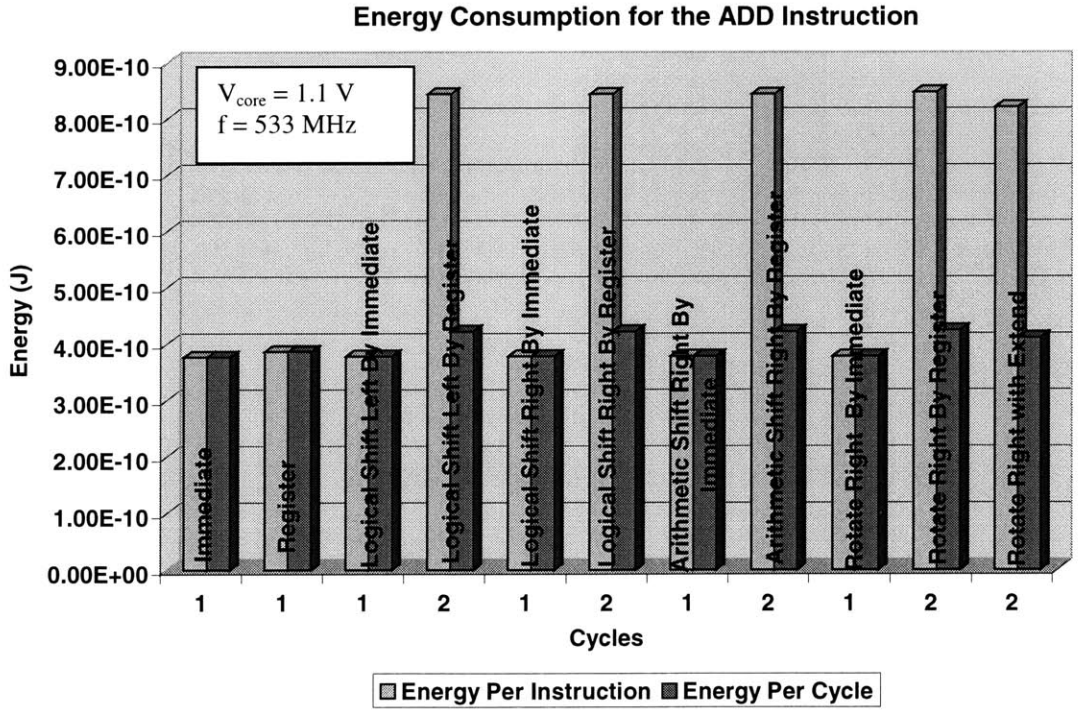

Figure 2.3: Energy consumption for the data-processing instructions

Next the miscellaneous arithmetic instruction (CLZ) is evaluated; this instruction has only one addressing mode. The energy consumption of this instruction is roughly the same as the data-processing instructions with non-register shift addressing modes. The CLZ instruction takes 1 cycle per instruction to execute; therefore $E_{inst} = E_{cycle} = .366$ nJ. This is on the same order as, lets say, the ADD instruction with immediate mode, where $E_{inst} = E_{cycle} = .376$ nJ.

The last group of instructions to be evaluated is the saturated addition and subtraction instructions, which are part of the extended instruction set. Table 2.4 provides the list of the 4 instructions in this category; again these instructions have only one addressing mode. These instructions take just 1 cycle to execute and on average these instructions consume $E_{inst} = E_{cycle} = .402$ nJ. There is only a worst-case variation of 3% between these four instructions, which can easily be attributed to measurement variation.

Table 2.4: List of Saturated Addition and Subtraction Instructions

Inst.	Description
QADD	Performs a saturated integer addition
QDADD	Performs a saturated integer doubling of one operand followed by a saturated integer addition with the other operand.
QSUB	Performs a saturated integer subtraction
QDSUB	Performs a saturated integer doubling of one operand followed by a saturated integer subtraction from the other operand.

The results for all these instructions show that energy consumed per cycle is relatively constant. To illustrate this point more effectively Figure 2.4 below shows the summarized results by averaging over the energy consumed per cycle per category for all of the above mentioned instructions. Based on the average energy per cycle values shown in Figure 2.5 there is a worst-case variation for 1-cycle instructions of 9.8% and an overall worst-case variation of 14.8%.

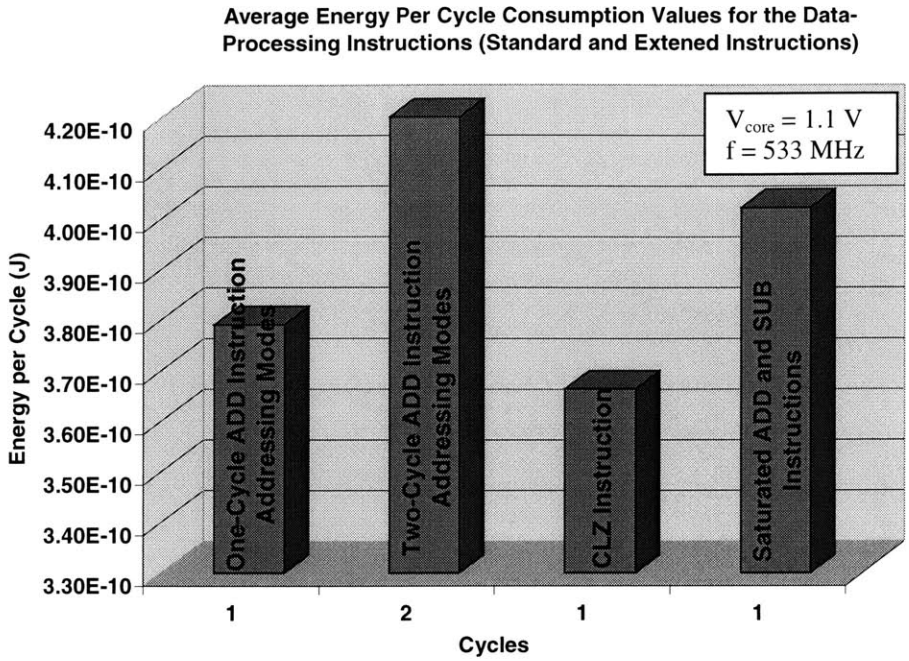

Figure 2.4: Average energy consumption per cycle for all data-processing instructions

2.4.2 Multiply Instructions (Standard, Extended, and Xscale Specific Instructions)

First the standard and extended multiply instructions are explored, where Table 2.5 provides the list of these instructions. The 5 instructions in the extended instruction set each have either 4 or 2 variations [44, page A10-6]. However, measurements showed the variations for each specific instruction consumed the same amount of energy. Thus only one variation for each instruction is shown in the bar charts below.

Table 2.5: List of Multiply Instructions

Instruction	Description
Standard Instructions*	
MLA	Multiply Accumulate
MUL	Multiply
SMLAL	Signed Multiply Accumulate Long
SMULL	Signed Multiply Long
UMULL	Unsigned Multiply Long
UMLAL	Unsigned Multiply Accumulate Long
Extended Instructions	
SMLA<x><y>	16 x 16 + 32 → 32 bit Signed Multiply
SMLAW<y>	32 x 16 + 32 → 32 bit Signed Multiply-Accumulate

SMLAL<x><y>	16 x 16 + 64 → 64 bit Signed Multiply
SMUL<x><y>	16 x 16 → 32 bit Signed Multiply
SMULW<y>	32 x 16 → 32 bit Signed Multiply

* Note: The S-bit on these six instructions was set to 1 for all measurements to ensure consistency in the results [44] and [17, page 14-6].

Figure 2.5 below shows the bar chart of energy consumption values for these instructions. The interesting point is that the MLA instruction takes roughly the same amount of energy as the MUL instruction. Therefore, whenever a multiply and accumulate is required it is clear that it is more energy efficient to use the MLA instruction than to use a MUL and ADD instruction. The other important fact is that the 3-cycle instructions consume roughly 50% more energy per instruction than the 2-cycle instructions, which implies that the energy per cycle must be roughly the same for all these instructions. This suggestion is indeed confirmed by the energy consumed per cycle also shown in Figure 2.5.

Excluding the three irregular instructions (SMLAW<y>, SMLAL<x><y>, and SMULLW<y>), which consume somewhat more energy per cycle than the others, there is a worst-case variation of only 9.2% among the other eight instructions. The average energy consumed per cycle for all eleven multiply instructions is .451 nJ. The average of the energy per cycle for all the data-processing instructions from the previous section is .392 nJ. Thus the multiply instructions consume on average 15% more energy per cycle. This is mainly due to the fact that the multiply instructions are either 2 or 3 cycle instructions, as opposed to the data-processing instructions which are mostly 1-cycle instructions.

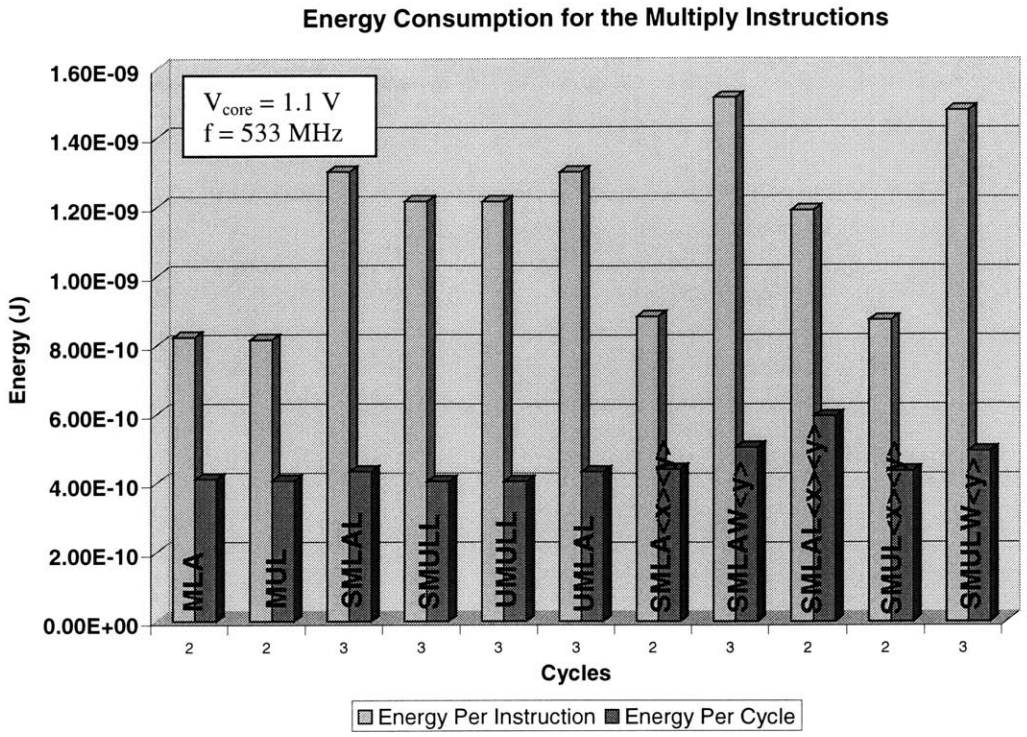

Figure 2.5: Energy consumption for the multiply instructions

Next the Xscale specific instructions are explored, there are three varieties of multiply-accumulate instructions listed in Table 2.6. These instructions accumulate the result into a single 40-bit internal accumulator, which is accessed with the two accumulator access instructions also listed in Table 2.6. The MIA<x><y> instruction also has 4 variations [17, page 2-6] and as before the variations consume the same amount of energy.

Table 2.6: List of Xscale Specific Instructions

Instruction	Description
List of Multiply with Internal Accumulate Instructions	
MIA	Multiply-accumulate using 40-bit internal accumulator
MIA<x><y>	One 16-bit signed multiply and result accumulated into a single 40-bit accumulator
MIAPH	Two 16-bit signed multiplies on packed halfword data and result accumulated into single 40-bit accumulator
List of Internal Accumulator Access Instructions	
MAR	Moves Values from Registers to 40-bit Accumulator
MRA	Moves Value from 40-bit Accumulator to Registers

The energy consumption values for this set of instructions are shown in Figure 2.6. From the energy consumed per instruction alone we see that the MIA instruction is a very energy efficient multiply-accumulate instruction, not only does it perform the multiply-accumulate operation but also it is capable of accumulating up to 40-bits. Comparing the energy per instruction for the MIA instruction with the energy per instruction for the MLA instruction we see that, the MIA instruction is 45.6% more energy efficient. Of course, this is mainly attributed to the fact that the MIA instruction is a single cycle instruction, whereas the MLA instruction takes 2 cycles to complete. Figure 2.6 also shows the corresponding energy per cycle for the instructions. Excluding the MIAPH instruction from the worst-case variation calculation, since it consumes roughly 20% more energy per cycle than the lowest energy per cycle value, the worst-case variation in the energy consumed per cycle for the other four instructions is only 8.7%. This again supports the claim that the energy per cycle is relatively constant to within an acceptable margin. There will however be exceptions and instructions that will consume somewhat more energy than the rest, as will become clear by the end of this chapter. Therefore it is perhaps better to say that the energy per cycle is roughly constant to within a first-degree approximation.

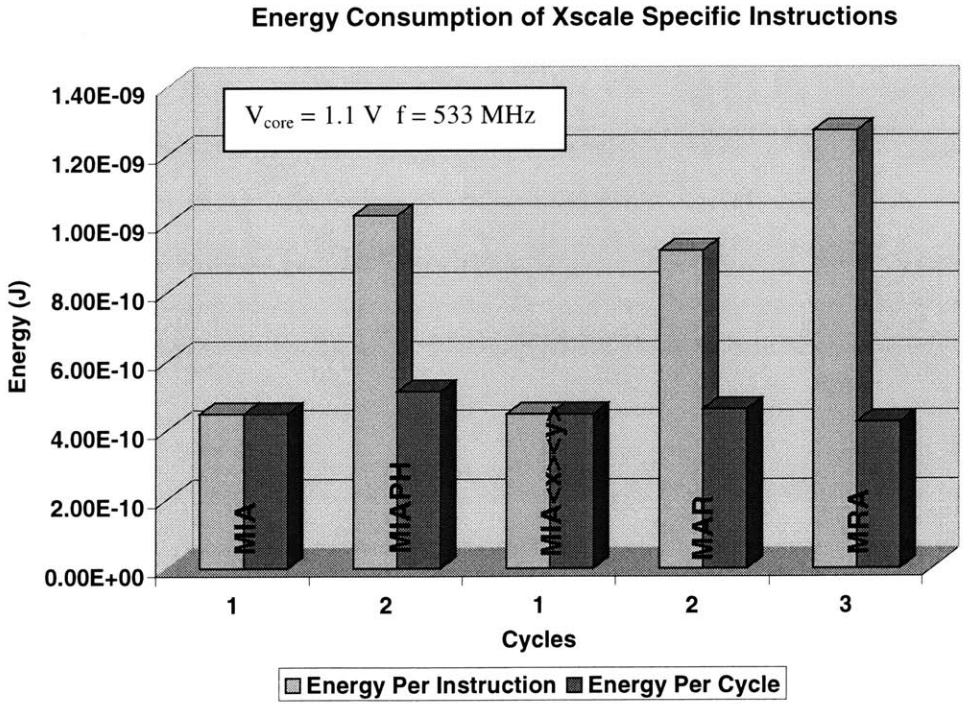

Figure 2.6: Energy consumption for the Xscale specific instructions

2.4.3 Load and Store Instructions (Standard and Extended Instructions)

Table 2.7 below lists all the load and store instructions available to Xscale by category. This section will first explore all the load instructions and then all the store instructions, with the discussion of both the load and store multiple instructions reserved to the end.

Table 2.7: List of Load and Store Instructions

Instruction	Description	Instruction	Description
List of Load and Store Word or Unsigned Byte Instructions			
LDR	Load Word	LDRB	Load Byte
LDRBT	Load Byte with User Mode Privilege	LDRT	Load Word with User Mode Privilege
STR	Store Word	STRB	Store Byte
STRBT	Store Byte with User Mode Privilege	STRT	Store Word with User Mode Privilege
List of Load and Store Halfword and Load Signed Byte Instructions			
LDRH	Load Unsigned Halfword	LDRSB	Load Signed Byte
LDRSH	Load Signed Halfword	STRH	Store Halfword
Two-Word Load and Store Instructions (Extended Instructions)			
LDRD	Loads Doublewords	STRD	Stores Doublewords
Cache Preload Instruction (Extended Instruction)			
PLD	Cache Preload		
List of Load and Store Multiple Instructions			
LDM(1)	Load Multiple	LDM(1)	Use Registers Load Multiple
STM(1)	Store Multiple	STM(2)	Use Registers Store Multiple

Each of the 9 load instructions listed above have various addressing modes; however, the addressing modes have no effect on energy consumption for each particular instruction. There is only a very small variance in energy across addressing modes, on the worst-case only 3%. Thus, Figure 2.7 illustrates only 2 bars for each of the 9 instructions, one for energy consumed per instruction and the other for the energy consumed per cycle. The energy values for each instruction is the average across the addressing modes for that instruction.

Again we see the same trend that the energy per cycle is relatively constant. For this set of instructions the overall worst-case variation in energy consumed per cycle across the 9 instructions shown in Figure 2.7 below is only 7.3%. Where the average energy per cycle across these instructions is .47 nJ.

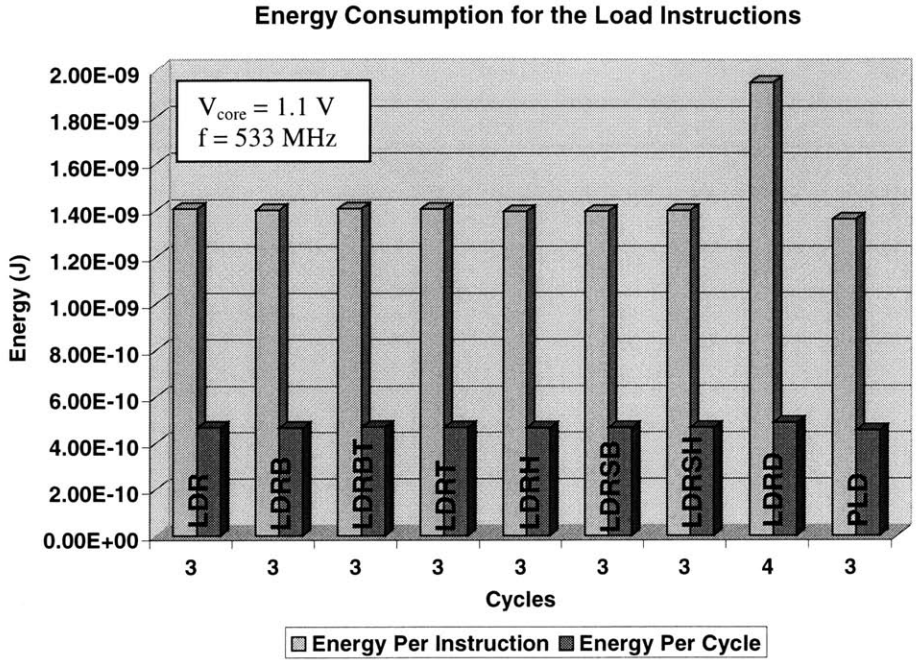

Figure 2.7: Energy consumption for the load instructions

On the other hand the addressing mode does influence the cycles taken and consequently the energy consumed by the store instructions in the first category of Table 2.8 (STR, STRB. STRBT, and STRT). Figure 2.8 below illustrates this, where even though this chart is for the STR instruction, all the other 3 instructions in this category also consume the same amount of energy in each of the addressing modes. Observing the results from Figure 2.8, we see that unlike the load operation which uses the same addressing modes, the scaled register pre-indexed and post-indexed addressing modes take 2 cycles and as a result approximately twice the energy. Figure 2.8 also shows the energy consumed per cycle for these instructions, where on the worst-case there is only a 3% variation in energy consumed per cycle.

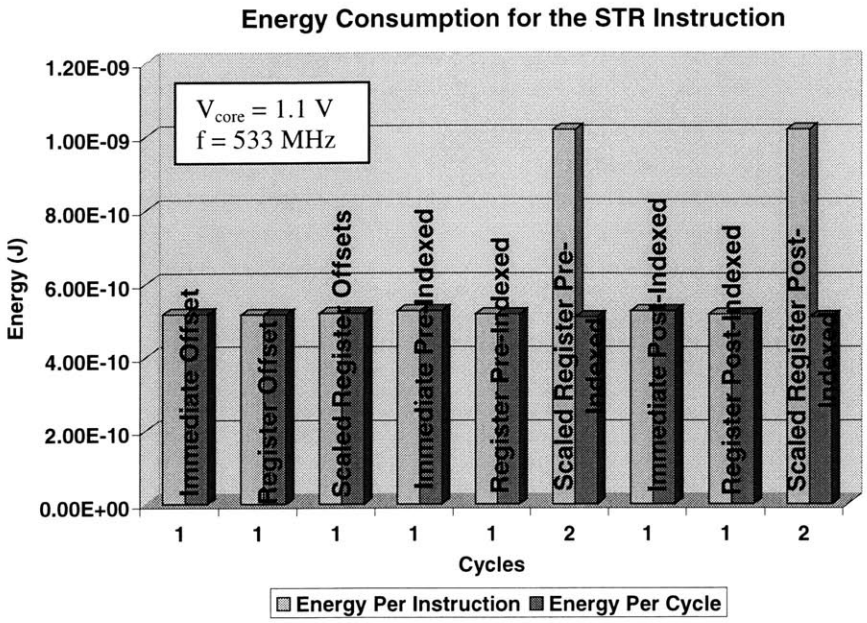

Figure 2.8: Energy consumption for the STR instruction

The energy consumed for the next two store instructions is independent on the addressing mode for the instruction. Thus Figure 2.9 shows the averaged values over the addressing modes. Figure 2.9 also shows the averaged energy for the 1 and 2 cycle addressing modes for the STR instruction. Again it is clear that over all the various store instructions and addressing modes the energy per cycle is constant.

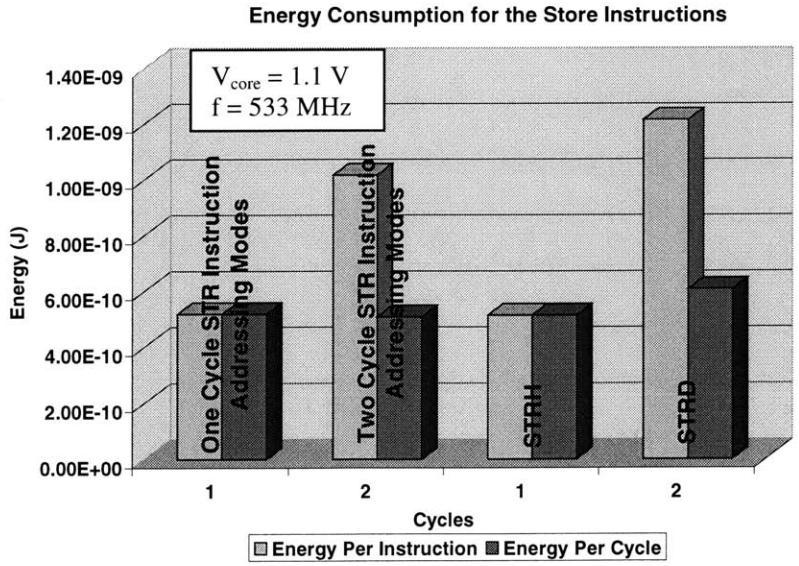

Figure 2.9: Energy consumption for all store instructions

Lastly, the load and store multiple instructions are evaluated. The load multiple instructions load from memory to the general-purpose registers. Likewise the store multiple instructions store from the general-purpose registers to memory [44]. Clearly the energy consumed by this class of instructions is dependent on the number of registers that is being loaded to or stored from. Both the LDM and STM instructions are tested for variation of energy consumption based of number of registers. From the results it can be concluded that, for just one register, 3 cycles are necessary, and an extra cycle is taken for each added register. This is true for both loads and stores, thus if 3 registers are to be used it takes 5 cycles to complete the instruction. The energy consumed per cycles remains relatively unchanged and thus the amount of energy consumed can directly be attributed to the cycles taken to execute the instruction. On average the energy consumed per cycle for the LDM and STM instructions is E_{cycle} = .58 nJ and E_{cycle} = .51 nJ with 3.8% and 4% variability across addressing modes respectively.

2.4.4 Miscellaneous Instructions

The above three sections described the three major groups of categories of instructions. Thus this section includes a brief discussion of the instructions that were left out in the above sections, where Table 2.8 below provides a list of these instructions.

Table 2.8: List of Miscellaneous Instructions

Instruction	Description
Status Register Access Instructions	
MRS	Move PSR to General-Purpose Register
MSR	Move General-Purpose Register to PSR
Semaphore Instructions	
SWP	Swap
SWPB	Swap Byte
Branch and NOP Instructions	
B	Branch
BL	Branch and Link
NOP	No Operation

Figure 2.10 below shows the energy consumed per instruction and per cycle for all of the above listed instructions. The energy results are included here for completeness;

however, no comparisons based on energy per cycle variations can be made since there is a great variation in energy per cycle across these instructions. As will be shown in the following sections, there is an increase in energy consumption per cycle as the execution cycles get higher. Final note to make is that the energy measured for the branch instructions are made for a correctly predicted branch. A mispredicted branch gets execution cycle penalties and thus consumes more energy [17, 14-2 and 14-4].

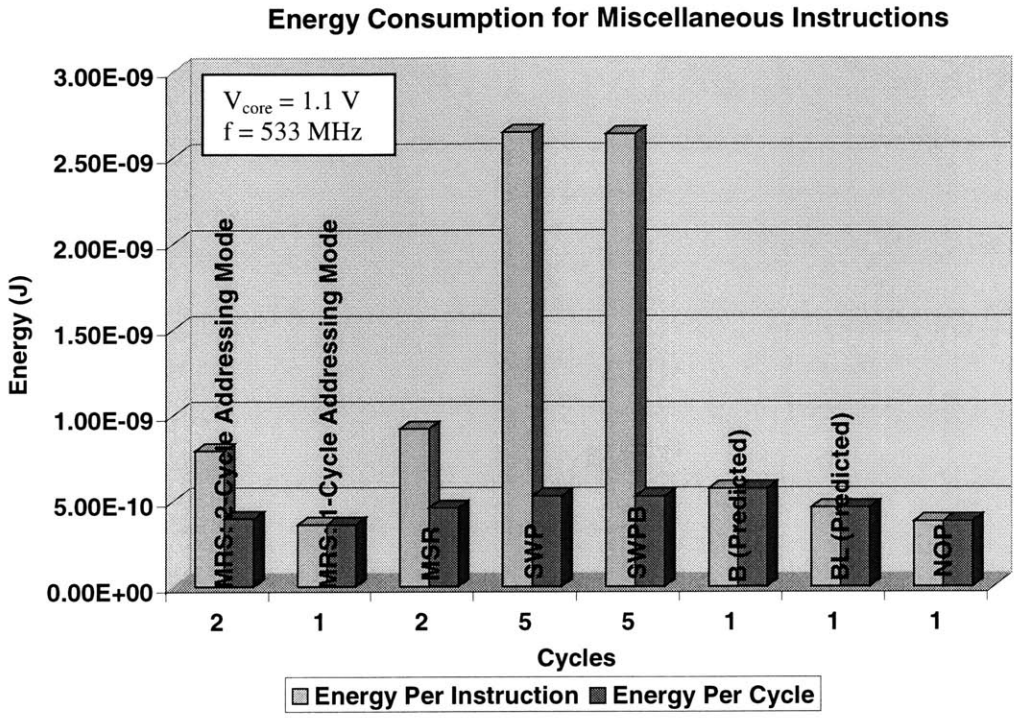

Figure 2.10: Energy consumption for miscellaneous instructions

2.5 Analysis of Results

In section 2.4 above very detailed energy profiling measurements were presented. This section uses these measurements to gain greater understanding and insight into how the processor core consumes energy for various instructions; and hopefully to be able to present a model that could be used to predict energy consumption values without need for large table lookups.

The results presented here for the most part show that: *The value of the energy consumed per instruction is directly dependent and proportional to the energy consumed per cycle for that instruction, where the energy per cycle is relatively constant for most instructions. Thus energy consumption per instruction is directly proportional to the cycles taken for the particular instruction to execute.* Therefore if it is known the number of cycles taken to execute a particular instruction, then the energy per instruction could easily be predicted, since the energy per cycle is constant for all instructions.

In an effort to show the validity of the above statement, first some statistics are presented followed by an illustrative analysis of the results. Figure 2.11 shows a scatter plot of the energy consumption values for all 327 instructions and accompanying forms. From the plot it is apparent that there seems to be a pattern for the first 193 instructions; lying either in the range of .36 - .4 nJ or .8 - .9 nJ, with 34 instructions consuming energy in between these ranges. The other 100 instructions seem to be more scattered about at higher energy values, nevertheless scattered in groups. The histogram plot in Figure 2.12 provides more detailed numbers and also supports this observation that most instructions consume energy in the above noted range.

Observations about the ranges of energy consumption do not tell much about how and why these ranges are as such. To gain greater insight into how the core consumes energy, the energy consumed per cycle for the instructions must be analyzed. Figure 2.13 shows the scatter plot for the energy per cycle for all the instructions and it is relatively flat and restricted to a small range. A more detailed plot is shown in Figure 2.14, which presents the histogram plot of the energy consumed per cycle for all the instructions. From this plot we see that there are some exceptions to the above mentioned theory, i.e. a few instructions consume up to 1.7 times the lowest energy consuming instructions. This means that even though instructions within each category tend to consume roughly the same energy per cycle, an overall look reveals that there are some exceptions.

While all these plots and histograms are informative statistics, they are not very insightful. More illustrative plots are shown in Figures 2.15 and 2.16, which show E_{inst}-

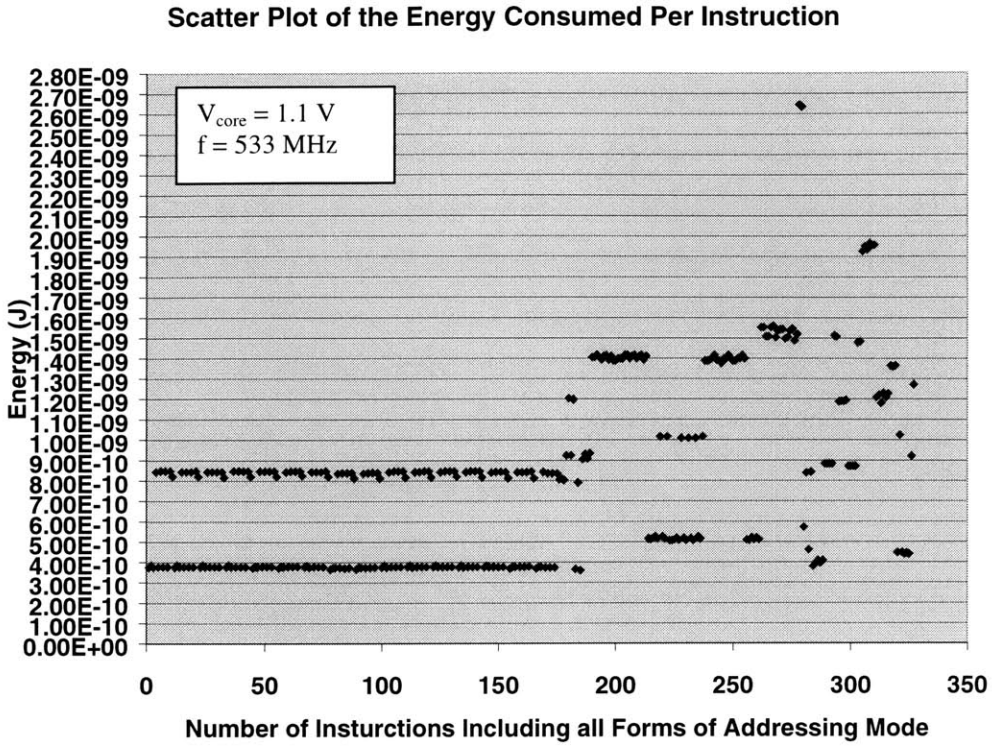

Figure 2.11: Scatter plot of the energy consumed per instruction

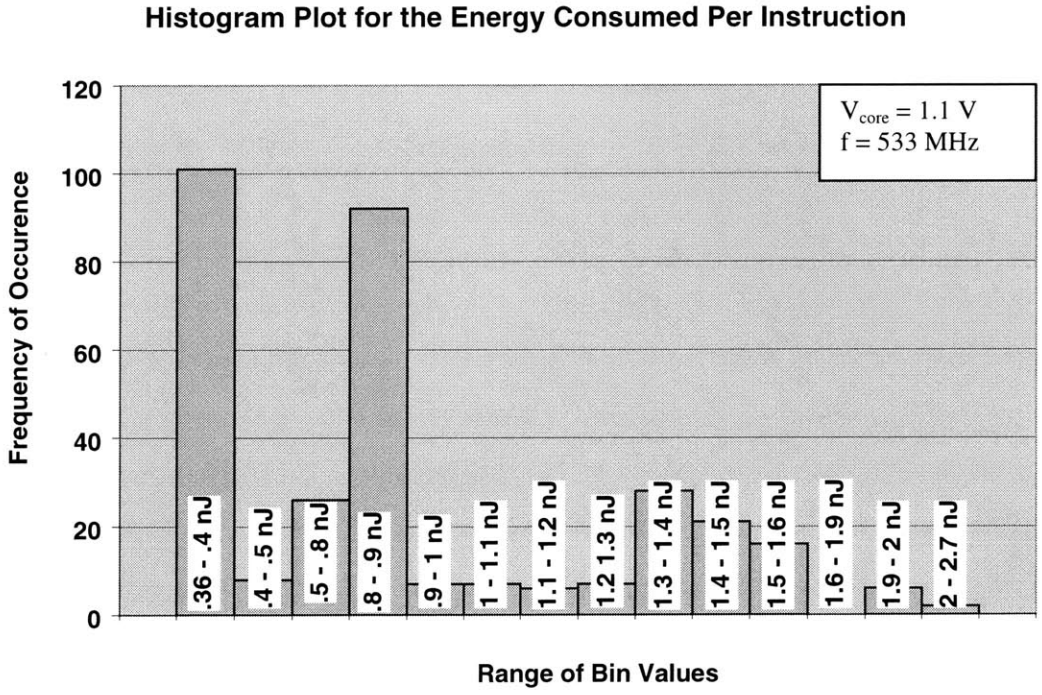

Figure 2.12: Histogram plot of the energy consumed for all instructions

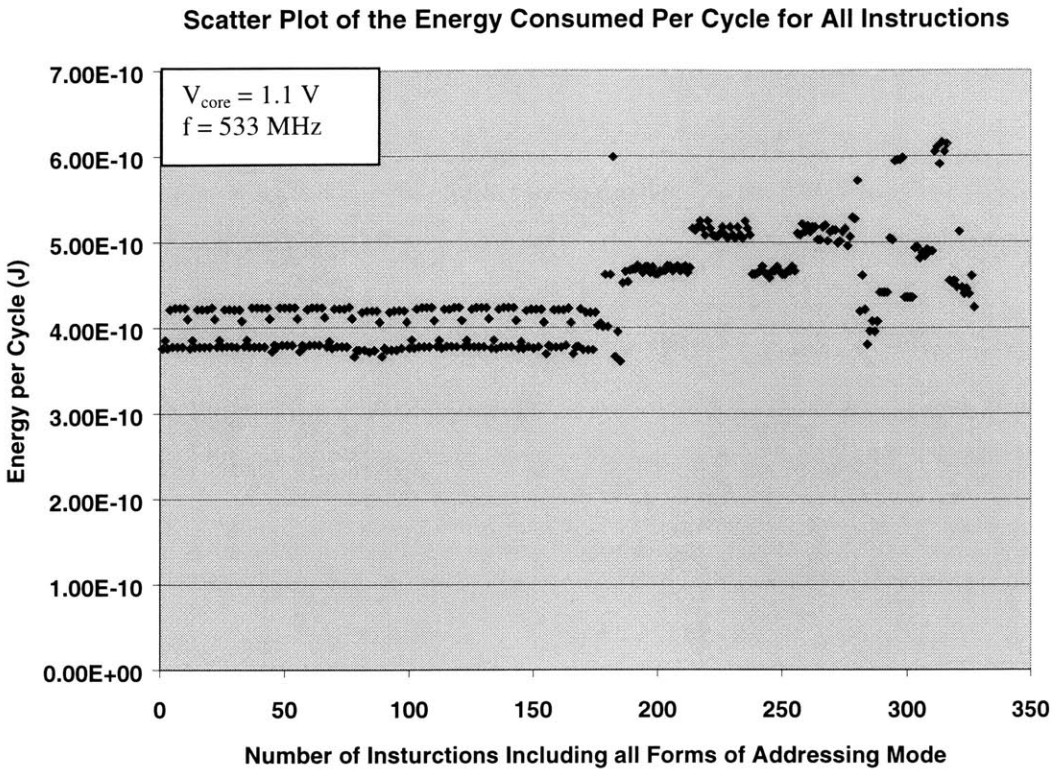

Figure 2.13: Scatter plot of the energy consumed per cycle for all instructions

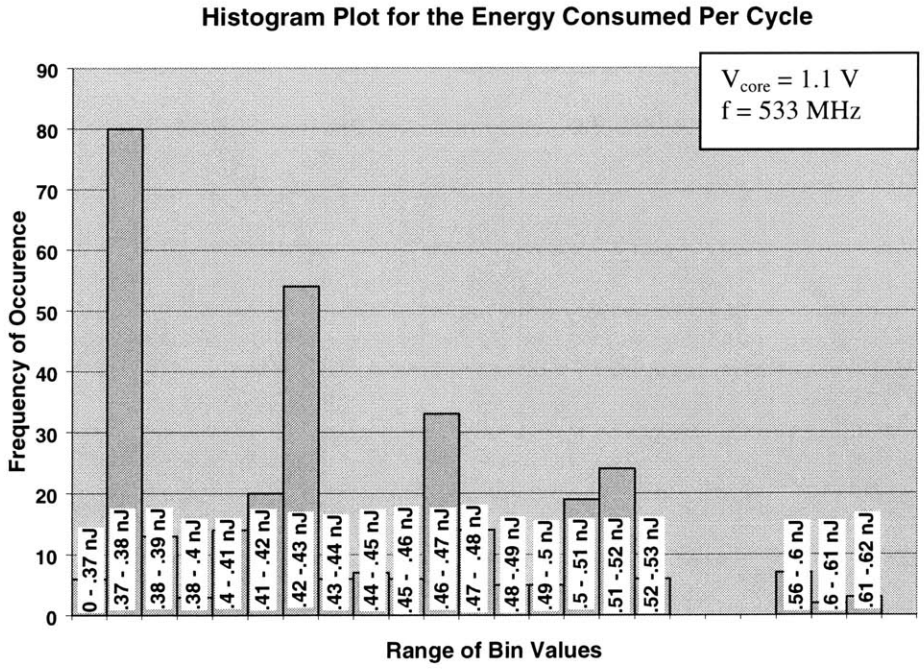

Figure 2.14: Histogram plot of the energy consumed per cycle for all instructions

vs.-cycles and E_{cycle}-cycles-cycles respectively. From Figure 2.15 we see that the energy consumption seems to double with twice the cycles and triple for 3 cycle instructions. Therefore the energy consumed per cycle must be roughly constant for most instruction types in order for this to be true (Figure 2.16). On average the energy per cycle does indeed appear to be constant; however, there clearly are some exceptions as pointed out earlier.

E_{cycle} is higher for the store and some other instructions. It is to be expected that store instructions will consume more energy per cycle, as it involves expensive memory operations. Per cycle the energy consumed for the load and store instructions is on average roughly 20% and 30% higher than the arithmetic instructions. The comparison is made with the arithmetic instructions since they comprise the vast majority of the instructions. Table 2.9 below gives a very detailed summary of various pertinent statistics for the energy consumed per cycle. Thus showing that the energy per cycle is relatively constant within each category; however, as a whole there are discrepancies and there are instructions that consume greater energy per cycle than the rest.

Table 2.9: Summary of Statistics for Energy Consumed Per Cycle

Average E_{cycle}	Worst-Case Variability	Number of Instructions	Cycles	Type of Instructions
.377 nJ	6.9%	99	1	Arithmetic Instructions, NOP, MRS
.425 nJ	12.9%	9	1	MIA, MIA<x><y>, QADD, QDADD, QSUB, QDSUB
.515 nJ	3.5%	24	1	Store Instructions
.419 nJ	7.3%	83	2	Arithmetic Instructions, MUL, MLA, MRS
.449 nJ	7.1%	15	2	SMUL<x><y>, SMLA<x><y>, MSR, MAR, SMLAL, UMLAL
.507 nJ	1.6%	7	2	Store Instructions and MIAPH
.602 nJ	4.2%	10	2	Store Instructions and SMLAL<x><y>
.466 nJ	4.4%	45	3	Load Instructions
.507 nJ	5.5%	20	3	Load and Store Multiple Inst., SMLAW<x>
.487 nJ	1.45%	6	4	LDRD
.528 nJ	.38%	2	5	SWPB, SWP

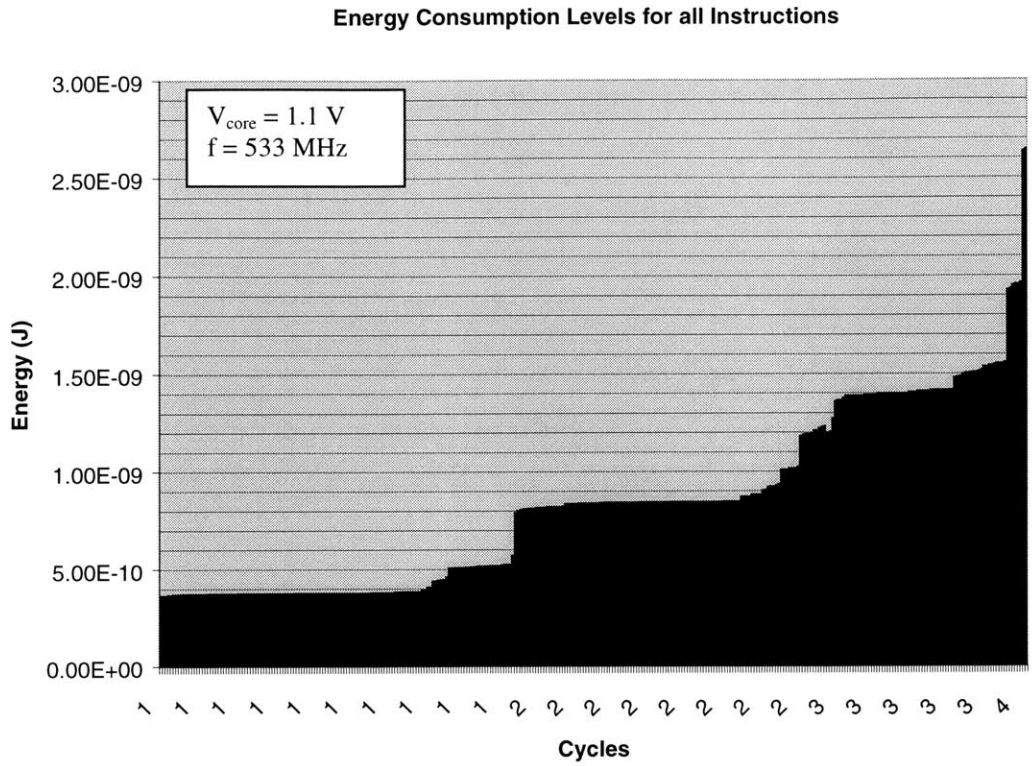

Figure 2.15: Plot of the energy per instruction versus cycles taken for all instructions

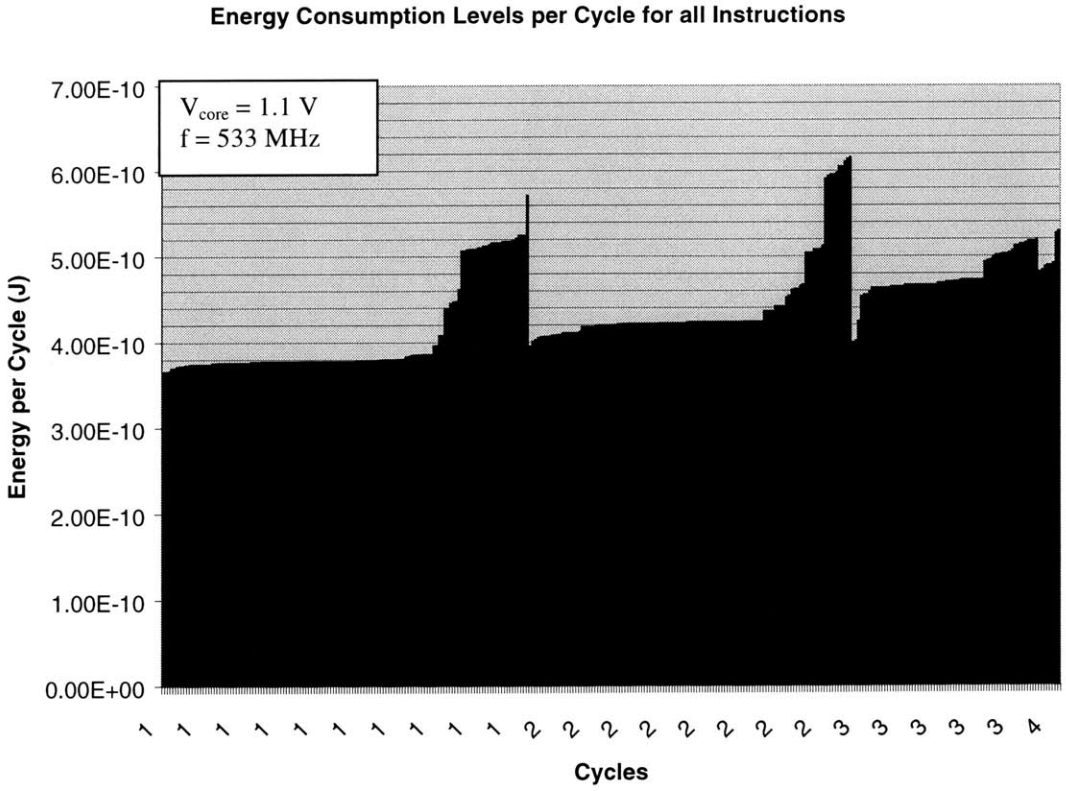

Figure 2.16: Plot of the energy per cycle versus cycles taken for all instructions

Despite the variations and irregularities that exist in energy per cycle measurements, it is for the most part relatively constant for most instructions within each of the separated groups listed in Table 2.9 (Figure 2.16). The constant nature of the energy per cycle regardless of the functionality of the instruction can be attributed to circuit overhead. Most of the processor's functional units are always on, regardless of the instruction being run, and therefore are continuously consuming energy. Of course there is some level of variability in the degree to which component is consuming energy, which leads to the exceptions pointed out earlier. The variability in energy consumption across various instructions may be partially due to the extensive clock gating features that exist for the Xscale processor. As well as the fact that there may be certain instructions that do not require the use of a certain functional block(s); however, since most blocks (caches, buffers, and the instruction pipeline for example) are steadily consuming energy, the effect of variability of instruction functionality is eliminated for the most part. This result is to be expected since general-purpose processors are specifically designed for performance and versatility, which is achieved with the use of the buffers, caches, and the instruction decoding capability.

Finally this section concludes with a small note about instruction timings. There is a detailed table of instruction timings for Xscale in [17, Chapter 14]. As was explained earlier in this chapter, all of the instruction timings or latencies for the instructions explored were measured using run time and clock frequency. The CPI for all the situations explored was in agreement with the existing timing values. However, the tables in the noted reference for each instruction provide additional information such as inter-instruction effects and other issues that affect the cycles taken by the instruction. So this information will be helpful for estimation of the energy consumed for a complete program. As far as illustrating the point about relative invariance of the energy per cycle values, the data collected is more than sufficient. The energy values for other configurations that exist for some instructions (such as the multiply instructions) can be easily determined from the energy per cycle numbers and the instruction timing reference material.

2.6 Concluding Remarks

The Xscale processor was introduced with detailed setup procedures used for all experiments. The interface software program as well as the energy measurement strategy for the instructions was also outlined. Finally, this chapter has explored through the instruction set available to the Intel Xscale processor with extensive energy profiling of each instruction and accompanying addressing modes. Very comprehensive measurements and results were presented and an in depth analysis and interpretation of the results were made with illustrative plots. A total of 327 experiments were done to obtain a very comprehensive energy profile for the instructions with numerous other miscellaneous experiments. The most important conclusion to be drawn from all these measurements is that the energy per cycle is generally constant for groups of instructions, and this mainly due to fixed circuit overhead that exists for all instructions in a general-purpose microprocessor.

Chapter 3

Instruction Level Energy Profiling of the Intel StrongARM SA-1100 Processor

3.0 Introduction

The goal of this chapter is to become familiar with the Intel StrongARM SA-1100 processor and more importantly to get the instruction level energy profiling for the instruction set of this processor. The Brutus SA-1100 Design Verification Platform board is used, where the core is the Intel Strong ARM SA-1100 processor, ARM architecture v4.

As in chapter 2, each instruction is tested carefully to measure the energy consumed for that particular instruction and form. Very comprehensive measurements were made and an in depth analysis of the results is presented in this chapter with illustrative plots. This processor has only the standard ARM instructions set is available; however there are some instructions that are specifically unavailable to Brutus, as will be discussed later in the chapter. Therefore a total of 36 instructions were profiled with a total of 250 measurements to account for the different addressing mode forms. Other miscellaneous experiments were done as well to demonstrate various other results that are not accounted for in the above count.

3.1 Intel StrongARM SA-1100 Processor

The Intel StrongARM processor is a high performance processor that also has an impressive list of features [10, 18-23]. Figure 3.1 shows the photo of the Brutus SA-1100 Design Verification Platform. This processor is capable of running from 59 MHz to 206.4 MHz in increments of 14.7 or 14.8 MHz. The table below shows the clock frequencies and the corresponding voltage that should be applied. The voltage values tabulated are the minimum required voltage to be applied to the core at the corresponding clock frequencies. These minimum voltage values were determined by trial and error. The table also shows the code value to be used in the set up StrongARM file: setupARM.s (appendix B1). This assembly program enables the core frequency to be changed according the code given in Table 3.1, the command that enables core frequency change is shown in bold in appendix B1, further details are reserved for chapter 4. Note that only a maximum of 2 V can be applied to the core.

Table 3.1: Frequency and voltage table for the StrongARM SA-1100 core

Core Frequency	Minimum Voltage	Code
206.4 MHz	1.650 V	0A
191.7 MHz	1.500 V	09
176.9 MHz	1.400 V	08
162.2 MHz	1.300 V	07
147.5 MHz	1.235 V	06
132.7 MHz	1.175 V	05
118.0 MHz	1.127 V	04
103.2 MHz	1.055 V	03
88.5 MHz	0.950 V	02
73.7 MHz	0.900 V	01
59.0 MHz	0.877 V	00

Unlike Xscale, Brutus board has pins that make the core of the processor accessible for current measurements. Also it is possible to put Brutus on external supply mode with just a switch; therefore, the voltage applied to the core can be controlled as well. Details of on board connections and other setup procedures will be discussed in the following section.

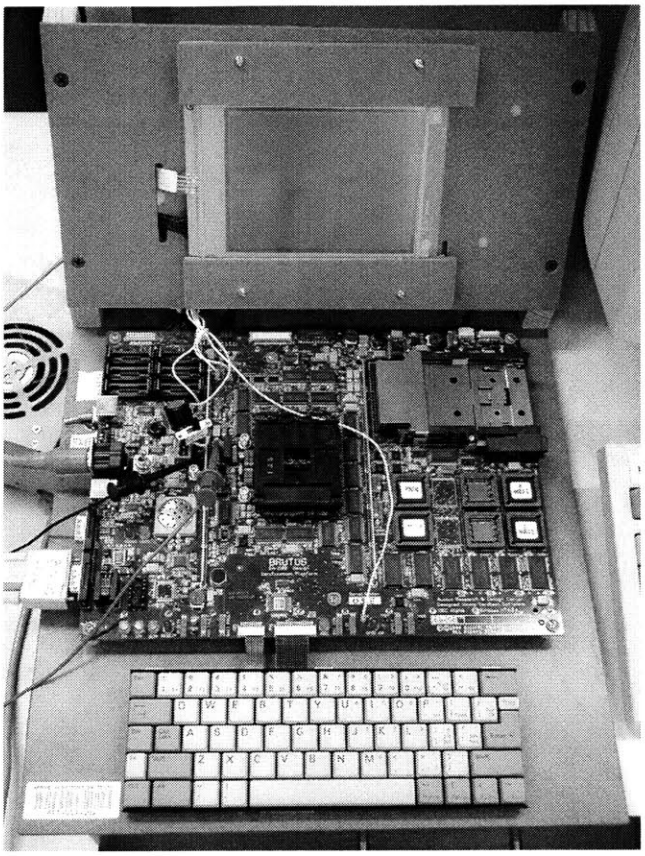

Figure 3.1: Brutus SA-1100 Design Verification Platform Board

3.2 The Setup of the Brutus Design Verification Platform

A very simplistic diagram of the overall system is shown below (Figure 3.2). This section describes the details of the overall setup, and then in the following section the details of the experiment procedure are outlined.

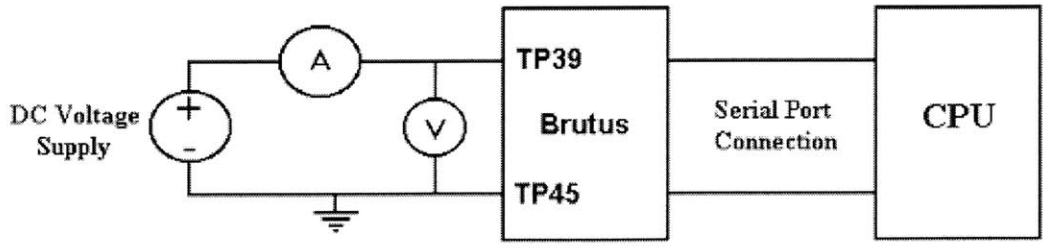

Figure 3.2: Diagram of the Setup of the Overall Measurement System

The setup of the Brutus board is slightly more complicated than Xscale; therefore, the steps are carefully explained below. The switch pointed by the arrow in Figure 3.3 must be set to external supply, as is in the photo. An arrow also points to the location where the gray fan box must be connected to on Brutus (pin J18). Brutus is connected to the computer's serial port via Brutus's own cable that should be connected to the J23 port on Brutus; an arrow also illustrates this. An arrow points to the reset switch that is needed frequently. There also an arrow that points to Brutus's power switch. Finally, the white arrows on the far right hand side of Brutus point to where the Angel chips are placed; this is done to be able to use the Angel Debugger Software program.

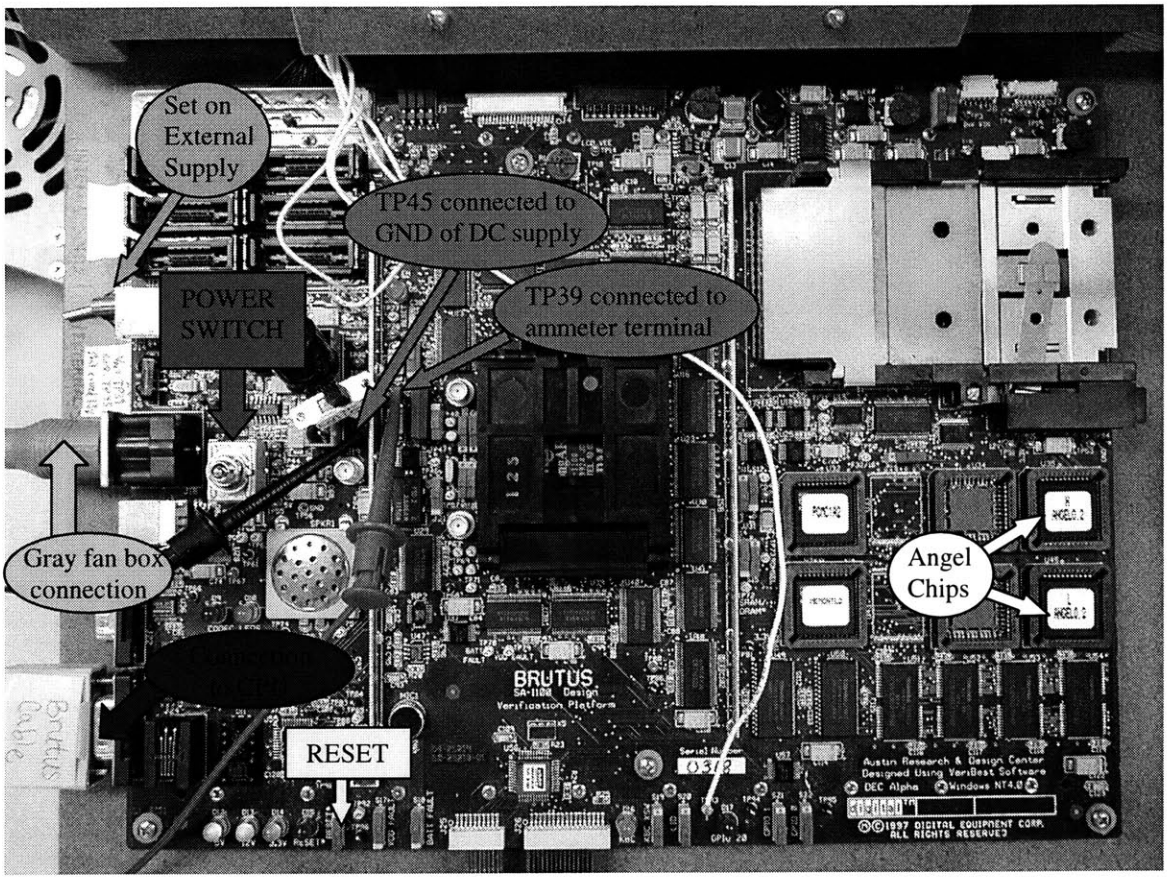

Figure 3.3: Close up of the Brutus board

3.3 Experimental Procedure

The procedure and methodology used in measuring the instruction level energy consumption is fundamentally identical to the method used for Xscale, and thus will not

be repeated here. The minor modifications that are made to adjust for StrongARM is mostly in the programming syntax. The entire program that does the instruction level energy profiling is approximately 650 pages, clearly too long to be included in the appendix. However, appendix B2 contains the main program without the 250 experiment segment blocks. The complete program in entirety in located in my MTL account at the following directory: ~mitra/MSThesis/StrongARM/Instruction_Profiling/test_inst.c. Also communication with Brutus is established through the CPU serial port, where the ARM SDT v2.11 [39-40] software package provides the software interface.

3.4 Results

As noted earlier, only the ARM standard instruction set is available to the StrongARM SA-1100 processor and even within the standard set of instructions there are some that are not available on Brutus [44-45]. Therefore, a total 7 categories and 36 instructions with a total of 250 various forms and addressing modes have been examined.

This section presents the summarized results for all the experiments to show that the hypothesis presented in chapter 2 also holds valid for the StrongARM processor. Unlike the previous chapter, where the measurements were presented in great detail for Xscale, the results for StrongARM will not be shown in such detail. This is due to the similarity in energy consumption patterns that exist between these two processors. Therefore only general observations with key points are presented. It is worth noting that there are some minor differences as far as cycles-taken per some addressing modes and so forth; however, elaboration about these minor differences does not provide additional insight, and thus is excluded. However, more specific charts including the addressing mode details for each instruction category is included in Appendix B3 for informative reference.

Figure 3.4 shows the summary of the energy profiling results for the seven categories of instructions explained earlier. In order to make a comparison between StrongARM and Xscale, Figure 3.5 shows the energy results of Xscale for the same instructions and

categories that are available on StrongARM. Both these charts are obtained through grouping and averaging over the energy consumed for the various addressing modes and in some cases instructions. There is no overlap in the frequency range of the two processors; however, Xscale can be operated at the frequency so that the core voltage can at least be the same for both processors. Thus the voltage is fixed at approximately 1.5 V for both processors. Thus StrongARM operates at 191.7 MHz and Xscale operates at 733 MHz. One immediate conclusion to draw from these two charts is that for most cases StrongARM consumes greater energy than Xscale. Table 3.2 below shows the energy consumption values for both processors. The first and second group of columns of the table provide the measurement results for StrongARM and Xscale respectively. The last group of columns provide the comparison results, where the numbers listed represent the factor by which StrongARM consumes energy in comparison to Xscale.

Analyzing the results presented in Table 3.2, leads to the following interesting points. For the cases where the CPI is the same for both processors, StrongARM consumes from 1.65 to 2 times more energy per cycle than Xscale. There are also instances where the CPI is larger for StrongARM, which of course is clear that it will consume more energy than Xscale. However, the interesting and final case to consider, is the case where the CPI for StrongARM is less than Xscale. Now the question becomes will StrongARM be more energy efficient for such instructions? For the load and semaphore instructions the answer is yes, even though per cycle StrongARM consumes more energy, the fact that it takes fewer cycles to execute the instruction results in the energy per instruction to be .73 times less than the energy per instruction of Xscale. Likewise for the store, load multiple, and store multiple instructions the energy per cycle is always higher for StrongARM; however fewer cycles result in the energy per instruction in StrongARM to be either the same or only slightly higher than the energy per instruction of Xscale.

However, since StrongARM is energy inefficient in all other cases, Xscale is the processor of choice for energy efficiency and also performance. The fact that Xscale takes extra cycles in some instances to compute an instruction does not give it a disadvantage over StrongARM for computation speed. All measurements for Xscale were

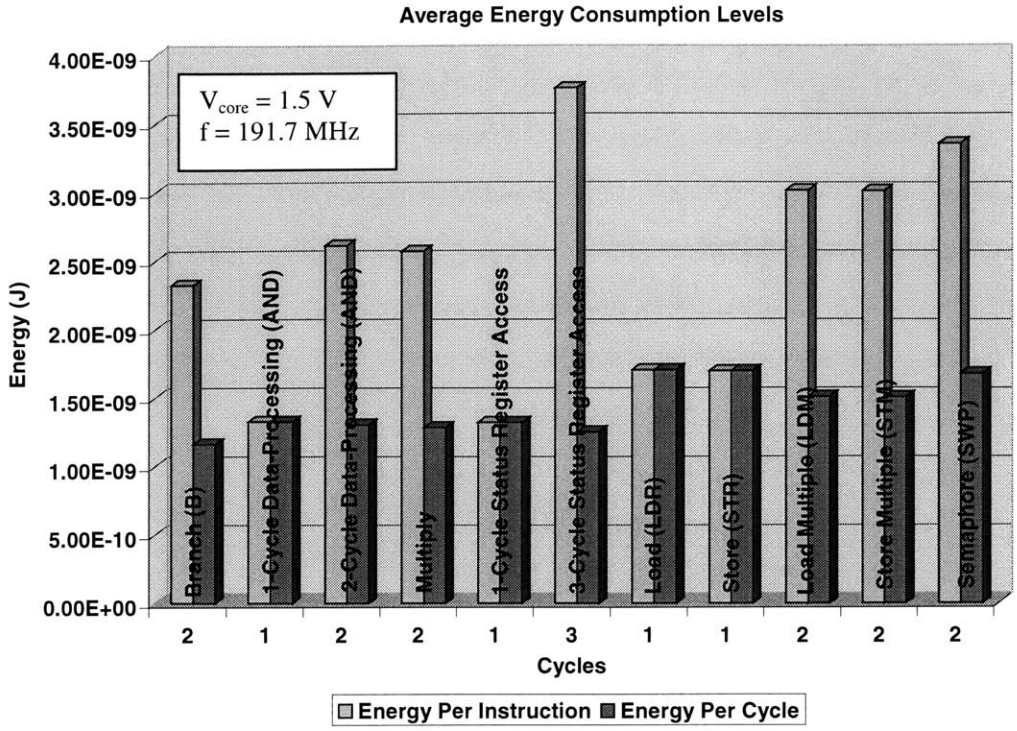

Figure 3.4: Average energy consumption levels for StrongARM

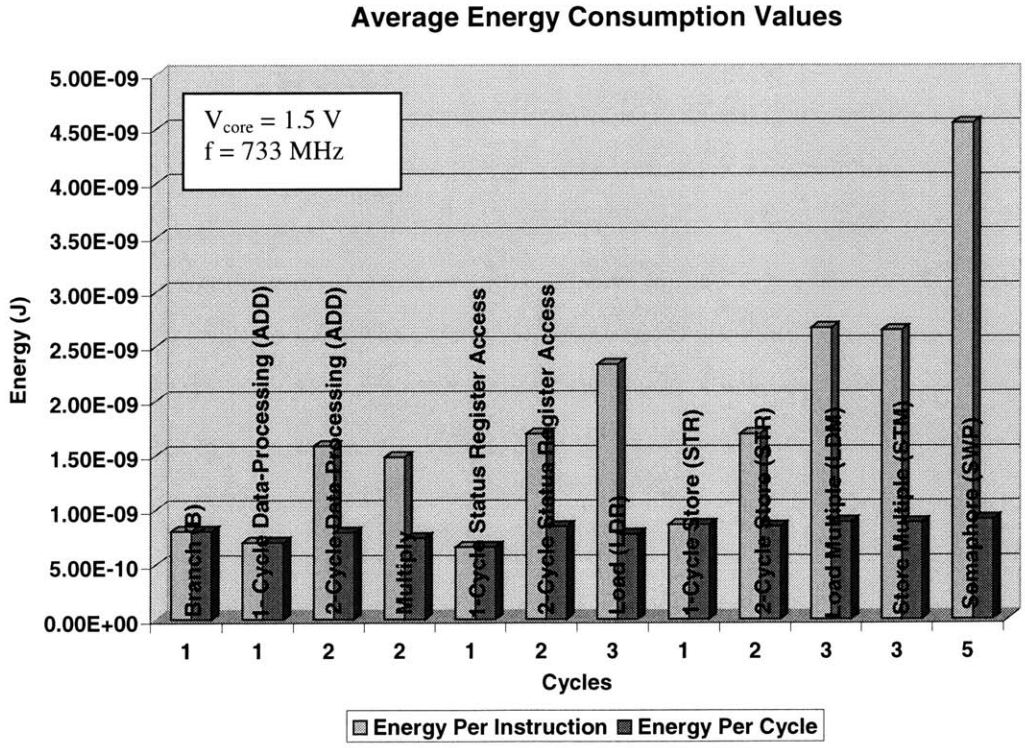

Figure 3.5: Average energy consumption levels for Xscale

at a core operating frequency of 733 MHz, whereas the measurements for StrongARM were at a core operating frequency of 191.7 MHz. Therefore, Xscale is not only more energy efficient, but also significantly faster than StrongARM. To illustrate this point further, the switching capacitance per cycle for each processor is calculated for each instruction category. This is calculated by making a simplifying assumption that is explained and justified below. The value of energy consumed per instruction that is evaluated is the total energy consumed by the processor during execution of that particular instruction. This is defined by the equation below [1]:

$$E_{total} = P_{total}\, T_{inst} = C_{total}\, V_{dd}^{2} + V_{dd}\, I_{leak}\, T_{inst}$$

The first term is the dynamic component, with C_{total} being the switching capacitance and V the applied core voltage. The second term is the leakage component and will be ignored for the purposes of this comparison. However leakage current will be explored in

Table 3.2: Summary of Instruction Statistics for StrongARM vs. Xscale

		StrongARM f = 191.7 MHz			**Xscale f = 733 MHz**			**Comparison of StrongARM vs. Xscale**	
Inst.	V (V)	E_{inst} (nJ)	E_{cycle} (nJ)	**CPI**	E_{inst} (nJ)	E_{cycle} (nJ)	**CPI**	**E_{inst} (factor)**	**E_{cycle} (factor)**
Branch	1.44	2.33	1.16	**2**	.80	.80	**1**	**2.91**	**1.45**
Data-Processing	1.46	1.33	1.33	**1**	.70	.70	**1**	**1.90**	**1.90**
Data-Processing	1.46	2.61	1.31	**2**	1.58	.79	**2**	**1.65**	**1.65**
Multiply	1.43	2.57	1.29	**2**	1.48	.74	**2**	**1.74**	**1.74**
Status Register Access	1.46	1.32	1.32	**1**	.66	.66	**1**	**2**	**2**
Status Register Access	1.46	3.76	1.25	**3**	1.69	.85	**2**	**2.22**	**1.47**
Load	1.41	1.71	1.71	**1**	2.33	.78	**3**	**.73**	**2.19**
Store	1.41	1.7	1.7	**1**	.86	.86	**1**	**1.98**	**1.98**
Store	1.41	-	-	**-**	1.69	.85	**2**	**1.01**	**2**
Load Multiple	1.42	3.01	1.51	**2**	2.67	.89	**3**	**1.13**	**1.70**
Store Multiple	1.42	3.01	1.51	**2**	2.64	.88	**3**	**1.14**	**1.72**
Semaphore	1.41	3.36	1.68	**2**	4.55	.91	**5**	**.74**	**1.85**

greater detail in chapter 4. Ignoring the leakage component is justified since the switching energy is the dominating factor, with a contribution of 90% to the total energy. Leakage energy counts for only 10% of the total energy consumption [25] it becomes a significant source of energy consumption during idling modes. Therefore, for comparison purposes it is safe to ignore the leakage component of energy consumption and concentrate on the dynamic component. Therefore the approximation to the switching capacitance is tabulated in Table 3.3 below, where the last column in the table shows the factor by which StrongARM has greater switching capacitance per cycle.

Table 3.3: Switching Capacitance Per Cycle Statistics for StrongARM and Xscale

	StrongARM		**Xscale**		**Comparison**
Inst.	C_{total} (nF)	CPI	C_{total} (nF)	CPI	**C_{total} (factor)**
Branch	.559	2	.386	1	**1.45**
Data-Processing	.624	1	.328	1	**1.90**
Data-Processing	.615	2	.371	2	**1.66**
Multiply	.631	2	.362	2	**1.74**
Status Register Access	.619	1	.31	1	**2**
Status Register Access	.586	3	.399	2	**1.47**
Load	.86	1	.392	3	**2.19**
Store	.855	1	.433	1	**1.98**
Store	-	-	.428	2	**2**
Load Multiple	.749	2	.441	3	**1.7**
Store Multiple	.749	2	.436	3	**1.72**
Semaphore	.845	2	.458	5	**1.8**

From all these comparisons with Xscale for the ARM instruction set we can conclude that Xscale is more energy efficient and is significantly faster then StrongARM. The improvement of Xscale can be partially attributed to technology scaling. Since a .18 micron process is used for Xscale [46] as opposed to a .35 micron process for StrongARM [47]. The energy efficiency and superior performance of Xscale over StrongARM has been explored in detail in this chapter on an instruction level basis. In the next chapter, comparisons based on a benchmark program will be used to quantify and evaluate the performance and efficiency of the two processors.

3.5 Analysis of Results

In section 2.5 of chapter 2, the results of the intensive energy profiling were analyzed for Xscale. In addition, a hypothesis was presented about the patterns of energy consumption. This same hypothesis can be validated for the StrongARM processor, as will be shown below. Some general observations can be made from the statistics in Figure 3.6 below. It is apparent that most of the energy values lie within four distinct regions, 116 instructions consume energy in the range of 1.3 – 1.5 nJ; the second largest region is in the range of 2.6 – 2.8 nJ, where there are 64 instructions. Again as is expected load and store instructions draw more current and thus consume somewhat more energy, as a result 49 instructions lie in the range of 1.8 – 1.9 nJ. Lastly 22 instructions seem to be scattered about and consuming energy within the range of 3 – 3.5 nJ. Even though the load and store instructions draw greater current, they are nevertheless single cycle instructions and likewise the 20 instructions that consume energy above 3 nJ are in majority 2 cycle instructions. Figure 3.7 below shows the energy consumed on a per cycle basis, where we see that it is relatively flat, which agrees with the hypothesis. Most values are scattered within the range of 1.3 – 1.5 nJ, with some scattered around 1.8 nJ.

Greater insight can be gained into how the processor consumes energy from the energy consumption versus cycles taken plot shown in Figure 3.8. It is clear that 2-cycle instructions take roughly twice the energy as the single cycle instructions. The relationship between cycles taken by instruction and corresponding energy consumption can be more clearly understood, by looking at the energy consumed per cycle versus cycles taken as shown in Figure 3.9. The energy consumed per cycle is fairly constant with only minor variations. Some exceptions are the apparent discrepancies in the single cycle instructions (Figure 3.9, section 2); that seem to take more energy than the others. There are also to a lesser extent some discrepancies in the 2-cycle instructions that consume somewhat more energy than most 2-cycle instructions (Figure 3.9, section 4). These exceptions are mainly due the load and store instructions. The plot of the energy per cycle is divided in subsections (Figure 3.9), sections 1 and 2 are for 1-cycle instructions and they have an average energy per cycle of 1.35 nJ with only an 8.6%

variability and 1.7 nJ with 4.8% variability respectively. Sections 3 and 4 are for 2-cycle instructions and they have an average energy per cycle of 1.32 nJ with a variability of only .5% and 1.51 nJ with 15% variability respectively.

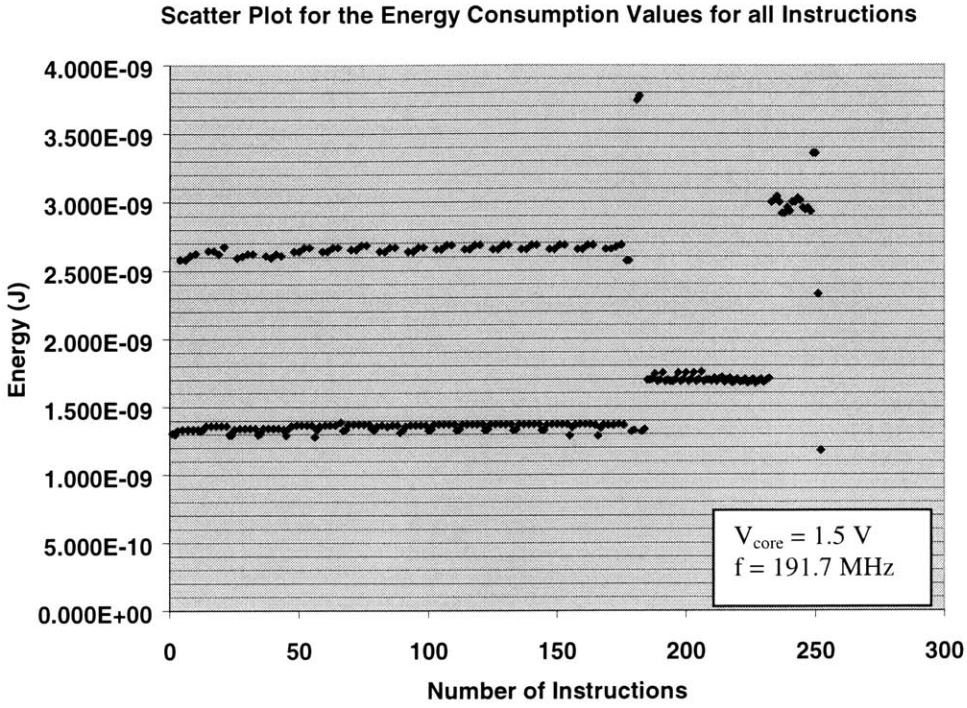

Figure 3.6: Scatter plot for energy consumption values of all instructions

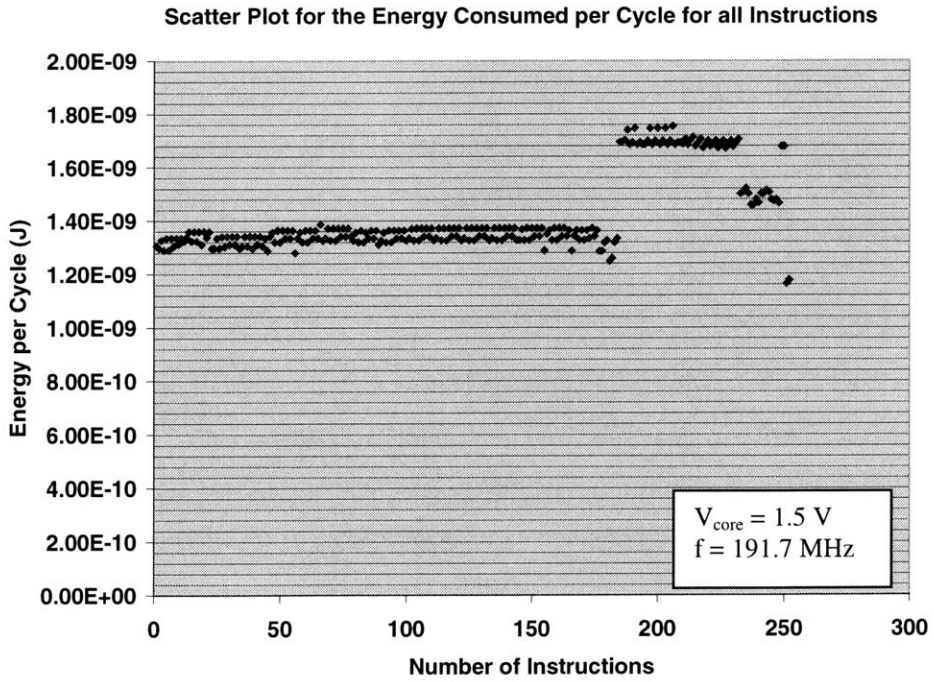

Figure 3.7: Scatter plot for energy consumption values per cycle for all instructions

Again the same patterns that were observed for Xscale hold true to a greater extent for StrongARM. As noted earlier there were some extreme cases for Xscale, were the energy per cycle for a few instructions were up to 1.7 times greater than the least energy consuming instructions. However, exceptions to this degree do not exist for StrongARM, thus suggesting that StrongARM has greater adherence to the proposed theory. These results enforce the concept presented that the energy per cycle will be for the most part constant regardless of instruction functionality. The reason for this is the same as for the Xscale processor; circuit overhead is mainly the same for all instructions with only minor dependency on functionality (case in point load and store instructions which require external memory access). As shown in [48] 95% of the total energy consumed for running an instruction on the StrongARM goes to powering the cache, control, global clock and I/O circuits. This leaves only 5% of the overall energy for the specific instruction, which of course is a very small portion of the total energy. Therefore, the independence of the energy consumed per cycle on the particular function of the instruction is justifiable.

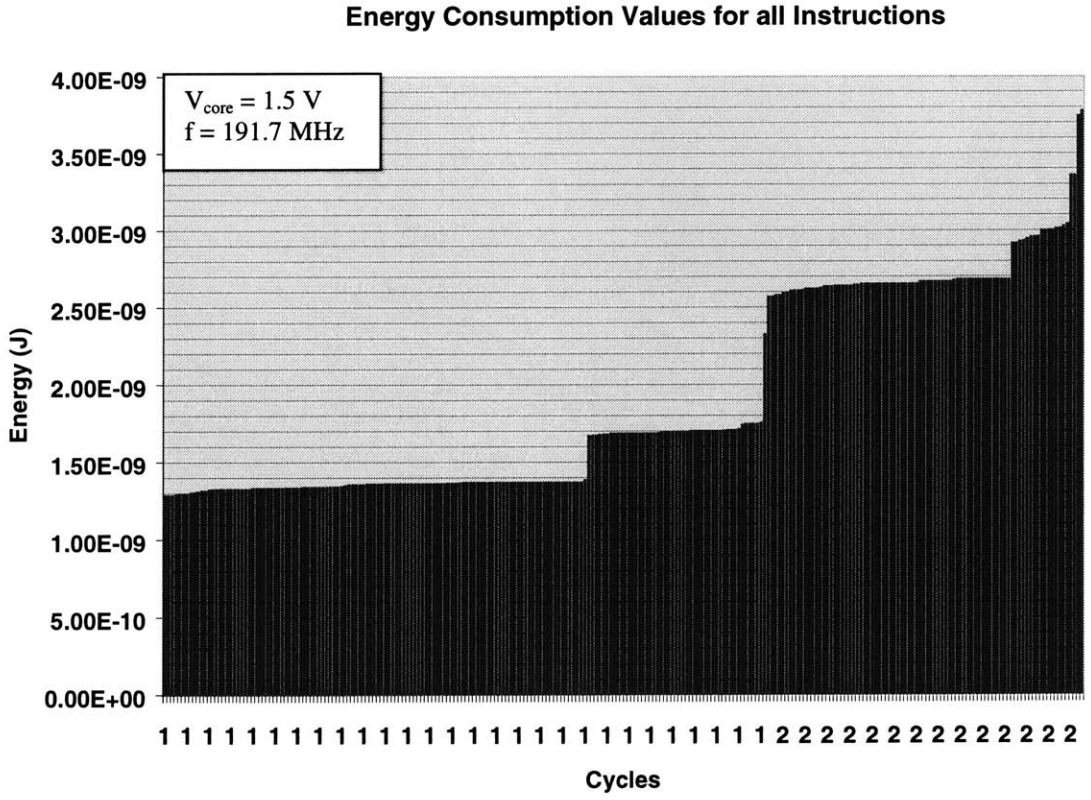

Figure 3.8: Plot of the energy per instruction versus cycles taken for all instructions

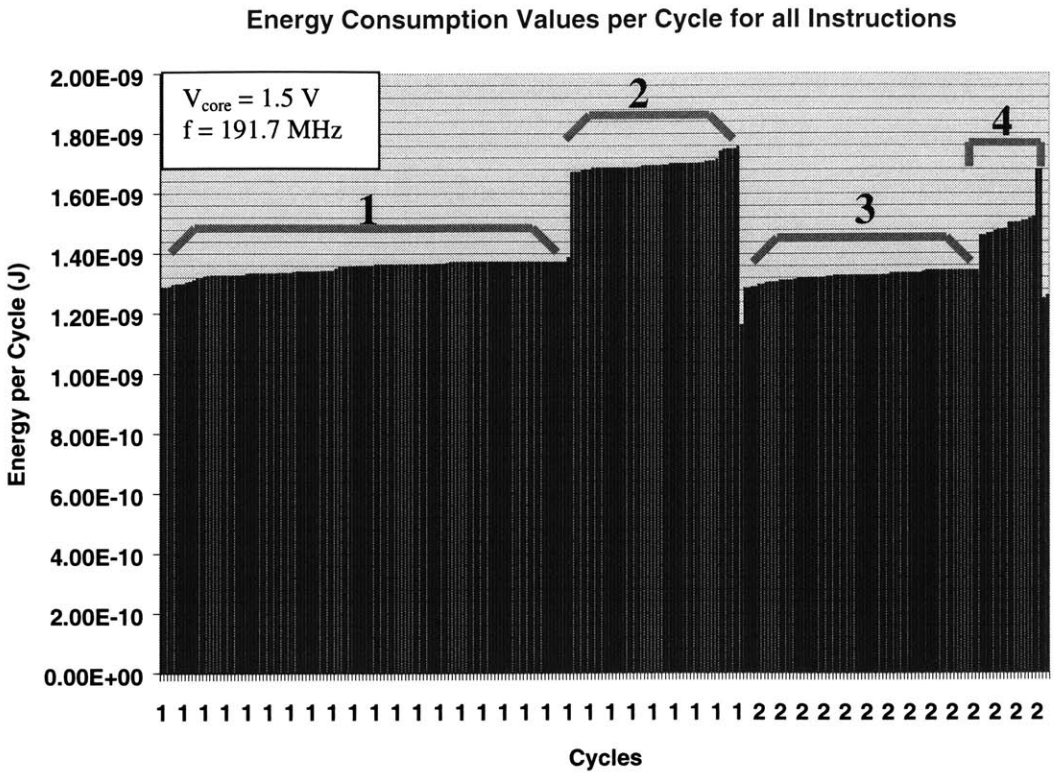

Figure 3.9: Plot of the energy per cycle versus cycles taken for all instructions

Chapter 4

Static Energy Consumption

4.0 Introduction

Up to this point emphasis has been placed on obtaining the instruction energy profiling of the Xscale and StrongARM processors. The first portion of this chapter attempts to draw together the important insights gained so far by returning with answers to some of the questions posed in chapter 1, and then those concepts are applied to a benchmark program. The second part of the chapter deals with the static energy consumption and idle modes of the processors.

One of questions relates to how a program can be designed to be more energy efficient with the knowledge of instruction energy profiling. One of the most important conclusions drawn from the results of the instruction energy profiling is that the variation of energy consumption per cycle across most of the variety of instructions is small. This simple model of instruction energy consumption, not only makes production of energy optimal code certainly feasible, but also considerably easy. Most performance optimization strategies already involve reduction in code size, which directly leads to reduction in cycles taken for the processor to execute that particular program; this results in energy reduction for the program.

This point is illustrated effectively with an 11-tap FIR filter [49] run on both the Xscale and StrongARM processors. Initially the benchmark is operated at a fixed voltage and frequency, with the same parameter values that were used in the instruction profiling. However, as mentioned in earlier chapters, each processor is capable of operating at a

wide range of clock frequencies and core voltages. Therefore, the effect of varying clock frequency and core voltage on energy consumption is analyzed for both processors.

4.1 Energy Optimization Example

This section goes through an analysis of applying the above suggested optimization technique to an 11-tap FIR filter. First the optimization is done on the Xscale processor and then on the StrongARM processor. Again the voltage and clock frequency used for both processors is the same that was used in chapters 2 and 3 for the instruction energy profiling.

The optimization results shown in the following sections are only illustrating one optimization point and that is the multiply-accumulate portion of the FIR filter. The filter for both processors is entirely optimized as far as minimal instructions and cycles except for the multiply-accumulate segment were the focus of this section is placed. As the multiply-accumulate is a very commonly used in digital signal processing programs, it is worthwhile to place emphasis on it.

4.1.1 Energy Optimization of an FIR Filter on the Xscale Processor

A 11-tap FIR filter was implemented in assembly and embedded into the main C program that was used for all experiments (included in appendix A3). The assembly code for the filter is included in appendix C for reference. Note that the program begins by writing some coefficients into memory, all of this is done outside of the 200, 000 iteration loop, so the time consumed for this operation is negligible. The filter instructions are as a group repeated 30 times to compensate for loop effects. Thirty times was considered sufficient for the filter program as opposed to 100 times of repetition, which was used for single instructions measurements. Since each one of the thirty filter program sections execute an approximate number of 450 instructions (this number takes into account inner looping of a smaller set of instructions), therefore the outer loop effects are indeed compensated. This filter has been optimized to consume minimal cycles, except for the multiply-

accumulate segment, shown in bold in appendix C. Below shows the two original instructions used to do the multiply-accumulate operation.

```
mul r9, r7, r8          /* multiply-accumulate segment*/
add r6, r6, r9
```

Clearly there is a more optimal way of implementing the same function; one way is to use the MLA instruction instead of a separate ADD and MUL instruction, as shown below.

```
mla r6, r7, r8, r6
```

As pointed out in chapter 2, this will clearly provide energy savings, since the MLA instruction takes 2 cycles to do the function of the MUL and ADD instructions, which take a combined total of 3 cycles to execute. There is however another alternative, and that is to use the MIA instruction. If this instruction is used, greater modifications need to take place. A simple replacement of the multiply-accumulate instructions is not sufficient, since the MIA instruction uses the internal 40-bit accumulator. Since the MIA instruction uses the accumulator as opposed to simple registers will cause additional overhead. Since accumulator access instructions are needed, this alternative becomes energy inefficient if the result of the multiply-accumulate is less than 40 bits.

The original program is modified in the following way, to make use of the MIA instruction. Since the alterations to the original program were only minor, the whole optimized program is not reiterated. Appendix C shows the relevant instructions in bold, so only those instructions or the location in the program that modifications are required are mentioned here. The beginning of the program initializes the sum, which in both the original and first optimized form use a register to store the result:

```
mov r6,#0           /* initialize sum */
```

Therefore, this instruction must be replaced with an accumulator initialization instruction so to use the MIA instruction as shown below:

```
mar acc0,r5,r6      /* initialize accumulator */
```

The next step is to replace the separate MUL and ADD instructions of the original program shown above with the MIA instruction shown below:

```
mia acc0,r7,r8          /* multiply-accumulate into 40-bit accumulator */
```

Then to store the result of the multiply-accumulate from the accumulator to memory, an accumulator access instruction must be inserted before the store into memory instruction (also shown in bold in appendix C), as shown below.

```
mra r5,r6,acc0          /* move result from accumulator to register */
str r5,[r10,#0]         /* Rewrites back to memory*/
```

Using the following instruction the sum is reset at the end, before the loop is iterated again.

```
mov r6,#0               /* initialize sum */
```

Therefore this instruction needs to be replaced with the instruction to reset the accumulator as shown below:

```
mar acc0,r5,r6          /* reset accumulator */
```

The results of the above mention optimizations are illustrated in Table 4.1 below. Where FIR filter is the original un-optimized program, FIR filter optimized 1 is the first form of optimization that uses the MLA instruction, and FIR filter optimized 2 is the second form of optimization that was just described in detail.

Table 4.1: Results for the 11-tap FIR filter and optimized forms (Xscale)

	T_{fir} (sec)	E_{fir} (J)	T_{run} (sec)	N_{rept}	N_{iter}	Current (A)	Voltage (V)	Frequency (MHz)
FIR Filter Original: Using MUL and ADD instructions	9.333E-06	2.595E-06	56.00	30	2.E+05	0.263	1.057	533
FIR Filter Optimized 1: Using the MLA instruction	6.000E-06	1.818E-06	36.00	30	2.E+05	0.29	1.045	533
FIR Filter Optimized 2: Using the MIA instruction	6.167E-06	1.899E-06	37.00	30	2.E+05	0.295	1.044	533

From Table 4.1 we see that the original implementation of the filter using separate multiply and add instructions consumes 42.7% more energy than if the filter were to be implemented using the MLA instruction (FIR Filter Optimized 1). On the other hand the FIR Filter Optimized 2 program approximately consumes the same amount of energy as the FIR Filter Optimized 1 program. Therefore, unless the expected result of accumulation is greater than 32-bits, using the simpler MLA instruction is sufficient for an energy optimal program. Otherwise, if the result will require 40 bits, the MIA instruction is optimal.

Taking a closer look into the instructions used to optimize the filter reveals an interesting point. It is obvious that replacing the MUL and ADD instructions with the single MLA instruction would yield energy savings (since the MLA instruction is a 2 cycle instruction while the MUL and ADD instructions together take 3 cycles). However, the result for the second optimization is not obvious, since the MIA instruction takes only 1 cycle to perform the multiply-accumulate (Figure 4.1). This is desirable, as it not only performs the multiply-accumulate function, but also the accumulator can store up to 40 bits. The accumulator access instructions, MAR and MRA are however energy expensive, since they take 2 and 3 cycles to execute respectively (Figure 4.1). Thus the overall program energy efficiency is clearly dependent on the frequency at which the accumulator is accessed. In this particular example, the result turns out to be very close to the simple register using MLA instruction optimization form. Clearly for other programs, results will vary depending on the use of the register access instructions.

One way to quantitatively represent performance versus energy is fixing the run time while varying the voltage and frequency. This is done to the original FIR filter that used separate multiply and add instructions (MUL and ADD) and FIR Filter Optimized 1 which used the MLA multiply-accumulate instruction. Since it is concluded that the most energy optimal program is FIR Filter Optimized 1. Table 4.2 shows the result of this experiment, where the frequency is changed to 733 MHz for the original filter accompanied by an increase in voltage to 1.277 V to make the run time of this filter comparable to the run time of the optimized filter (as much as practically possible). Thus

with the performance of both filters now comparable, energy efficiency could be addressed more fairly. Therefore it can be concluded that the Optimized filter is 51% more energy efficient than the original filter.

Table 4.2: Performance versus energy for the 11-tap FIR Filter (Xscale)

	T_{fir} (sec)	E_{fir} (J)	T_{run} (sec)	N_{rept}	N_{iter}	Current (A)	Voltage (V)	Frequency (MHz)
FIR Filter Original: Using MUL and ADD instructions	6.667E-06	3.703E-06	40.00	30	2.E+05	0.435	1.277	733
FIR Filter Optimized1: Using the MLA instruction	6.000E-06	1.818E-06	36.00	30	2.E+05	0.29	1.045	533

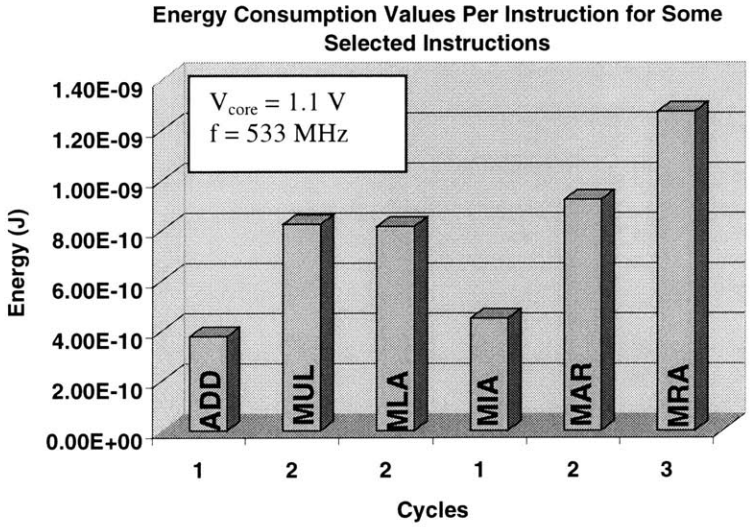

Figure 4.1: Energy consumption values for some instructions (Xscale)

4.1.2 Energy Optimization of an FIR Filter on the StrongARM Processor

The same 11-tap FIR filter that was used for Xscale is also used for StrongARM with some syntax modifications. However, the filter segment implemented in assembly is be repeated 20 times within the main 100,000 iteration loop, to compensate for loop effects. Since this program is the same as the one used for Xscale, except for some minor syntax modifications, it has the same characteristics mentioned previously. Therefore, only the multiply-accumulate segment will be optimized, shown for Xscale in bold in appendix C.

Note that for StrongARM there is only one optimization possibility available, the use of the MLA instruction instead of the separate MUL and ADD instructions.

Table 4.3 below shows the results of the energy measurements for both the original filter and the optimized filter. The original filter consumes only 18.7% more energy than the optimized filter program. Whereas, the same optimization on the same program was done for Xscale, and it was concluded that for Xscale the original program consumed 42.7% more energy than the optimized program. This is because, the MLA instruction takes 2 cycles to execute on StrongARM, whereas the MUL and ADD instructions together consume 3 cycles (Figure 4.2).

Table 4.3: Results for the 11-tap FIR filter and optimized forms (StrongARM)

	T_{inst} (sec)	E_{inst} (J)	T_{run} (sec)	N_{inst}	N_{iter}	Current (A)	Voltage (V)	Frequency (MHz)
FIR Filter Original: Using MUL and ADD instructions	2.300E-05	7.195E-06	46.00	20	1.E+05	0.215	1.455	191.7
FIR Filter Optimized1: Using the MLA instruction	1.850E-05	6.062E-06	37.00	20	1.E+05	0.226	1.45	191.7

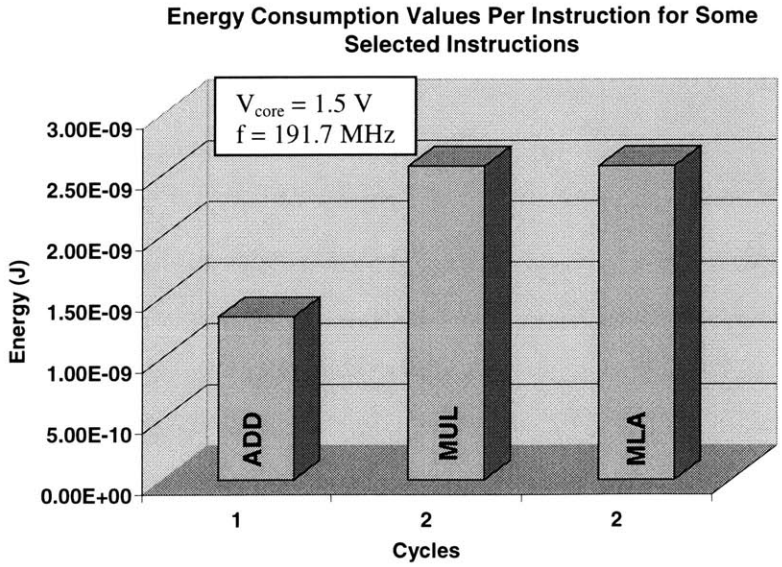

Figure 4.2: Energy consumption values for some instructions (StrongARM)

4.2 Clock Frequency Variation Considerations

Up until this point both processors have been analyzed at a fixed clock frequency and corresponding core voltage. This section attempts to take the analysis further by considering energy consumption at other processor frequencies using the same benchmark program used earlier. For each processor the most energy optimal program as determined by the analysis in the previous section is used. The measurement procedure is the same as was outlined in chapters 2 and 3 for processors Xscale and StrongARM respectively. However, the procedure for changing the processor frequency is briefly discussed below for each processor, followed by the discussion of the results of testing the benchmark program at various frequencies. Finally the two processors are again compared.

4.2.1 Changing Frequency on the Xscale Processor

Changing the frequency on this processor is simply accomplished through the setup commands at the start of the main program shown in appendix A3. The specific line of - code that must be changed to achieve the desired processor operating frequency is shown in bold in appendix A3, as well as included here below:

```
#define PLL_MULTIPLIER 6          // Clock frequency  = 533 MHz
```

The value defined for the PLL multiplier determines the frequency. Table 2.2 lists the values of the PLL multiplier and the frequency that it corresponds to. Therefore, simply changing this value in the main program to the value that corresponds to the desired operating frequency sets the clock frequency to the desired frequency.

4.2.2 Changing Frequency on the StrongARM SA-1100 Processor

Using the codes given in Table 3.1 for each corresponding frequency changes the processor frequency. This code must be placed in the setupARM.s file in the immediate number position of the following instruction, also shown in bold in appendix B1.

```
ORR          r1, r1, #0x09          ; Sets the frequency to 191.7 MHz
```

Table 3.1 also shows the corresponding core voltage that should be applied for each frequency.

4.2.3 Results of Frequency Variations Xscale and StrongARM

Two sets of measurements were made for both processors. For the first one, both the frequency and core voltage are varied, while for the second set of measurements the core voltage is kept at a constant maximum value and the frequency is sequentially lowered. For the Xscale processor the four operable frequencies are tested and for the StrongARM processor all eleven frequency possibilities are tested. The FIR filter program used for these measurements for each processor is the energy optimal filter program from section 4.1. Figure 4.3 below shows the results of the measurements for both processors on the same plot. This Figure also supports the fact that Xscale consumes considerably less energy than the StrongARM processor. Table 4.4 shows statistics about energy consumption and performance for both processors for six data points that use roughly the same core voltage for both processors. Thus the comparison will be a fair one, since the core voltage parameter is fixed. The last two columns in the table show the factor by which StrongARM consumes more energy than Xscale, and the factor by which StrongARM operates slower than Xscale respectively. The results presented in the table below validate that Xscale is a superior processor over StrongARM in terms of energy consumption and performance. Over the voltage range given in Table 4.5, on average StrongARM consumes 1.71 times more energy than Xscale and is 4.89 times slower. These results make Xscale undoubtedly the processor of choice.

Table 4.5: Energy and Performance Statistics for StrongARM and Xscale

	StrongARM		Xscale		StrongARM	
Core Voltage	E_{FIR}	Operating Frequency	E_{FIR}	Operating Frequency	MORE E_{FIR} (factor)	SLOWER (factor)
0.875 V	2.32 µJ	59 MHz	1.28 µJ	400 MHz	1.81	6.78
0.950 V	2.63 µJ	88.5 MHz	1.51 µJ	466 MHz	1.74	5.27
1.100 V	3.59 µJ	118 MHz	2.03 µJ	533 MHz	1.77	4.52
1.200 V	3.90 µJ	132.7 MHz	2.42 µJ	600 MHz	1.61	4.52
1.300 V	4.73 µJ	162.2 MHz	2.87 µJ	666 MHz	1.65	4.11
1.400 V	5.55 µJ	176.9 MHz	3.30 µJ	733 MHz	1.68	4.14

The next discussion involves the second set of measurements, which the voltage was kept fixed for the various operating frequencies for both processors. This set of measurements is used to determine the static component of energy consumption. The energy that is measured and plotted is actually the total energy consumed by the processor for the execution of a particular program. This total energy includes both the dynamic and static energy [1]:

$$E_{total} = E_{dynamic} + E_{static} = C_{total} V^2_{core} + V_{core} I_{leak} t_{run}$$

Where C_{total} is the total switching capacitance, V_{core} is the applied core voltage, I_{leak} is the leakage current, and t_{run} is the run time for the execution of the program. Therefore, it is clear that as the core voltage is increased the dynamic energy consumption will increase quadratically with voltage. Since dynamic energy consumption is the dominating factor in the total energy consumption and the total switching capacitance does not change with clock frequency [1], the quadratic increase in energy consumption with increasing voltage is apparent from Figure 4.3 for both processors.

Since the core voltage is kept approximately constant, the dynamic energy consumption will be constant. The only variation will be due to the leakage current. Thus the static component of energy consumption can be determined, since the slope of the curve will directly reflect the leakage current [1]. The leakage energy model and current measurements have already been previously done for StrongARM [1]. A brief summary of the leakage results for StrongARM that were done for the same voltages as Xscale are included here for comparison in the next section.

Table 4.6: Leakage Current for StrongARM [1]

	Leakage Current		
Core Voltage	Measured	Model	% Error
1.4 V	16.35 mA	16.65 mA	-1.84%
1.3 V	13.26 mA	13.8 mA	-4.04%
1.2 V	12.07 mA	11.43 mA	5.27%
1.1 V	9.39 mA	9.47 mA	-0.87%
1.0 V	7.96 mA	7.85 mA	1.40%
0.90 V	6.39 mA	6.53 mA	-1.70%

It is shown in [1] that a very simplified model for leakage current is sufficient and accurately represents that leakage current for the StrongARM processor. This same model shown below is also attempted to be verified to hold true for Xscale as well in the following section.

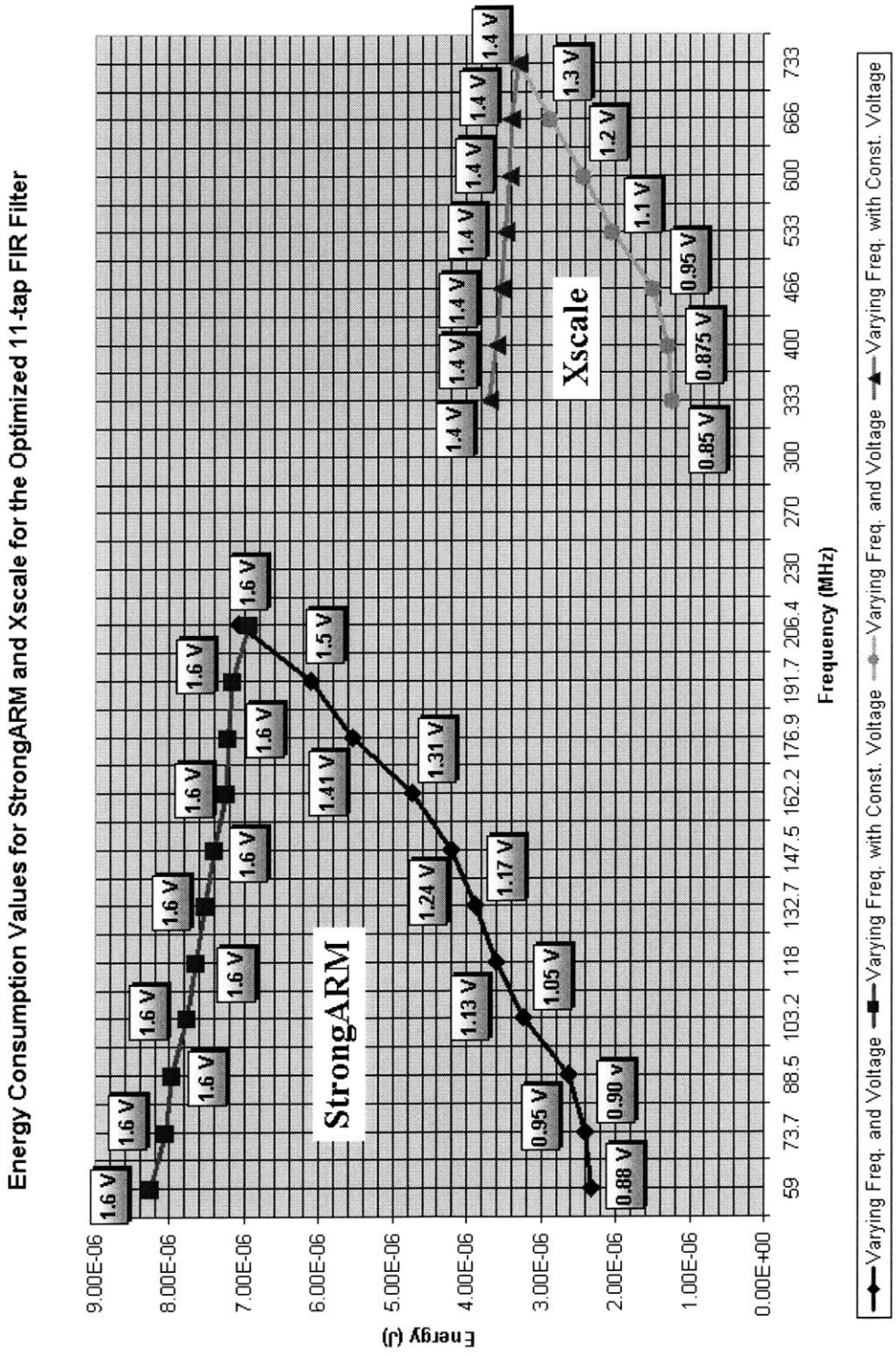

Figure 4.3: Energy consumption for various frequencies (Xscale and StrongARM)

Leakage Current Model for the StrongARM Processor [1]	
$I_{leak} = I_0 e^{\frac{V_{core}}{n'V_T}}$	Parameters: I_0 = 1.196 mA, n' = 21.26, and V_T = kT/q = .025 V Where the thermal voltage is calculated at room temperature.

4.3 Static Energy Consumption

This section presents the leakage current results for the Xscale processor and verifies that it adheres to the same leakage model proposed for StrongARM [1], which was shown in the previous section. In the previous section it was shown that the leakage current can be easily determined from the slope of the energy vs. run-time curve provided the core voltage is held constant. If the energy is normalized by the core voltage, charge is obtained; therefore, the slope of the charge vs. run-time curve exactly gives the leakage current. Figure 4.4 below shows the chart for charge versus run-time for the optimized FIR filter program. The data points for each of the three curves are determined by running Xscale at the constant core voltage values shown on the right hand side of each corresponding curve. The dashed lines on each curve are the linear fit to each curve, and as shown in the Figure it is an almost exact fit. Therefore, the slopes represent the leakage current at each of those corresponding voltages. The values of the leakage current are tabulated in Table 4.7 below, where the comparison is made with the results of the leakage model and the results clearly show that Xscale indeed does adhere to the leakage model shown earlier. The parameter values for Xscale are: I_0 = 2.116 mA and n = 17.466.

Table 4.7: Leakage Current for Xscale

	Leakage Current		
Core Voltage	Measured	Model	% Error
1.40 V	53.23 mA	52.24 mA	1.87%
1.30 V	39.74 mA	41.54 mA	-4.54%
1.20V	33.42 mA	33.04 mA	1.14%
1.10 V	26.71 mA	26.28 mA	1.62%
0.95 V	19.13 mA	18.64 mA	2.57%
0.875 V	15.28 mA	15.70 mA	-2.72%

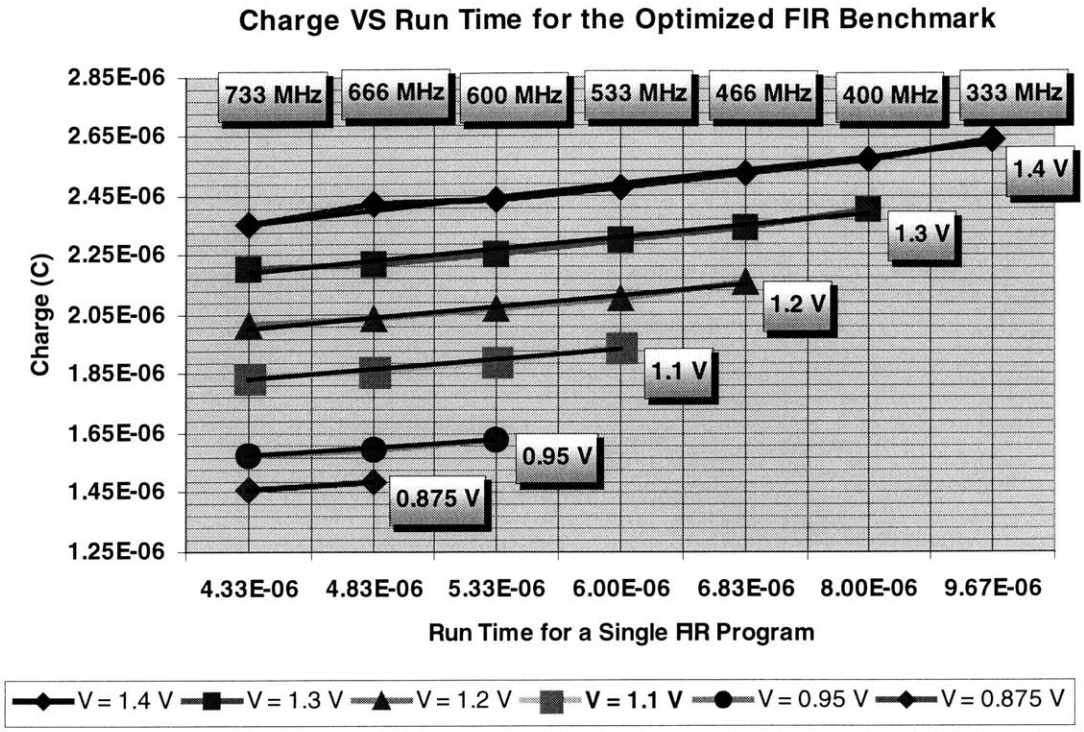

Figure 4.4: Charge versus run-time (Xscale)

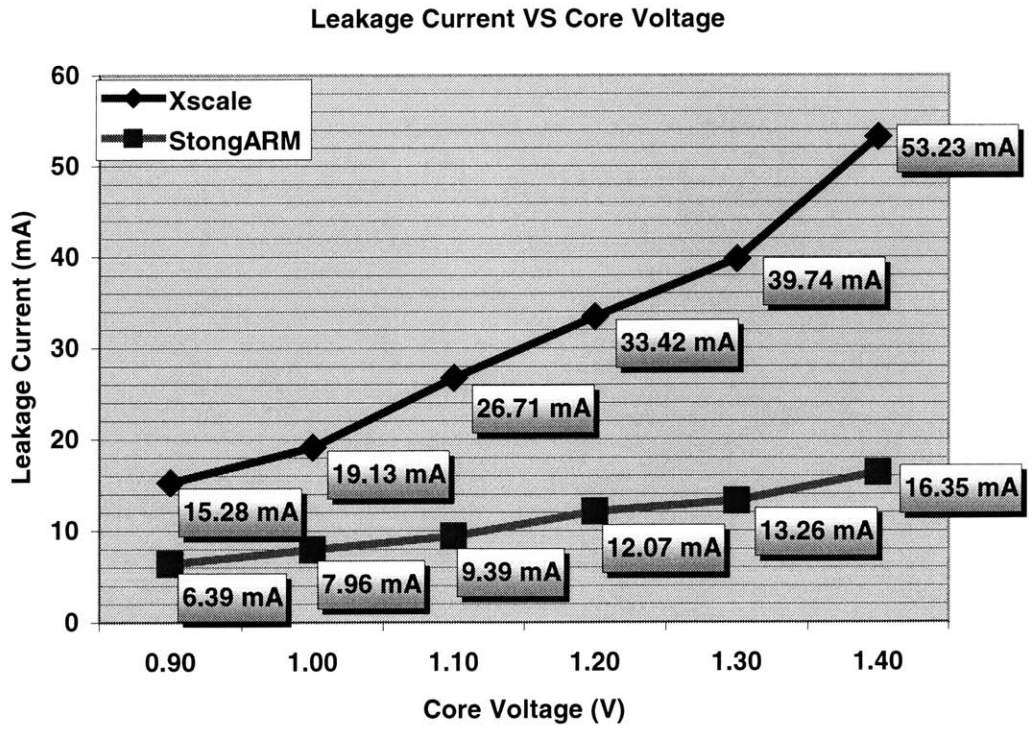

Figure 4.5: Leakage Current versus Core Voltage (Xscale and StrongARM)

Figure 4.5 above shows the leakage current plots versus core voltage for both Xscale and StrongARM. As far as static current consumption is concerned, Xscale consumes more than StrongARM as the plot illustrates. For these voltage values, 0.90 V, 1.0 V, 1.1 V, 1.2 V, 1.3 V, and 1.4 V Xscale consumes 2.39, 2.4, 2.85, 2.77, 3, and 3.36 times more leakage current than StrongARM respectively. The ratios should be in increasing order; however, the small drop at 2.77 is most likely due to experimental error. Note that the measurements for leakage current for StrongARM were done using a FFT benchmark program in [1]. However, it has been proven in earlier chapters that to a first order approximation the energy consumed per cycle and thus likewise the current drawn is independent on functionality of the instructions. Therefore, the values of the current (including leakage current) drawn for the benchmark used in [1] does indeed provide a reasonable comparison with the leakage current measurement results for Xscale using the FIR filter program.

Since the operating frequency on the two processors is not the same, it is important to look at the leakage energy consumption. The same FIR filter program used for Xscale was run on StrongARM to measure the cycles taken by StrongARM to run the same program as Xscale. The results of the energy measurements are shown in Table 4.8 below.

For the same core voltage is was shown above that Xscale consumes greater leakage current, where the first column in the comparison category illustrates this. However, it is important to note that Xscale operates much faster than Xscale, so even though it may be leaking more than StrongARM, it is leaking for a shorter time, and thus the leakage energy is less than StrongARM. The comparison column in Table 4.8 illustrates this point effectively, the three columns in this category is showing the factor by which Xscale is consuming greater leakage current, lesser leakage energy, and operating faster with respect to the same parameters of StrongARM. Therefore it has been proven that Xscale is energy efficient in both dynamic and static energy consumption. Lastly Figure 4.6 shows the total, dynamic and static current consumption for Xscale for the FIR filter program.

Table 4.8: Static Energy Consumption Statistics for Xscale and StrongARM

	StrongARM Cycles: 3525			Xscale Cycles: 3206			Comparison		
V_{core} (V)	I_{leak} (mA)	E_{leak} (μJ)	Frequency (MHz)	I_{leak} (mA)	E_{leak} (μJ)	Frequency (MHz)	I_{leak} (factor)	E_{leak} (factor)	Frequency (factor)
0.90	6.39	.275	73.7	15.28	.107	400	2.39	.39	5.43
1.0	7.96	.272	103.2	19.13	.125	466	2.40	.46	4.52
1.1	9.39	.309	118	26.71	.177	533	2.85	.57	4.52
1.2	12.07	.385	132.7	33.42	.214	600	2.77	.56	4.52
1.3	13.26	.375	162.2	39.74	.249	666	3.00	.66	4.11
1.4	16.35	.456	176.9	53.23	.326	733	3.26	.72	4.14

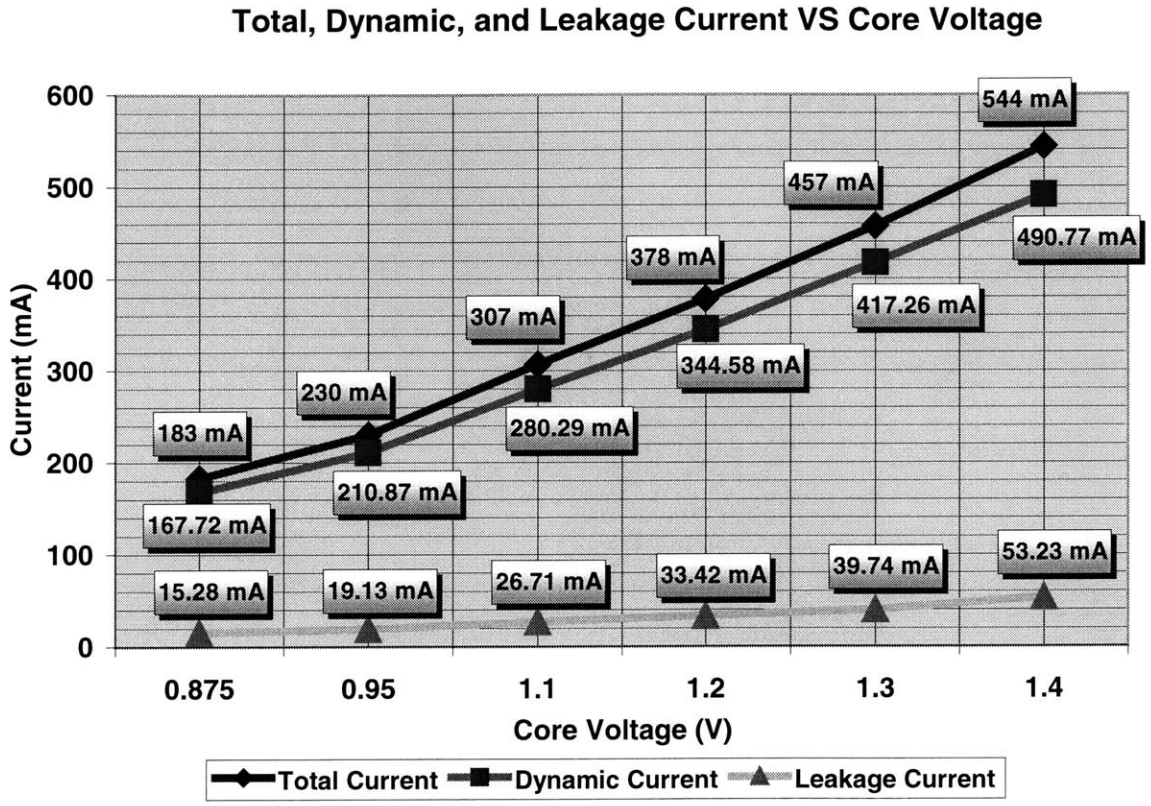

Figure 4.6: Total, dynamic, and static current vs. core voltage for Xscale

4.4 Low Power Modes

This section explores the 3 idling modes available on the Xscale processor, "Idle," "Drowsy," and "Sleep" [17]. Even though in the "Idle" mode the PLL is on, the result of the current drawn during this mode is independent on the value the clock frequency is set to (within a 6% worst-case variation), but clearly depends on the value of the core

voltage. For both the "Drowsy" and "Sleep" modes the PLL is turned off. The results of these measurements are shown in Table 4.9 below, where the modes are listed in increasing power saving order.

Table 4.9: Low Power Mode Statistics

	V = 1.5 V		V = 1.35 V		V = 1.25 V		V = 1.18 V	
Low Power Mode	P (mW)	I (mA)	P (mW)	I (mA)	P (mW)	I (mA)	P (mW)	I (mA)
Idle	38.3	25.5	25.2	18.7	20.5	16.4	16.6	14.16
Drowsy	16.3	10.8	10	7.37	9.33	7.46	9.13	7.76
Sleep	15.2	10.1	10	7.37	7.34	5.86	5.77	4.89

4.5 Concluding Remarks

This chapter concludes the measurements for this thesis and concludes the evaluation of software energy on the two chosen microprocessors. From the results of this chapter it can be concluded that, Xscale is a high performance, energy efficient processor in both static and dynamic energy consumption. The low power modes of Xscale have also been explored as well.

Chapter 5

Conclusion

Energy efficiency of systems are increasingly becoming an important issue. As the demand for speed and performance grow, so does the number of transistors used in microprocessors. Thus directly leading to a dramatic increase in energy consumption. For instance maximum processor energy consumption has tended to increase by approximately two fold every four years [2]. The issue of energy efficiency has been thoroughly explored in hardware design. However, it is also important to minimize software energy consumption. Since the energy consumed by the hardware components are in part due to the software being run on the hardware. As a result the goal of this thesis was to evaluate the energy consumed by software and present a possible technique for optimal software design.

The software energy consumption of two Intel microprocessors, the Xscale and the StrongARM, has been explored. Initially, the software energy consumption measurements were done at the individual instruction level taking into account all the addressing modes, and then later expanded to a complete program. The results of these extensive experiments led to the conclusion that the energy consumed per instruction is to first order part independent on the functionality of the particular instruction. Rather the energy per instruction depends on the cycles taken by the instruction to execute, since it has been shown that the energy per cycle is approximately constant for most instructions. However, some extreme cases were also determined, where the energy per cycle was higher than most other instructions. For instance the load and store instructions on the Xscale processor consume roughly 20% and 30% more energy per cycle than the arithmetic instructions.

These results can be explained by understanding how the processor consumes energy during the execution of an instruction. General-purpose processors are designed for high performance and versatility. This is achieved at the cost of continuous operation of many processor components such as caches, buffers, and control sections. This presents an overhead that can't be avoided. As a result almost all instructions regardless of functionality consume the same amount of energy. Most of the energy is spent powering the other components of the processor; however, there is some level of variability in the degree to which component is consuming energy, which leads to the exceptions pointed out earlier.

The fact that the energy consumed per cycle is roughly constant directly implies that optimizing for performance (minimal cycles) can optimize a particular program for energy. This optimization strategy is shown for an 11-tap FIR filter for both processors and some interesting results were obtained. For instance on average the StrongARM consumed 1.71 times more energy and it operates 4.89 times slower than Xscale.

The next step was to evaluate the leakage current and static energy consumption. Results showed that, Xscale consumes 2.39 – 3.26 times more leakage current than StrongARM for a voltage range of 0.9 – 1.4 V. This is mainly due to the drop in the threshold voltage of the transistors in Xscale, which is a result of the technology scaling implemented in Xscale. On average the leakage current for Xscale is 8.8 % of the total current. However it is important to note that, even though the leakage current is higher in Xscale, it also operates on average 4.54 times faster, thus it is leaking for a shorter period of time. As a result the leakage energy consumption for Xscale is .39 - .72 times less than StrongARM, for a voltage range of 0.9 – 1.4 V.

In conclusion, the results of all the extensive instruction level energy profiling and benchmark program testing showed that the Xscale processor consumes significantly less energy than StrongARM in both dynamic and static energy consumption.

References

[1] Amit Sinha, "Energy Aware Software," Master of Science in Electrical Engineering and Computer Science Thesis, Massachusetts Institute of Technology, December 1999

[2] Stephen H. Gunther, Frank Binns, Douglas M. Carmean, and Jonathan C. Hall, "Intel Technology Journal, 1st Quarter: Managing the Impact of Increasing Microprocessor Power Consumption," With Specific reference to the section on "Processor Power Trends." Entire reference found at: http://developer.intel.com/technology/itj/q12001/articles/art_4.htm

[3] Tajana Simunic, Luca Benini, and Giovanni De Micheli, "Energy-Efficient Design of Battery-Powered Embedded Systems" International Symposium on Low Power Electronics and Design, 1999 Proceedings, pp 212-217.

[4] Application Note 34: Writing Efficient C for ARM, January 1998. This can be found on the following web site: http://www.arm.com/armwww.ns4/html/Application_Notes?OpenDocument

[5] Chingren Lee, Jenq Kuen Lee, and TingTing Hwang, "Compiler Optimization on Instruction Scheduling for Low Power," The 13th International Symposium on System Synthesis, 2000 Proceedings, pp 55 –60.

[6] Huzefa Mehta, Robert Michael Owens, Mary Jane Irwin, Rita Chen, and Debashree Ghosh, "Techniques for Low Energy Software," International Symposium on Low Power Electronics and Design, 1997 Proceedings, pp 72 –75.

[7] Ching-Long Su, Chi-Ying Tsui, and Alvin M. Despain, "Low Power Architecture Design and Compilation Techniques for High-Performance Processors," Compcon Spring '94, Digest of Papers, pp 489 – 498.

[8] N. Vijaykrishnan, M. Kandemir, M. J. Irwin, H. S. Kim, and W. Ye, "Energy-Driven Integrated Hardware-Software Optimizations Using SimplePower," Proceedings of the 27th International Symposium on Computer Architecture, 2000, pp 95 –106.

[9] G. Esakkimuthu, N. Vijaykrishnan, M. Kandemir, and M. J. Irwin, "Memory System Energy: Influence of Hardware-Software Optimizations" Proceedings of the 2000 International Symposium on Low Power Electronics and Design, ISLPED '00, pp 244 –246.

[10] http://developer.intel.com

[11] http://developer.intel.com/design/intelxscale/

[12] http://developer.intel.com/design/iio/docs/iop310.htm

[13] Intel 80200 Processor based on Intel Xscale Microarchitecture Datasheet, October 2000, Reference Number 273414-002

[14] Intel 80200 Processor based on Intel Xscale Microarchitecture – Delivers Core Performance Breakthrough, Product Brief, Order Number 273427-001

[15] Intel Xscale Microarchitecture: Serves Up Breakthrough I/O, Order Number 273434-002

[16] Intel Xscale Microarchitecture: Technical Summary, found at the following web site:
http://developer.intel.com/design/intelxscale/XScaleDatasheet4.htm?iid=xscale+leftnav&

[17] Intel 80200 Processor based on Intel Microarchitecture Developer's Manual, November 2000, Order Number 273411-002,

[18] http://developer.intel.com/design/strong/

[19] http://developer.intel.com/design/strong/sa1100.htm?iid=strongarm+leftnav&

[20] http://developer.intel.com/design/strong/collateral.htm?iid=strongarm+leftnav&

[21] Intel StrongARM SA-1100 Microprocessor for Embedded Applications: Brief Datasheet, June 1999, Order Number: 278092-005.

[22] Intel StrongARM SA-1100 Microprocessor: Specification Update, February 2000, Order Number: 278105-025

[23] Intel StrongARM SA-1100 Microprocessor Developer's Manual, August 1999, Order Number 278088-004

[24] A. Sinha and A. Chandrakasan, "JouleTrack-A Web Based Tool for Software Energy Profiling," Proceedings of the 38th Conference on Design Automation Conference, 2001, pp 220 – 225.

[25] 6.374 Analysis and Design of Digital Integrated Circuits, Class Notes by Professor Dr. Anantha Chandrakasan, Massachusetts Institute of Technology, Fall 1999.

[26] James Kao, Anantha Chandrakasan, and Dimitri Antoniadis, "Transistor Sizing Issues and Tool For Multi-Threshold CMOS Technology," Proceedings of the 34th Design Automation Conference, 1997, pp 409-414.

[27] Vivek Tiwari, "Logic and System Design for low Power Consumption," Ph.D. Thesis Dissertation, Princeton University, Chapters 5 & 6, November 1996.

[28] Vivek Tiwari, Sharad Malik, and Andrew Wolfe, "Power Analysis of Embedded Software: A First Step Towards Software Power Minimization," IEEE Transactions on Very Large Scale Integration (VLSI) Systems, Volume: 2, Issue: 4, Dec. 1994, pp 437 – 445.

[29] Vivek Tiwari, Sharad Malik, and Andrew Wolfe, "Compilation Techniques for Low Energy: An Overview," IEEE Symposium on Low Power Electronics, Digest of Technical Papers, 1994, pp 38 –39.

[30] Mike Tien-Chien Lee, Vivek Tiwari, Sharad Malik, and Masahiro Fujita, "Power Analysis and Low-Power Scheduling Techniques for Embedded DSP Software," Proceedings of the Eighth International Symposium on System Synthesis, 1995, pp 110 –115.

[31] Vivek Tiwari, Sharad Malik, Andrew Wolfe, and Mike Tien-Chien Lee, "Instruction Level Power Analysis and Optimization of Software," Ninth International Conference on VLSI Design, 1996 Proceedings, pp 326 –328.

[32] S. Wiratunga, C. Gebotys, "Methodology for Minimizing Power with DSP Code," 2000 Canadian Conference on Electrical and Computer Engineering, Volume: 1, pp 293 –296.

[33] http://cygwin.com/

[34] GNU Main Page: http://www.fsf.org or http://www.gnu.org

[35] GNU Documentation: http://www.fsf.org/doc/doc.html

[36] GNU Online Manual: http://www.gnu.org/manual/manual.html

[37] GNU Assembler Documentation: http://www.fsf.org/manual/gas-2.9.1/html_chapter/as_toc.html

[38] http://www.arm.com

[39] ARM Software Development Toolkit Version 2.11 Reference Guide, Advanced RISC Machines Ltd (ARM) June 1997 , ARM DUI 0041B

[40] ARM Software Development Toolkit Version 2.11 User's Guide, Advanced RISC Machines Ltd (ARM) May 1997 , ARM DUI 0040C

[41] Amit Sinha and Anantha Chandrakasan, "Energy Aware Software," Thirteenth International Conference on VLSI Design, 2000, pp 50 –55.

[42] "Data Sheet: ADI/80200EVB Evaluation Board Documentation," footnote 3, page 12, March 2001: http://www.adiengineering.com/products.html

[43] Keithley Model 2400 SourceMeter User's Manual, Keithley Instruments Inc., 1996

[44] D. Seal ed., "ARM Architecture Reference Manual 2nd ed.,"Addison-Wesley, 2000.

[45] D. Jaggar ed., "Advanced RISC Machines Architectural Reference Manual: ARM Architectural Reference Manual," Prentice Hall, 1996.

[46] http://developer.intel.com/design/intelxscale/benchmarks.htm

[47] "Intel StrongARM SA-1110 Brief Data Sheet," Order Number: 278241-005, April 2000.

[48] Amit Sinha, "Energy Efficient Operating Systems and Software," Doctor of Philosophy in Electrical Engineering and Computer Science Thesis, Massachusetts Institute of Technology, August 2001

[49] Alan V. Oppenhiem, Ronald W. Schafer with John Buck, "Discrete-Time Signal Processing," Prentice Hall, New Jersey, 1999.

[50] Anantha P. Chandrakasan, Robert W. Brodersen, "Low Power Digital CMOS Design," Kluwer Academic Publishers, Boston, 2000.

[51] M. Johnson, D. Somasekhar, and K. Roy, "Models and Algorithms for Bounds on Leakage in CMOS Circuits," IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems, Vol. 18, pp714-725, June 1999.

[52] J. M. Rabaey, "Digital Integrated Circuits: A Design Prespective," Prentice Hall, New Jersey, 1996.

Appendix A

Supplementary Material for Xscale

Appendix A1: Preparation of Xscale for Applying Core Voltage Externally

Appendix A2: Commands and Batch Files Used for GNU Compiler and Debugger

Appendix A3: Main Program Excluding Profiling Segment

Appendix A1

Preparation of Xscale for Applying Core Voltage Externally

Top and bottom view pictures of the 80200 Evaluation Platform are shown below in Figures A.1 and A.2 respectively. The board operates with only a 12 V DC supply that goes to connector J9 on the board, the arrow in Figure A.1 points to the exact location. This voltage is then divided and voltage is supplied to the core internally, the connection trace (JP2) can be seen in Figure A.2, as pointed by another arrow.

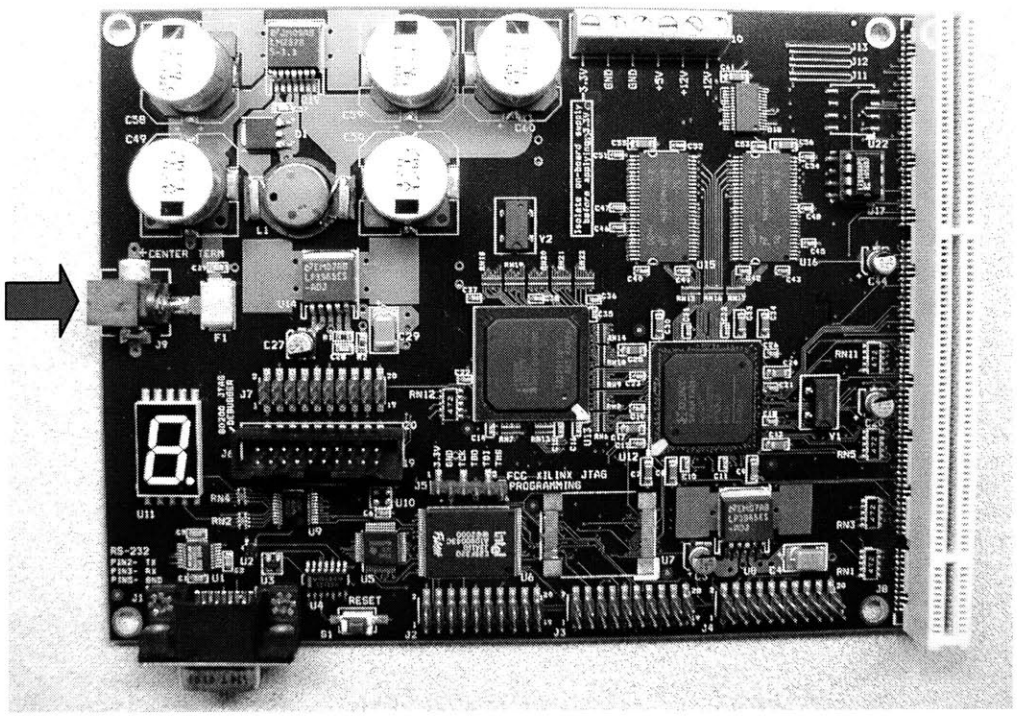

Figure A.1: Top view of the 80200 Evaluation Platform with Intel Xscale core processor

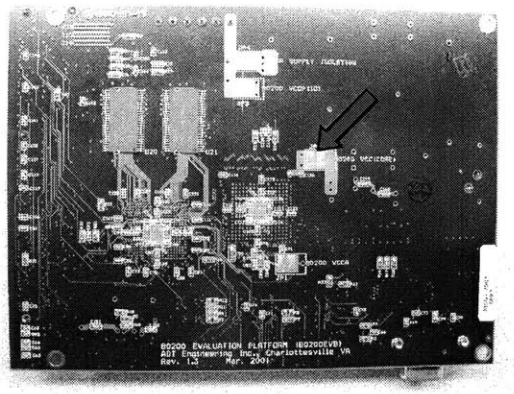

Figure A.2: Bottom view of the 80200 Evaluation Platform with Intel Xscale core Processor

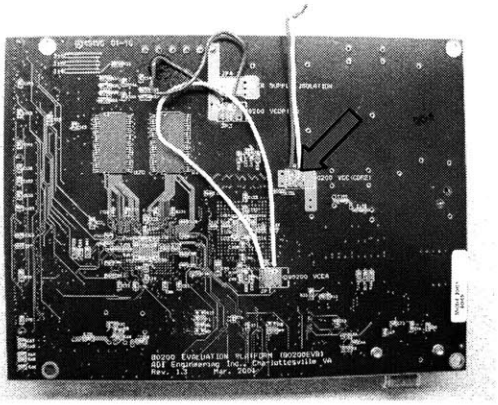

Figure A.3: Bottom view of the 80200 Evaluation Platform with Intel Xscale core processor with modifications to provide the core with external supply

Therefore in order to be able to make core current measurements and have the ability to manually change the voltage with corresponding frequency changes, this trace needs to be cut and the core will be run on external supply only. The 12 V DC supply will of course be supplied to provide power for the rest of the board. The result of doing this is shown in Figure A.3, again an arrow points to the exact location. The brown wire is used to provide voltage to the core. Note that the other wires that are there are not used for this experiment and so they are just reconnected to original configuration.

Appendix A2

Commands and Batch Files Used for the GNU Compiler and Debugger

The following command is required for compiling a program in C (it should of course be in one line):

/xscale-001130/H-i686-pc-cygwin/bin/xscale-elf-gcc -g -Llib -specs=lrh.specs -o <file_name>.elf <file_name>.c

This command creates an .elf file, which in turn is used by the debugger (GDB) to run on the board. The next step is to run the debugger, of course before entering this stage the board should be started up and the reset button pressed, as outlined in section 2.4.1 above. Also note that every time the debugger is exited the board must be reset. The following command is used to run the debugger:

/xscale-001130/H-i686-pc-cygwin/bin/xscale-elf-gdb –nw <file_name>.elf

After this command line, the GDB prompt is given, and then the following sequence of commands are needed to run the program on Xscale:

(gdb) set remotebaud 57600
(gdb) target remote COM1
(gdb) load
(gdb) continue

After the program is completed then "Ctrl-C" will quit and it is back to the (gdb) prompt, then entering "q" will quit the debugger (note no quotes are needed). This process could be repeated again for different programs or modifications can be done to the original file. Note that it is very important that the board be RESET after the debugger is exited. There is a simpler way of doing the above explained procedure, by using two separate batch files: Makefile and xscale.gdb, shown below. Therefore to compile fir.c, the commands are simply reduced to the following: **make fir.elf**. The command for the debugger is likewise reduced to: **make fir.gdb** which starts running on Xscale, this automatically starts up the debugger, it is not necessary to explicitly startup the debugger.

1. Makefile

```
XCC = /xscale-001130/H-i686-pc-cygwin/bin/xscale-elf-gcc
XLD = /xscale-001130/H-i686-pc-cygwin/bin/xscale-elf-ld
GDB = /xscale-001130/H-i686-pc-cygwin/bin/xscale-elf-gdb

PLL = 6

all:

%.elf: %.c
	$(XCC) -g -Llib -specs=lrh.specs -o $*.elf $*.c
%.o: %.s
	$(XCC) -g -c  -x assembler-with-cpp -o $*.o -DPLL=$(PLL) $*.S
%.o: %.S
	$(XCC) -g -c  -x assembler-with-cpp -o $*.o -DPLL=$(PLL) $*.S
%.elf: %.o
	$(XLD) -Ttext 0xC0008000 -o $*.elf $*.o
%.gdb: %.elf
	$(GDB) -nw -x xscale.gdb $<

clean:
	- rm *.o
	- rm *.elf
```

2. xscale.gdb

```
set remotebaud 57600
target remote COM1
load
continue
```

Appendix A3

Main Program Excluding Profiling Segment

```
//This program tests the entire Xscale instruction set, addressing Modes, and all possible
//formats for each instruction that is available or that can be run on the Intel Xscale
//processor.

//By Mitra M. Osqui for M.S.E.E. Degree
//June 28, 2001

#include <stdio.h>
#include <math.h>
#include <time.h>
#include <stdlib.h>

#define PLL_MULTIPLIER 6

#define LED ((volatile unsigned char *) 0x00500000)

char led_table[] = {0x7E, // 0
                    0x30, // 1
                    0x6F, // 2
                    0x79, // 3
                    0x33, // 4
                    0x5B, // 5
                    0x7D, // 6
                    0x07, // 7
                    0x7F, // 8
                    0x6F // 9
};

#define WRITE_LED(x) \
 do { \
   *(LED) = (0xFF ^ led_table[x]); \
 } while (0)

#define LED_OFF() \
 *(LED) = 0xFF
```

```
void
setupXSCALE(void)
{
        unsigned int pll = PLL_MULTIPLIER;
   asm volatile ("mcr p14,0, %0,c6,c0,0" : : "r"(pll) /*:*/);
   asm volatile ("mrc p14,0, %0,c6,c0,0" : "=r"(pll) : /*:*/);
        WRITE_LED(pll);
   printf("Now running at %.0fMHz:\n", (pll+2)*66.666666);
}

// Here the block name must be defined in order to make that block visible to the
//compiler, for instance if the instruction adc with the 1st type of addressing mode is
//chosen for energy profiling "block_name" must be replaced with adc1.

#define block_name

int main() {

  printf("Setting up Xscale ..... Please Wait\n");

  setupXSCALE();

  printf("Done :)\n");

/**************************************************************************
**************************************************************************
Here is where the profiling segment of the code is placed. There are a total of 347
separate experiment blocks that go in here. The whole program could not be included in
entirety since it was approximately 300 pages; however, a copy of it is located in my
MTL directory: MSThesis/Xscale/Instruction_Profiling/inst_e.c.
**************************************************************************
**************************************************************************/

  while(1);             //this endless loop is here since the core requires
                        // a never ending loop
  return (0);
}
```

Appendix B

Programs and Results for StrongARM

Appendix B1: setupARM.s file

Appendix B2: Main Program Excluding Profiling Segment

Appendix B3: Instruction Energy Profiling Result Details

Appendix B1

setupARM.s File

```
AREA Utility, CODE, READONLY

        EXPORT      setupARM

setupARM

        STMFD       SP!, {r0-r1}            ;
        MOV         r0, #0x17               ;
        SWI         0x123456                ;
        MRC         p15, 0, r1, c1, c0, 0   ;
        ORR         r1, r1, #0x1000         ; enables ICACHE
        ORR         r1, r1, #0x1            ; enables MMU
        ORR         r1, r1, #0x4            ; enables DCACHE
        ORR         r1, r1, #0x8            ; enables write buffer
        MCR         p15, 0, r1, c1, c0, 0;
        MOV         r0, #0x90000000         ;
        ADD         r0, r0, #0x20000        ;
        ADD         r0, r0, #0x14           ;
        LDR         r1, [r0]                ;
        BIC         r1, r1, #0xf            ;
        ORR         r1, r1, #0x09           ; Sets the frequency to 191.7 MHz
        STR         r1, [r0]                ;
        MRS         r0, CPSR                ; read the CPSR
        BIC         r0, r0, #0x1f           ;
        ORR         r0, r0, #0x10           ;
        MSR         CPSR, r0                ;
        LDMFD       SP!, {r0-r1}            ;
        MOV         PC, LR                  ;

        END
```

Appendix B2

Main Program Excluding Profiling Segment

```
//This program tests the entire StrongARM instruction set, Addressing Modes, and all
//possible formats for each instruction that is available or that can be run on the Brutus
//board. This is for the Intel StrongARM processor SA-1100 Brutus board.

//By Mitra M. Osqui for M.S.E.E. Degree
//Aug 2, 2000

#include <stdio.h>
#include <math.h>
#include <time.h>
#include <stdlib.h>

#define ITER 100000000

int main(int argc, char *argv)
{
 time_t start_time, end_time;
 double diff_time;

 void setupARM(void);

 printf("Setting up Strong ARM..... Please Wait\n");
 setupARM();
 printf("Done :)\n");

/*****************************************************************
*****************************************************************
Here is where the profiling segment of the code is placed. There are a total of 250
separate experiment blocks that go in here. The whole program could not be included in
entirety since it was approximately 650 pages; however, a copy of it is located in my
MTL directory: MSThesis/StrongARM/Instruction_Profiling/test_inst.c.
*****************************************************************
*****************************************************************/
```

```
 diff_time = difftime(end_time, start_time);
 printf("time of %d iterations = %f seconds\n", ITER, diff_time);

 return (0);
}
```

Appendix B3

Instruction Energy Profiling Result Details

This appendix provides some of the energy profiling details that were left out in the discussions of chapter 3. First the data-processing instructions and addressing modes are explored, then we move on to the other instructions. Figure B3-1 shows the energy results for the eleven addressing modes for the AND instruction. Again this graph is only for one instruction out of the 16 for this category, but it gives a good representation of the energy consumption values and patterns for the other 15 instructions as well. Since they consume to an acceptable tolerance the same amount of energy in each of the 11 addressing modes. The general pattern of energy consumption is very similar to that of Xscale, the shift-by-register addressing modes take 2-cycles to execute and thus consume roughly twice the energy. The pattern across addressing modes is also very similar to the Xscale, with the difference that the last addressing mode, rotate right with extend, is a 1-cycle instruction as opposed to a 2-cycle instruction for Xscale. Note that the CLZ instruction is unavailable to this board.

As noted in chapter 3 all standard ARM instructions are not available to this board, among them are the four multiply instructions SMLAL, SMULL, UMLAL, and UMULL. Thus from 6 instructions multiply instructions that exist in this category only 2 of them are available on Brutus, the MLA and MUL instructions. The energy consumed per instruction is the same for both instructions, and both instructions are 2-cycle instructions, thus we have: E_{inst} = 2.57 nJ and E_{cycle} = 1.28 nJ.

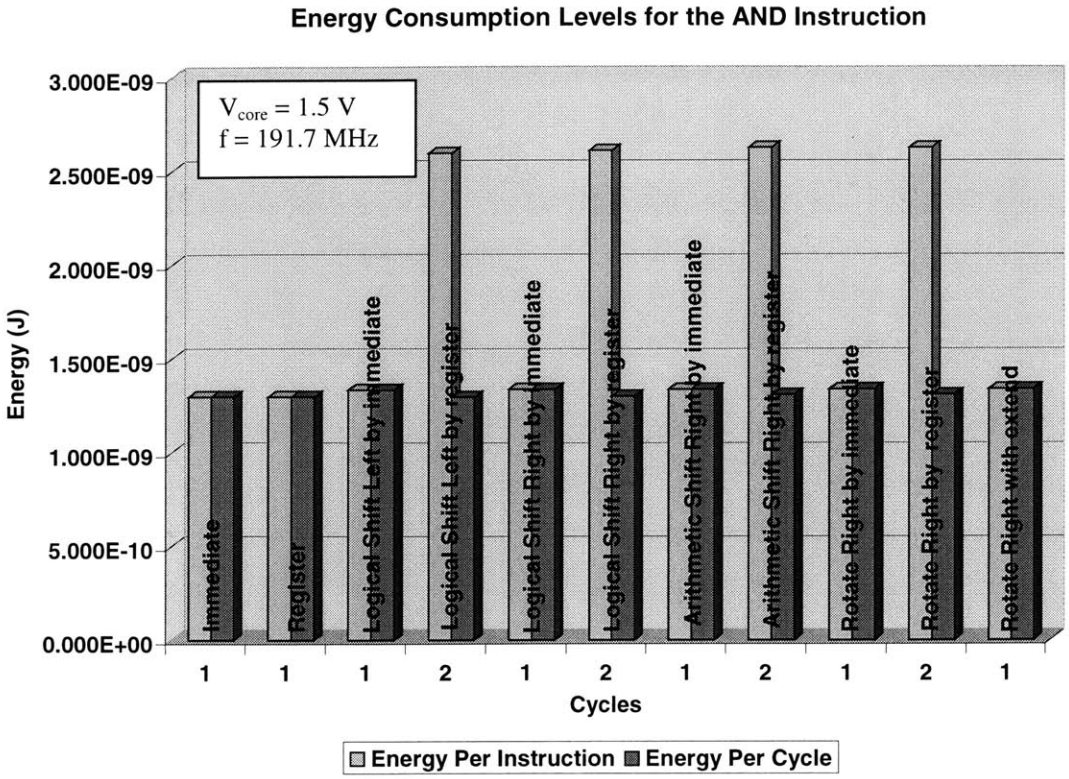

Figure B3-1: Energy Consumption for the Data-Processing Instructions

Now we move on to the load and store category of instructions, where again there are some instructions out of the standard instruction set that are not available, such as the following: LDRH, LDRSB, LDRSH, and STRH. Figure B3-2 shows the energy consumption results for the load and store instructions. Each bar in the figure represents the energy value averaged over the 9 possible addressing modes, this is done since the energy variation across the addressing modes is negligible. Both the load and store instructions take 1-cycle to execute for the StrongARM processor, while on the other hand the Xscale processor takes 3-cycles to do a load operation and depending on the addressing modes takes either 1 or 2 cycles to do a store operation. However, as shown in chapter 3 (Table 3.2), Xscale is still energy efficient.

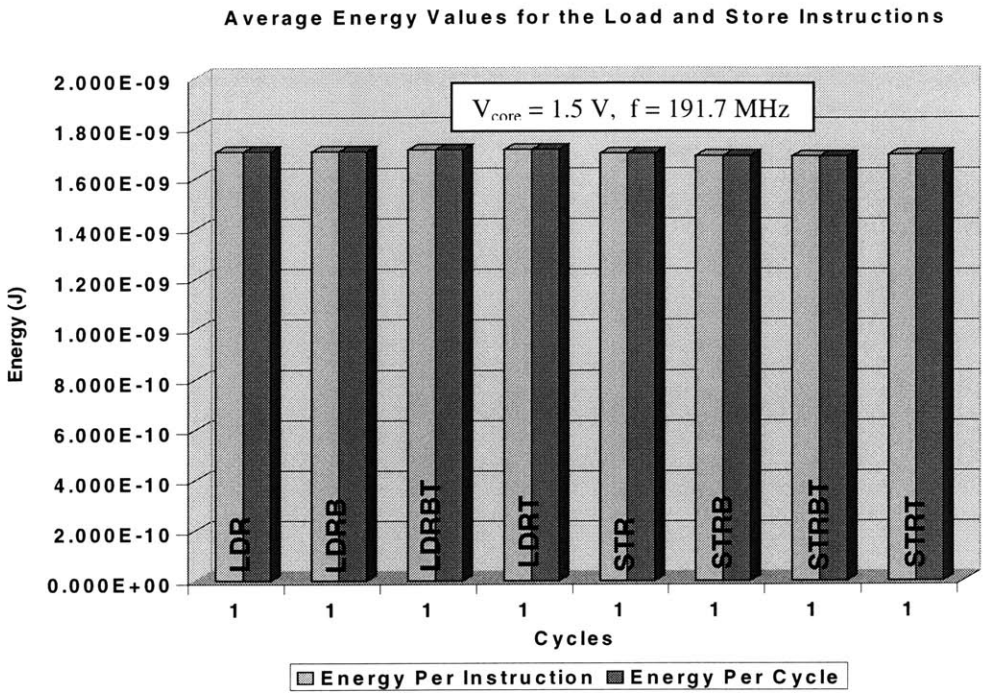

Figure B3-2: Energy Consumption for the Load and Store Instructions

The next instructions to explore is the load and store multiple instructions. As was the case for the Xscale processor the energy consumed by this class of instructions is dependent on the number of registers that is being loaded to or stored from. The base energy consumption for 1 register takes 2-cycles to complete. The same holds true if 2 registers are used, again it is a 2-cycle instruction. However, beyond two registers the processor takes one extra cycle to execute the instruction per added register. Thus if five registers are used, it takes 5-cycles to execute the instruction.

The following discussion of the energy consumption values for various addressing modes and instructions within this category are base energy consumption values, where only one register is used. The energy consumed per cycle for both load and store instructions are identical within a worst-case variation of 4.11%. Also addressing mode has no effect on the energy consumed by these instructions, where the average energy consumed is E_{inst} = 2.98 nJ and E_{cycle} = 1.49 nJ

An interesting observation can be made when the results of StrongARM are compared with that of Xscale. The base energy consumption (with only 1 register) for Xscale takes 3 cycles to complete either a load or store operation, also if 2 registers were to be used for Xscale it would require an extra cycle, which StrongARM does not require. Again even though these differences are pointed out, Xscale would be the processor of choice for energy efficiency and speed as was shown earlier in chapter 3 (Table 3.2).

Lastly the status register access, semaphore and branch instructions are explored. Figure B3-3 below shows the energy results for these instructions. From the results we see that 4 out of 6 status register access instructions are single cycle instructions with only two instructions 3-cycle instructions. The Xscale processor on the other hand has 5 out of 6 instructions that take 2-cycles to execute and only 1 instruction takes 1-cycle to execute. The semaphore instruction is also shown in the same figure and both instruction types consume identical amounts of energy and take 2 cycles to execute. On the other hand Xscale takes 5-cycles to execute the same semaphore instructions. Lastly the only available branch instruction is B, and this is a 2-cycle operation as opposed to Xscale, which executes the same instruction in a single cycle. As a final note all energy comparisons and analysis are once again referred to chapter 3 with specific reference to Table 3.2.

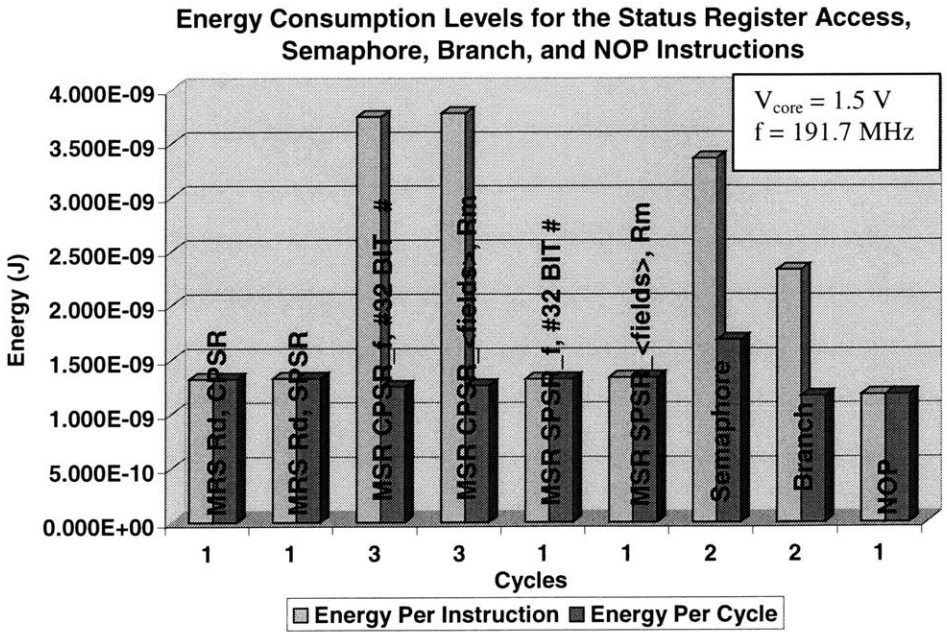

Figure B3-3: Energy Consumption for Miscellaneous Instructions

Appendix C

11-tap FIR Filter Program for Xscale

```
asm volatile
  ("

xdata:  .word 0
        .word 0
        .word 0
        .word 0
        .word 0
        .word 0
        .word 0
        .word 0
        .word 0
        .word 0
        .word 5
        .word 12
        .word 3
        .word 6
        word 8
        .word 25
        .word 49
        .word 11
        .word 2
        .word 30
        .word 23
        .word 18
        .word 15
        .word 3
        .word 1
        .word 24
        .word 9
        .word 35
        .word 21
        .word 7
        .word 0
```

```
        .word 0
        .word 0
        .word 0
        .word 0
        .word 0
        .word 0
        .word 0
        .word 0
        .word 0

coeff:  .word 5                 /* filter length: 11 */
        .word 4
        .word 10
        .word 12
        .word 7
        .word 31
        .word 24
        .word 1
        .word 6
        .word 19
        .word 20

yresult:.word 0

        ldr r0,=200000          /*the loop is executed 200,000 times*/
        mov r1, #00             /* loop counter */

        ldr r10,yresult         /*load starting address of result*/

outer_loop:

        add r1,r1,#01

.rept 30
        ldr r2,xdata            /* load starting address of xdata */
        ldr r3,coeff            /* load starting address of coeff*/
        add r3,r3,#44   /* point to the address of the last element in the array coeff*/

        mov r4,#0               /* initialize x counter */
        mov r5,#0               /* initialize h counter */
        mov r6,#0               /* initialize sum */

        ldr r7,[r2,#0]          /* load value in element 1st of array x*/
        ldr r8,[r3,#0]          /* load value in last element of array coeff*/

        mov r11,r2              /* make backup of r2*/
```

```
        mov r12,r3              /* make backup of r3*/

1:      add r4,r4,#1            /* loop by length of virtual x data */

2:      add r5,r5,#1            /* loop by length of coefficients of FIR filter*/

        mul r9,r7,r8            /* multiply-accumulate segment*/
        add r6,r6,r9

        ldr r7,[r11,#4]!        /* load next value*/
        ldr r8,[r12,#-4]!       /* load next value */

        cmp r5,#11              /*length of coefficients of FIR filter*/
        bne 2b

        str r6,[r10,#0]         /*  Rewrites back to same location in memory*/

        ldr r7,[r2,#4]!         /* point to next value */
        mov r11,r2              /* reset r11*/

        ldr r8,[r3,#0]          /* point to next value */
        mov r12,r3              /* reset r12*/
        mov r5,#0
        mov r6,#0               /* reset sum*/

        cmp r4,#40              /*length of virtual x data */
        bne 1b
.endr

        cmp r1,r0
        bne outer_loop

");
```