AN EVENT-LEVEL POWER MEASUREMENT AND ANALYSIS METHODOLOGY

BY

JEFFREY JAY NAMKUNG

B.S., University of Illinois at Urbana-Champaign, 1999

THESIS

Submitted in partial fulfillment of the requirements
for the degree of Master of Science in Electrical Engineering
in the Graduate College of the
University of Illinois at Urbana-Champaign, 2001

Urbana, Illinois
ABSTRACT

Researchers have been investigating low-power devices for many years. However, the recent surge in consumer demand for portable and low-power devices has increased the need for better techniques at lowering the power needed to run applications on these devices. Much of the existing research has focused on analyzing a processor’s power consumption behavior at a fixed level of abstraction, for example, transistor-level, gate-level, or instruction-level. Power estimation tools may rely either on a very detailed circuit-level description or on a very abstract process level. This thesis presents a new approach to power analysis by removing the abstraction levels and characterizing power consumption based on relevant factors that directly dictate how much power is consumed in an applications life span. The goal of this study is to produce as much groundwork as possible to create an efficient and accurate power estimation tool.
To my sister.
ACKNOWLEDGMENTS

This thesis would not have been able to be completed without the help of my adviser Sanjay Patel who supplied many insights and creative solutions to seemingly impossible problems, my supervisor Jagdish Patel for his advice on writing this thesis and his support on building the testbench used, Jet Propulsion Lab for their funding of this project, mom and dad for being my mom and dad, and the many other people who fed me constant encouragement, which I claim is the most important factor in the completion of this thesis.
# TABLE OF CONTENTS

<table>
<thead>
<tr>
<th>CHAPTER</th>
<th>PAGE</th>
</tr>
</thead>
<tbody>
<tr>
<td>1 INTRODUCTION</td>
<td>1</td>
</tr>
<tr>
<td>2 BACKGROUND</td>
<td>4</td>
</tr>
<tr>
<td>2.1 The Power-Energy Relationship</td>
<td>4</td>
</tr>
<tr>
<td>2.2 Energy in Circuits</td>
<td>5</td>
</tr>
<tr>
<td>2.2.1 Dynamic power dissipation</td>
<td>5</td>
</tr>
<tr>
<td>2.2.2 Dynamic logic</td>
<td>7</td>
</tr>
<tr>
<td>3 THE TEST BENCH</td>
<td>9</td>
</tr>
<tr>
<td>3.1 Basic Measurement Technique</td>
<td>9</td>
</tr>
<tr>
<td>3.2 Harware Hurdles</td>
<td>9</td>
</tr>
<tr>
<td>3.3 Measuring Power</td>
<td>10</td>
</tr>
<tr>
<td>3.4 The Computer</td>
<td>11</td>
</tr>
<tr>
<td>3.5 The Environment</td>
<td>11</td>
</tr>
<tr>
<td>3.6 Revising the Test Bench</td>
<td>12</td>
</tr>
<tr>
<td>3.7 Test Development</td>
<td>13</td>
</tr>
<tr>
<td>4 ENERGY CONSUMPTION MEASUREMENT FRAMEWORK</td>
<td>14</td>
</tr>
<tr>
<td>4.1 The Goal</td>
<td>14</td>
</tr>
<tr>
<td>4.2 Dissecting the PowerPC 750</td>
<td>15</td>
</tr>
<tr>
<td>4.3 Test Breakdown</td>
<td>16</td>
</tr>
<tr>
<td>4.3.1 System register unit</td>
<td>17</td>
</tr>
<tr>
<td>4.3.2 Integer instruction testing</td>
<td>17</td>
</tr>
<tr>
<td>4.3.3 Memory access tests</td>
<td>18</td>
</tr>
<tr>
<td>4.3.4 Floating-point instruction testing</td>
<td>18</td>
</tr>
<tr>
<td>4.3.5 Advanced tests</td>
<td>18</td>
</tr>
<tr>
<td>5 ANALYSIS</td>
<td>19</td>
</tr>
<tr>
<td>5.1 Testing the Register File</td>
<td>20</td>
</tr>
<tr>
<td>5.1.1 Results of register number effects</td>
<td>20</td>
</tr>
<tr>
<td>5.1.2 Operand data effects</td>
<td>24</td>
</tr>
<tr>
<td>5.2 Testing the Integer Units</td>
<td>29</td>
</tr>
<tr>
<td>5.2.1 Isolating IU1 and IU2</td>
<td>29</td>
</tr>
<tr>
<td>5.2.2 Execution of both IUs</td>
<td>30</td>
</tr>
<tr>
<td>5.2.3 Testing data forwarding</td>
<td>31</td>
</tr>
<tr>
<td>5.3 Testing the Cache</td>
<td>33</td>
</tr>
<tr>
<td>5.3.1 Memory address effects</td>
<td>33</td>
</tr>
<tr>
<td>5.3.2 Data value effects</td>
<td>34</td>
</tr>
</tbody>
</table>
5.3.3 Address translation ........................................... 34
5.3.4 Cache hits and misses ......................................... 37
5.4 Testing the Floating-Point Unit ................................. 39
  5.4.1 Floating-point registers ...................................... 41
  5.4.2 Floating-point loads and stores ............................ 41
  5.4.3 Fixed-point versus floating-point .......................... 46
5.5 Miscellaneous Tests ............................................. 49
  5.5.1 Stalls .......................................................... 49
  5.5.2 Multiple loads and stores .................................. 50

6 CONCLUSIONS .................................................. 53
  6.1 Summary ....................................................... 53
  6.2 Future Outlook ................................................ 54

REFERENCES .................................................. 55
## LIST OF ABBREVIATIONS

<table>
<thead>
<tr>
<th>Abbreviation</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>ALU</td>
<td>Arithmetic logic unit</td>
</tr>
<tr>
<td>A</td>
<td>Amperes</td>
</tr>
<tr>
<td>CMOS</td>
<td>Complementary metal-oxide-semiconductor</td>
</tr>
<tr>
<td>DARPA</td>
<td>Defence advanced research projects agency</td>
</tr>
<tr>
<td>EA</td>
<td>Effective address</td>
</tr>
<tr>
<td>FPU</td>
<td>Floating-point unit</td>
</tr>
<tr>
<td>ILP</td>
<td>Instruction level parallelism</td>
</tr>
<tr>
<td>IPC</td>
<td>Instructions per cycle</td>
</tr>
<tr>
<td>IU</td>
<td>Integer unit</td>
</tr>
<tr>
<td>JTAG</td>
<td>Joint test action group</td>
</tr>
<tr>
<td>LSU</td>
<td>Load-store unit</td>
</tr>
<tr>
<td>mJ</td>
<td>Millijoules</td>
</tr>
<tr>
<td>NVRAM</td>
<td>Nonvolatile random access memory</td>
</tr>
<tr>
<td>RTOS</td>
<td>Real-time operating system</td>
</tr>
<tr>
<td>SRU</td>
<td>System register unit</td>
</tr>
<tr>
<td>VLIW</td>
<td>Very long instruction word</td>
</tr>
</tbody>
</table>
CHAPTER 1

INTRODUCTION

The importance of low-power optimizations and techniques has risen with the recent surge in portable and battery-operated electronic devices. Laptops, cellular phones, MP3 players, and personal digital assistants (PDAs) are a few of the devices in the technology market where manufacturers are probing the low-power research area to produce longer lasting products.

Additionally, the need for low-power techniques does not stop at portable and battery-operated devices. With the advance of technology, processor manufacturers are able to squeeze more and more transistors onto a chip. This increase in chip density comes at a cost in the form of heat and power dissipation. The denser the chip becomes, the more power is dissipated and the more heat generated. Heat inhibits functionality of the transistors, which results in poorer performance. In addition to poorer performance, excessive heat in the transistors can cause long-term damage, which decreases the reliability of the chips.

Low-power research has brought about many different tools and techniques to allow devices to run much longer and still maintain a decent level of performance. Optimizations made in hardware or software require a detailed power consumption analysis of the processor. This thesis focuses on a new perspective for analyzing power consumption. An actual measurement setup derived from a previous work [1] is used as a test bench for this paper. The analysis methodology presented in this paper is a preliminary step to designing various power estimation tools.

Optimizing hardware or software or both to have lower power consumption requires a few preliminary steps. To test new optimization techniques, researchers cannot just go to the processor fabrication plant and manufacturer a couple of chips to test power optimizations. Additionally, if the optimization techniques are in software, then there is no hardware to be designed and fabricated,
For such software optimizations, simulation and estimation tools have been developed to test new low-power optimization techniques. Furthermore, estimation and simulation tools are based on models developed, which emulate, to some extent, the hardware or software being optimized.

The models themselves must correlate accurately with the actual hardware or software. How accurate depends on the level of abstraction at which the model analyzes the actual hardware. Developing power models at a certain level of abstraction will have advantages and disadvantages. A power model that has a high level of abstraction (i.e., instruction-level) will have less accuracy than will a power model with a low level of abstraction (i.e., transistor-level). In terms of efficiency, a transistor-level model will be less efficient than an instruction-level model since the number of transistors the model has to account for is orders of magnitude larger than the number of instructions an instruction-level model needs to account for. Efficiency translates to longer execution time; therefore, the execution time of a transistor-level power estimator will be much longer than the execution time of the instruction-level power estimator.

Some examples of power estimators that use the transistor-level model as their basis for estimations are PowerMill by Epic, LSimPower by Mentor Graphics, and Star-HSpice by Avant! There are many other transistor-level power estimation tools and are presented in [2]. Frenkil [2] also shows some other power estimators that run at a slightly higher level of abstraction. WattWatcher by Sente and DesignPower by Synopsis are a couple of tools that run at the gate-level abstraction level. Sente also developed another version of WattWatcher, which performs register transfer level analysis to estimate power consumption.

Instruction-level analysis was introduced in [1] and since then there have been many different analyses performed at this level. Sami et al. [3] developed an instruction-level power estimator for VLIW processors, Brandolese et al. [4] took the instruction-level model and presented a more detailed model that was based off of the actual functional complexity of the instructions. A similar work in [5] also developed a new approach to the instruction-level model by analyzing how the data in the processor’s internal units affect power consumption. Russell and Jacome [6] ported the measurement setup in [1] to an embedded system and modified the measurement setup by replacing an ammeter with an oscilloscope.

The analysis methodology presented in this thesis brings together the works in [1, 4, 5, 6]
and some new approaches. The level of abstraction is not set to the instruction-level at which these other works operate. Instead, this paper presents an abstraction-level absent approach to the power analysis of a 32-bit microprocessor. A PowerPC 750 processor was used for analysis since the research was done in correlation with a Defence Advanced Research Projects Agency (DARPA)-sponesened project at Jet Propulsion Laboratory. The removal of a fixed abstraction level, e.g., transister-leve, gate-level, or instruction-level, allows for more relevant correlations in modeling the power.

The rest of this paper is divided into five chapters. Chapter 2 provides some background information on the origin of power consumption in circuits. An understanding of where power consumption exists in the circuits is essential to analyzing the measurements made in this paper. Chapter 3 introduces the test bench and the measurement methodology used. Chapter 4 discusses how the tests were devised and what they are testing. Chapter 4 will also examine the functionality within the PowerPC 750 to help verify that the tests are coming up with relevant data. The amount of details discussed for the PowerPC 750 will concentrate only on those details necessary to understanding the tests and their results. Chapter 5, the largest portion of this thesis, will discuss the results and analysis. Chapter 6 wraps up this thesis with some concluding remarks and future outlooks.
CHAPTER 2

BACKGROUND

2.1 The Power-Energy Relationship

*Power* and *energy* are often used interchangeably. However, for this thesis, it is important to distinguish between the two. A measurement in terms of average power may not reflect much relevance, while an energy consumption measurement may. Similarly, a measurement in terms of average power may be sufficient enough for analysis.

The difference between power and energy is that power does not take time into account, whereas energy is a function of time. This can be seen by Equations (2.1) and (2.2):

\[ p = v \times i \]  
\[ e = p \times t \]

Average power is a function of the CPU's voltage and the average current being drawn. Energy is a function of average power and time. For the rest of this paper, results will be presented as either average power or consumed energy, depending on which unit is more meaningful for analysis. Furthermore, since power is a function of voltage and current and since for the 32-bit microprocessor under investigation, voltage is constant, only the current measurements will be presented as an indicator of power consumption.
2.2 Energy in Circuits

Understanding where and why energy is consumed within a processor is essential to the analysis performed in this paper. Power dissipation in CMOS circuits can be split into two categories: static and dynamic. Static power dissipation refers to the power dissipated in a circuit, which is independent of frequency. Dynamic power dissipation is the power dissipated in the circuit that is dependant on the frequency. Most of the power dissipated in circuits comes from dynamic power dissipation. Therefore discussion will be limited to only dynamic power dissipation.

2.2.1 Dynamic power dissipation

Dynamic behavior refers to the switching effects that occur within the circuits. Dynamic power dissipation can be further divided into two components: capacitive and short-circuit. To better understand why dynamic power dissipation exists, a CMOS inverter will be used as an example. Figure 2.1 shows a transistor-level diagram of a CMOS inverter.

![CMOS Inverter Diagram](image)

Figure 2.1: A CMOS Inverter

Capacitive loads

Capacitive power dissipation can be seen when the circuit input transitions from either low to high, or high to low. When \( C_L \) gets charged through the PMOS transistor, its voltage rises from 0 to \( V_{DD} \). The current flowing from \( V_{DD} \) to \( C_L \) flows through the PMOS transistor and energy is dissipated in the PMOS transistor. When \( C_L \) is discharged through the NMOS transistor, the
stored energy is dissipated in the NMOS transistor.

The exact measurement of capacitive power can be calculated. The derivations of this exact measurement have been left out for brevity. The equation for capacitive power dissipation can be seen in Equation (2.3), where \( f \) represents the number of times the gate switches per second:

\[
P_{\text{cap}} = C_L V_{DD}^2 f
\]  \hspace{1cm} (2.3)

**Short-circuit currents**

Short-circuits are the second component of dynamic power dissipation. In the previous discussion on capacitive loads, an assumption was made that when the circuit switches, the transistors exhibit zero rise and fall times. However, zero and rise fall times, though idealistic, are not realistic. Hence, during a transition, there is a momentary instant when both transistors are on and there exists a path from \( V_{DD} \) to \( GND \). Figure 2.2 shows the same transistor-level diagram of the inverter, but with the short circuit current \( I_{SC} \) labeled.

![Figure 2.2: A CMOS Inverter in Transition](image)

Momentary short-circuits occur when the inputs switch from 1 to 0 or vice versa. Figure 2.3 shows a graph of the short circuit current \( I_{SC} \) during transitions in the input \( V_{IN} \).

The value of this short-circuit current depends on how fast the transistors are able to switch. The faster the transistors switch, the less time the short-circuit current flows. The measured value of short-circuit currents can also be calculated, but will be left out for brevity as well. Equation (2.4)
displays the equation for power dissipated by short-circuit currents:

\[ P_{SC} = t_{SC}V_{DD}I_{peak} = C_{SC}V_{DD}^2f \]  \( (2.4) \)

Adding both components of dynamic power dissipation, the total power dissipated by dynamic behavior of the circuit is equal to the capacitive power plus the short circuit power. Equation (2.5) shows the equation for total dynamic power:

\[ P_{total} = (C_{SC} + C_L)V_{DD}^2f \]  \( (2.5) \)

2.2.2 Dynamic logic

Most processors rely on a technique called dynamic CMOS. Dynamic CMOS presents an additional factor to power dissipation in terms of a clock. A dynamic logic circuit is shown in Figure 2.4.

Since the clock is switching periodically, it can be seen that the load capacitance is either left charged or discharged on each clock cycle depending on the logic inbetween the two clocked transistors. However, the load capacitor always exhibits some charging when the clock is low. If the output of the logic is 0, then the precharge that the load capacitor developed during the low phase of the clock is discharged to ground when the clock goes high. On the other hand, if the output of the logic is 1, then the load capacitance stays charged. Thus, it can be seen that a 0 value exhibits more power dissipation than a 1 value in dynamic CMOS. This behavior will become more relevant in the Chapter 5.
The equations presented earlier are solely for the purpose of showing where power consumption exists in circuits. It is important to realize how all these equations relate to the tests developed. To generalize, power consumption in circuits is for the most part composed of dynamic power dissipation. Therefore, circuits that have a high switching frequency will exhibit more power consumption than less active circuits. Additionally, since processor technology relies on mostly dynamic CMOS logic, a 0 value on the output of some logic will consume more power than a 1 at the output.
CHAPTER 3

THE TEST BENCH

3.1 Basic Measurement Technique

The initial step to this research was to take Tiwari et al.’s [1] work and extend it into a more abstract power measurement tool. However, due to advances in technology, creating a test bench to measure power consumed by the processor under investigation proved to be quite challenging.

3.2 Harware Hurdles

Directly measuring energy consumed by a processor was deceivingly difficult. Past processors were connected to the motherboard a la PGA (pin grid array) connection, which would allow the processor to be pulled on and off the motherboard without any tools other than one’s hand and perhaps the help of a release lever. As technology advanced, so did the connections between processor and motherboard. Instead of a PGA connection, today’s processors utilize a BGA (ball grid array) connection, where the processor is soldered directly to the motherboard leaving a gap of one millimeter or less between the processor and motherboard. BGA connections are designed to be permanent, and existing techniques to remove BGA connections are far from reliable. Therefore, to probe the power pins of the processor, a special connector was necessary to obtain the measurements. A BGA-to-PGA connector was used to make the connections to the processor more accesible. However, additional complexity arose from the method of power distribution on the PowerPC 750,
To evenly distribute power, the processor requires 12 CPU core voltage pins, 20 I/O voltage pins, 12 L2 cache pins, and 2 more additional pins for the CPU and L2 cache clocks. Figure 3.1 highlights the pin mapping of these voltage pins.

![Figure 3.1: PowerPC 750 Pins](image)

As Figure 3.1 shows, the pins are distributed within the perimeter of the grid array, which proved to be difficult to probe. Thus, an intermediate board was implemented, which could be inserted between the processor and motherboard. This intermediate board basically serves as a “break-out” board, which passes all the pins through. To isolate the power supply, the pins that supply the power and ground connections were removed from the intermediate board, which connects to the motherboard. Four connections providing necessary voltage and ground to the processor are accessible on the board to allow a separate power supply to provide power to the processor.

### 3.3 Measuring Power

To measure the current drawn into the processor, an oscilloscope with a 50-MHz current probe was connected to the wires connecting the isolated power supply and the break-out board. Figure 3.2 shows the PowerPC 750 attached to the break-out board. Since the PowerPC 750 runs at an operating speed of 266 MHz, the oscilloscope needs to have a fairly high sampling rate. A Tektronix
TDS7000 series digital phosphor oscilloscope was used. The scope was able to capture up to 10 megasamples per second, which equates to a 1 GHz sampling frequency; plenty for this study.

![Image](image.png)

Figure 3.2: PowerPC 750 "break-out" Board

3.4 The Computer

The rest of the test setup was supplemented with a single board computer configuration developed by Embedded Support Tools under Wind River Inc. The board incorporates all the essential components to run a fully capable system. One megabyte of L2 cache was used in addition to 64 MB of main memory. NVRAM, system bus, serial and ethernet ports, and JTAG connections with software support for application compilation and download were available. The use of these added features served only to set up the tests before measurement. Thus, the use of these features did not inject their own power measurement characteristics into the results of the tests.

3.5 The Environment

VxWorks, a real-time operating system (RTOS) was used for the environment. However, since the tests were written as small loops of simple instructions, VxWorks did not inject any OS-specific instructions or tasks. Therefore, measurements were able to focus on the instructions pertaining to
the particular test. The RTOS was mainly used to run specific tests in correlation with a DARPA-sponsored project performed in parallel with this study at Jet Propulsion Laboratory. For future work, the RTOS environment could be used to analyze possible power optimization techniques at the application and operating system level.

3.6 Revising the Test Bench

After acquiring and assembling the test bench, it was soon realized that providing an isolated power source to the PowerPC via the breakout board was unnecessary. The design of the single board computer gave access to specific power lines to the processor. To simplify the test bench, the breakout board and additional power supply were removed, and several wires were soldered between the power module and the daughterboard housing the PowerPC. The probe was attached to the wires supplying the voltage to the processor. A picture of this connection can be seen in Figure 3.3. Doing so, also created a “cleaner” test setup that eliminated error injected from outside sources, i.e., the isolated power supply.

![Figure 3.3: Probing Core CPU Power](image)

The resulting test bench can be seen in Figure 3.4. The computer used to compile and download the tests (not shown) is a standard PC. The oscilloscope is shown with an actual measurement of a running test.
3.7 Test Development

The basic measurement method involved running a specific instruction or set of instructions repeatedly over a loop and taking a measurement of current over a specified time. The measurement was averaged and assigned to the instruction or event executing for their specific tests. Note that the tests varied in the number of processor cycles. Thus, each test was averaged over the number of times an event of instruction occurred, depending on whether the test was instruction-level or event-level. All of the measurements were in units of current. The voltage level of the PowerPC 750 was measured to be constant at 2.67 V and can be factored in later to calculate the average power. However, since current is the factor that varies from test to test, it is the amount of current being drawn that is of interest for analysis.
CHAPTER 4

ENERGY CONSUMPTION MEASUREMENT FRAMEWORK

This chapter will provide an overview of how the tests were formulated and what they were intended to discover. Energy consumption behavior is a combination of many complex units and circuits within the processor. Isolating a unit or event that occurred in the processor is a combined effort requiring many different tests. The framework for testing the processor will give insight into how the measurements were used to profile power.

4.1 The Goal

This thesis serves as a preliminary step for characterizing power consumption on a processor at different levels of abstraction. The benefits of bringing together different levels of abstraction to design a power-profiling tool could mean better accuracy and efficiency. Better accuracy means that the estimates produced by the power-profiling tool would closely resemble the actual power profile of an application. Efficiency refers to the amount of time the power-profiling tool requires to generate a profile of the application. Instead of differentiating between levels of abstraction, this characterizing methodology relates power measurements with certain events that occur within the processor. Event-level measurement does not sustain a particular level of abstraction. An event could be the execution of a certain instruction, which might utilize the register file, cache, and load/store unit, or the event might be more specific, e.g., when the integer units perform data forwarding. Thus, by taking measurements with some freedom at the abstraction level, this characterization methodology can associate measurements more appropriately. Certain characteristics of a processor may not
need as much detail, when associating power measurements, while other characteristics are very specific and need to have much more detailed measurements.

To accomplish this goal, tests were developed, which exercised the processor appropriately to exhibit certain behaviors desired for measurement. In most cases, a single test was not sufficient to isolate a certain behavior. Thus, extensive testing required an in-depth understanding of the logic and functionality of the processor.

4.2 Dissecting the PowerPC 750

For this thesis a PowerPC 750 processor was used. A layout of the chip can be seen in Figure 4.1.

![Figure 4.1: The Layout of the PowerPC 750](image)

The PowerPC 750 has separate level 1 caches for instructions and data. There are also two integer units, a floating-point unit, and a load-store unit. What Figure 4.1 does not show is the register files and the logic flow. Figure 4.2 shows the internal datapath flow.
Figure 4.2: General Datapath of the PowerPC 750

In addition to knowing what functional blocks compose the processor, knowing how these blocks interact is just as important. Though Figure 4.2 shows the basic flow of execution, it does not give insight into how instructions utilize these units. Logic structures such as reservation stations and data forwarding are not shown. However, the existence of such mechanisms is vital to the performance of the processor and contribute to the total energy consumed. The Motorola specification \cite{7} provided much of the details needed to characterize events in the processor.

4.3 Test Breakdown

The units tested were the two integer units, the floating-point unit, the system register unit, the load/store unit, and the level 1 cache. Many tests overlapped more than one of these units; thus, the tests were categorized into five categories: register file unit, integer units, level 1 cache, floating-point unit, and some more advanced tests of interest. Before discussing the tests created to analyze each of these units, it is important to understand the focus of the analysis performed in the
next chapter, Conceptually, it can be seen that a specific unit in the microprocessor contains many logical components that all contribute to the unit’s energy consumption characteristics. However, this study is aimed at finding the energy consumption behavior of these units with respect to how they are being used. Thus, factors that remain constant regardless of the situation in which the unit is being used can be disregarded since they scale the energy consumption characteristics by a constant value. The factors that are variable and result in variable energy consumption values are the main concerns in characterizing the behavior of the unit under test. Therefore, tests were devised to exercise the factors that varied when a unit is being used.

4.3.1 System register unit

The first batch of tests developed exercised use of the system register unit, Characterizing the register file was the first logical step to creating a model of the microprocessor since the register file is accessed by all instructions used for the rest of tests in this study.

To characterize the register file, it was important to understand what factors are variable when the register file is accessed. The variable factors fall into two categories: the registers being accessed and the data transferred from the registers. Thus, two types of tests were developed to exercise either the registers used or the data transferred from the registers.

4.3.2 Integer instruction testing

Integer instructions include those instructions that operate on 32-bit integers and use the general purpose registers. These tests were split into four groups. The first group was a comprehensive test of many basic integer instructions, which include arithmetic, logic, and shift operations. These tests were used to get the general behavior of the integer units. The second group tested the effect of using differing registers as operands. These tests gathered information about the system register unit. Additionally, tests were run to observe the dependency of energy consumption on data that was held in the registers. The last group tested the interinstruction effects that occur when there is a data dependency between instructions.
4.3.3 Memory access tests

Testing memory accesses were divided into several different tests. First, the cache was tested to see the difference in energy consumption between memory accesses that hit or missed in the cache. The second batch of tests were integer loads and stores, which observed the energy consumption behavior of different addressing modes and the number of bytes transferred for the operation. The third batch of tests performed floating-point loads and stores. The last test performed the more advanced load and store instructions, which worked with a variable number of words in memory.

4.3.4 Floating-point instruction testing

Floating-point instructions used tests similar to the integer instructions category. First, a comprehensive test was performed to help characterize general behavior of the floating-point unit. Second, the floating-point registers were tested by varying which registers were used as operands. Tests were also performed to compare the energy consumption of the floating-point unit and the integer-units.

4.3.5 Advanced tests

Two more sets of tests were performed. The first set tested cases in which a stall occurs and consumes more than one processor cycle to execute the instruction that caused the stall. Stalls can occur whenever there is a resource conflict or when the amount of functionality required by an instruction requires extra processor cycles. During a stall, subsequent instructions are paused until the current instruction moves forward in the datapath. A few simple tests were run to cause a certain number of stalls to occur.

The second set of tests observed the advanced memory operation instructions: load multiple word \((lmw)\) and store multiple word \((stmw)\). These tests exhibited differences in their execution time, which depended on the number of words being transferred. Intuitively, the more words transferred, the more processor cycles required to execute the transfer.
CHAPTER 5

ANALYSIS

The results shown in the following sections are either displayed as current drawn or energy consumed. The relationship between current draw and energy consumed can be seen in Equation (5.1):

\[ E = I \times V \times t \times C \]  

(5.1)

where \( E \) represents the energy consumed; \( I \) and \( V \) represent the current draw and voltage being applied to the processor, respectively; \( t \) is the processor’s cycle time and; \( C \) is the number of processor cycles an instruction takes. Therefore, \( t \times C \) is the time an instruction takes to complete.

With this in mind, note that current draw will be shown when the number of cycles an instruction takes is not important. Also remember that power is directly proportional to current draw. Therefore, when a 20% difference in current is shown in comparing two instructions, the power will also have a difference of 20%. When the results of a test have variance in the number of processor cycles consumed for completion, energy consumed will be used as the unit for comparison.

This analysis section is divided into several discussions focusing on the different portions of the processor that were examined. Section 5.1 presents results of testing the register file. Section 5.2 goes into the integer unit test results, which have many correlations with the register file results. Section 5.3 discusses cache power consumption behavior. Section 5.4 analyzes the results obtained from the floating-point unit tests and also draws some comparisons between the integer units and the floating-point unit. Section 5.5 discusses stalls and more advanced load and store tests.
5.1 Testing the Register File

It is nearly impossible to test the energy consumed by the register file directly. Therefore, indirect tests were performed with as many outside factors as possible held at some constant value. Holding factors not affecting the register file constant, helped determine energy consumption behavior of the register file by itself. The register file is accessed with most integer instructions. Floating-point instructions access registers specifically for floating-point operations and are examined later in this thesis.

5.1.1 Results of register number effects

The register file contains 32 general purpose registers, which contain data that an instruction requires for execution. An instruction chooses which registers it needs via 5 bit lines. Thus, energy consumption behavior is dependent on which registers are accessed. To analyze the dependency of energy consumption on which registers are being accessed, tests were devised to exercise the use of different registers based on the bit line settings used to select these registers.

The “add” instruction was used since it is a basic instruction and appears more often in programs than other instructions. It does not require the use of the caches, and the base cost of the instruction was close to the energy consumption cost for most instructions. The only concern came from the fact that both integer units can execute the “add” instruction simultaneously. To keep things consistent, test cases were written carefully to model behavior that produces better instruction level parallelism (ILP), i.e., execution of both integer units.

To keep as much of the running environment consistent as possible, all the registers in the register file were initialized to hold a 0 value. Registers not used during execution held this value for the length of the measurement. Since the decimal value to select the registers does not hold much relevance in the digital arena, tests were designed to exercise the 5 bit lines that select the registers. Therefore registers 0, 1, 3, 7, 15, and 31 were used to show the effect of having 5, 4, 3, 2, and 1 number of register select lines held at 0, respectively. The “add” instruction utilizes a three register format, with two source registers and one destination register; thus, a total of 15 bit lines are used for selection. Tests were performed comprehensively holding either one destination and one source register constant or both destination registers constant while varying the register number
of the third register. Results of executing the add instruction while using R0 as the destination register and varying the source registers are shown in Figure 5.1.

Figure 5.1 shows that there is a linear behavior in the current drawn as the number of 0 bits increases in the second source register. The deviation in behavior shown in the bottom line and at the end of the other lines where the second source register is R0 represents the effect of having the same register used in both destination and source registers. Utilizing the same register as a destination and a source register suggests the effect of register dependency. Since the measurement is made over a loop of 100 iterations of the same instruction, each sequential instruction has a dependency on the instruction previously executed. This dependency causes the number of instructions completed on each processor cycle to decrease. Though, there are two integer units, each instruction must wait for the previous instruction to complete and generate a result. Therefore, simultaneous execution of two integer instructions becomes sequentialized due to the dependencies. Instead of completing two instructions per cycle, the dependencies restrict throughput to one instruction per cycle. The result of register dependency in the execution of instructions can be seen in Figures 5.2 - 5.6. Each “dip” in the graph represents a data dependency situation, as does the lowest line on each graph.
Figure 5.2: Current Draw When Destination Register is R1

Figure 5.3: Current Draw When Destination Register is R3
Figure 5.4: Current Draw When Destination Register is R7

Figure 5.5: Current Draw When Destination Register is R15
Figures 5.2 - 5.6 show the effects of varying the two destination registers while keeping the source register constant. Figure 5.7 shows averages of all the graphs to represent the effect of varying all the registers. It can be seen that the general behavior is that energy consumed increases as the number of 0 bits increases in an operand specifier. Register 0, as a result, requires more energy to access.

5.1.2 Operand data effects

The second factor in which energy consumed in the register file varies is the data values being read out of the registers. Each register can hold a 32-bit value, which is significantly more than the number of bits to address the registers. Therefore, one can assume that the register file’s energy consumption behavior is largely dependent on the data being used. The number of registers required by an instruction is also variable. For these tests, the results are split into three categories representing three operand formats available for basic integer instruction testing. The first format takes two source registers and puts the result into one destination register. The second format takes one source register and an immediate value and places the result in one destination register. The last format has one source register and one destination register. Other operand formats are beyond the scope of this thesis.
Figure 5.7: Average Current Draw for General Purpose Registers

Two register operand only

Tests with two operands only are presented first since they have the least amount of variable factors needed out of the three categories. Instead of having two sources of data, which can vary, these tests only have one. These tests also present the general behavior of energy consumption and data values in an operand. The negate instruction was used for these tests, and produced results that can be seen in Figure 5.8.

It can be seen that the larger the number of 0 bits in the source register, the greater the amount of current drawn into the processor. This behavior was found to be followed by the next tests, which have different operand formats.

Three register operands

The second batch of tests were designed to uncover the effect of data values in one of the source registers. Values in the destination register were ignored since they are dependent on the source registers as well as the instruction being executed. The results show that data in the destination register had little or no correlation to current drawn. This makes sense because the destination register value before the instruction is of no consequence to the instruction.

The base instruction used was \textit{addR3, R4, R5}. This format was kept consistent across all tests.
to reduce interference from other factors. Figure 5.9 shows the results of holding R4 at 0 while altering the number of bits held low in R5.

Figure 5.9 also shows the results of holding R5 at 0 while altering the number of bits held low in R4. The two lines converge where both R4 and R5 are 0. This makes sense since those instructions are the same. The two lines do not match, however, and have similar but different behavior. The general pattern found in both is that the current drawn increases as the number of 0 bits in each register increases. The difference between each is in the degree of change from increasing the number of 0 bits. At the time of measurement, two proposed reasons were formulated to explain this phenomena. The first hypothesis was that the particular register specifier (i.e., R4 versus R5) had a complex correlation with energy consumption, where the energy consumption profile of using a specific register varied more than just a simple vertical shift in the profile. Turning back to the register specifier operand tests in the previous section, this hypothesis was ruled out due to the following explanation: altering the register number merely shifts the generic profile vertically and does not alter the slope or the rate at which the current draw changes with respect to the register number under test. The second hypothesis was that the operand position (i.e., first operand or second operand) determined the generic behavior of energy consumption relative to the data held.

Figure 5.8: Current Draw for the Negate Instruction
Figure 5.9: Current Draw versus Data Values in Source Operands (Test Case 1)

within the registers. The second batch of tests confirmed this explanation. Figure 5.10 shows the same tests run for Figure 5.9 except that when R4 and R5 were held constant, they were done so at the value $0xFFFFFFF$. In this case the two lines converge when both operands are $0xFFFFFFF$. By combining Figures 5.9 and 5.10 together, a relationship can be derived that shows the current profile dependent on the value in each of the source registers. Figure 5.11 attempts to show this in a two-dimensional graph, which shows a simple way of deriving the relationship. By drawing a line that intersects the points where the lines converge, the relationship between data in the source registers and current drawn can be found. The dependency between current draw and data values held in the source operands is such that the amount of current drawn increases when there are more 0 bits in the data. The amount of current drawn by an instruction is dependent on both of its source operands.

**Two register and immediate value**

Figure 5.12 shows the effects of varying the immediate values for three instructions, which utilize a two register and one immediate value format. The two registers are initialized to zero.

The graph shows that the current draw increases with the number of 0 bits, which is an expected result. Note that the graph only shows the behavior of current for 16 bits. This is because the
Figure 5.10: Current Draw versus Data Values in Source Operands (Test Case 2)

Figure 5.11: Current Draw versus Data Values in Source Operands (Combined Results)
immediate value in the “addi” instruction is a 16-bit value, which is sign extended and added to the source register.

5.2 Testing the Integer Units

The PowerPC 750 has two integer units. Only one of the units can execute all integer instructions, while the other cannot execute divide or multiply instructions. For the rest of this paper, the integer units will be referred to as either IU1 or IU2. IU1 is the integer unit that can execute all integer instructions including multiply and divide instructions, IU2 can execute all integer instructions except multiply and divide instructions. Execution of these integer units occurs simultaneously, so testing each one independently is not a simple test of specifying which integer unit to test.

5.2.1 Isolating IU1 and IU2

Since it is known that IU2 cannot execute divide or multiply instructions, it is an easy task to isolate execution of instructions only on IU1. If only multiply or divide instructions are being executed, IU1 will be running while IU2 will be idle. Thus, measuring the current when executing multiply or divide instructions should reflect the current draw of IU1 only. However, it is important to note that the measurement will not be the current draw of only IU1, but will include other factors such as register file accesses, instruction decoding, etc. These other factors can be disregarded for
this investigation since those factors are kept constant for measurements, which are being compared to each other. Isolating the other IU2 is not as simple. To measure the energy consumption of IU2, two tests were devised,

The first test looped the “mullw” instruction to see the energy consumed by IU1 only. However, the number of cycles required for a “mullw” instruction is dependent on its second operand value according to [7]. Analysis of this dependency is left for a later section in this chapter. However, for analysis of isolating the integer units, the results are also displayed in this section for comparison purposes. The important thing to realize from the “mullw” instruction test is that there are variable number of stalls, which attribute to different energy consumption values. Thus, the results are shown in millijoules instead of current draw, since instruction latency is a factor in energy consumed per instruction. The results of this first test reflect the energy consumption behavior of IU1 only.

The second test alternated execution of the “mullw” instruction with the “add” instruction. In this test, both integer units are executing. However, it can be assumed that IU2 is executing only the “add” instructions since IU2 cannot execute multiply instructions. Additionally, the IU1 was assumed to be executing only the “mullw” instruction. Keeping things simple, it can be seen that this test is composed of two components: executing “mullw” in IU1 and “add” in IU2.

Since the energy consumption of IU1 executing the “mullw” instruction was found in the first test, the energy consumption of IU2 executing the add instruction can be found by calculating the difference. Figure 5.13 shows the results of both tests,

The difference between the tests can be seen, which represents the energy consumption of IU2. Calculating the difference, IU2 consumed 0.003 mJ per instruction it executed. The additional energy consumed for IU2 was constant for each case of the “mullw” instruction.

5.2.2 Execution of both IUs

Characterizing the energy consumption behavior of both IUs executing simultaneously was done by performing a simple loop of instructions. The “add” instruction was performed while varying the registers used for operands. The instructions were sequenced to make sure that there would not be any dependency problems. The test utilized all of the general-purpose registers to calculate the
Figure 5.13: Energy Consumption of the Integer Units

average energy consumption of both IUs, with the least amount of interference from the register file. The results have been combined with the next section, since they do not represent much relevance by themselves.

5.2.3 Testing data forwarding

To increase performance, many current processors support data forwarding. Data forwarding occurs when there is an instruction that is dependent on the results of a previous instruction that has not completed yet. Data forwarding provides the results of the previous instruction before it is written back to the register file.

To test the energy consumption in a data dependency event, the same instructions that were executed in the batch of tests described in Section 5.2.2 were sequenced to contain data dependencies on every instruction. The observed results show the current draw for executing these same instructions in addition to the amount of extra energy needed to perform data forwarding. Calculating the difference between the results found here and the results from the non-data-dependent sequence of instructions should reflect the energy consumed by only the data forwarding logic.
Figure 5.14: Current Draw for Data Dependencies

To be more comprehensive, more than one test was run to see the results of multiple data dependencies. The results can be seen in Figure 5.14.

Figure 5.14 also shows the current draw for the same instructions sequentialized to have no dependencies. The differences between having data dependencies and having no data dependencies reflect the energy consumption for the processor to forward data.

Notice that the energy consumed for the processor to forward data is less than when the data forwarding is not needed. This can be explained by the fact that when there is a data dependency, the values needed by the instruction will not be found in the register file. Hence, the data is forwarded through some mechanism. This mechanism appears to consume less power than accessing the register file in the case where no dependency occurs. Thus, accessing the register file means that one or more ports are needed to read data from the registers. Additionally, the data bus lines are utilized once more to transfer the data from the register files to the integer units. According to the results, this functionality to execute a nondependent instruction consumes more energy than the functionality to perform data forwarding in the case of a data dependent instruction.

Another interesting phenomenon that can be seen in Figure 5.14, is the large difference in current
draw when the number of instructions with data dependencies increases from two to four. The
reason for this can be explained by limited throughput of instructions. Since there are two integer
units, which can execute instructions at the same time, the maximum throughput of instructions is
two instructions per cycle. In the case where there are four or more data-dependent instructions,
the sequence of instructions still allows a throughput of two instructions per cycle. This is because
the data dependencies never occur between two instructions that are being executed in both integer
units at the same time. The data dependencies only cross the two stages in the pipeline of the
processor, when the instructions finish executing in the integer units and are in the process of
writing the results back to the register file. With only two data-dependent instructions, however,
the sequence creates a scenario where the two instructions, which are dependent on the result of
each other, are located in both integer units. This causes the throughput of instructions to drop
to one instruction per cycle, since the instruction in IU1 must wait for the instruction in IU2 to
generate a result, and vice versa. The result of limited throughput is a decrease in the current
draw, which makes sense since the integer units are alternating from being idle to executing, but
never executing at the same time.

5.3 Testing the Cache

The cache is by far the most complex unit in terms of variable factors involved. Comprehensively
exercising all these factors is beyond the scope of this paper. Thus, tests were performed on events
that occur more frequently in the cache: cache hits and cache misses. The amount of testing was
simplified further to only include cache hits and misses in the level 1 cache. Other tests performed
included memory addressing, data value, and address translation effects. For these tests, the
instruction used was an “lwz” hit to limit the conflict with other factors.

5.3.1 Memory address effects

Memory addressing refers to the value of the address locating the piece of memory that is needed.
The instruction used was “lwz”, which has the following format: lwz rD, d(rA).

The effective address is calculated by adding d (a 16-bit value) to the value in rA. The first batch
of tests varied rA, while the second batch of tests varied d. The results are shown in Figure 5.15.
Figure 5.15: Current Draw versus Memory Address Value

The results shown in Figure 5.15 do not have the expected behavior. Looking further at the instruction, it was determined that the values in the operands are not as important as the effective address that is calculated.

5.3.2 Data value effects

More significant than memory addressing effects is the piece of data being read or written to the cache. To test this effect, a batch of tests were performed where the address being accessed was held constant, and the value at that specific memory location was initialized to some value that was varied. Figure 5.16 shows the results, which follows the trend that the more 0 bits, the more energy consumed.

5.3.3 Address translation

There are four address translation modes for load and store operations, Table 5.1 displays each address translation mode for each class of load instruction and their measured current.

Table 5.2 shows the table of store instructions. The tables show the different class of instructions
Figure 5.16: Current Draw versus Data Read from Memory

Table 5.1: Load Instructions

<table>
<thead>
<tr>
<th>Instruction Mnemonic</th>
<th>Operand Format</th>
<th>Description</th>
<th>Current</th>
</tr>
</thead>
<tbody>
<tr>
<td>lbz</td>
<td>rD,d(rA)</td>
<td>EA ← d + rA</td>
<td>1.957</td>
</tr>
<tr>
<td>lbzu</td>
<td>rD,d(rA)</td>
<td>EA ← d + rA</td>
<td>1.971</td>
</tr>
<tr>
<td>lbzx</td>
<td>rD,rA,rB</td>
<td>EA ← rA + rB</td>
<td>2.006</td>
</tr>
<tr>
<td>lbzux</td>
<td>rD,rA,rB</td>
<td>EA ← rA + rB</td>
<td>2.024</td>
</tr>
<tr>
<td>lhz</td>
<td>rD,d(rA)</td>
<td>EA ← d + rA</td>
<td>1.962</td>
</tr>
<tr>
<td>lhzu</td>
<td>rD,d(rA)</td>
<td>EA ← d + rA</td>
<td>1.975</td>
</tr>
<tr>
<td>lhzx</td>
<td>rD,rA,rB</td>
<td>EA ← rA + rB</td>
<td>2.010</td>
</tr>
<tr>
<td>lhzux</td>
<td>rD,rA,rB</td>
<td>EA ← rA + rB</td>
<td>2.027</td>
</tr>
<tr>
<td>lha</td>
<td>rD,d(rA)</td>
<td>EA ← d + rA</td>
<td>1.964</td>
</tr>
<tr>
<td>lhau</td>
<td>rD,d(rA)</td>
<td>EA ← d + rA</td>
<td>1.979</td>
</tr>
<tr>
<td>lhax</td>
<td>rD,rA,rB</td>
<td>EA ← rA + rB</td>
<td>2.014</td>
</tr>
<tr>
<td>lhaux</td>
<td>rD,rA,rB</td>
<td>EA ← rA + rB</td>
<td>2.032</td>
</tr>
<tr>
<td>lwz</td>
<td>rD,d(rA)</td>
<td>EA ← d + rA</td>
<td>1.972</td>
</tr>
<tr>
<td>lwzu</td>
<td>rD,d(rA)</td>
<td>EA ← d + rA</td>
<td>1.987</td>
</tr>
<tr>
<td>lwzx</td>
<td>rD,rA,rB</td>
<td>EA ← rA + rB</td>
<td>2.020</td>
</tr>
<tr>
<td>lwzux</td>
<td>rD,rA,rB</td>
<td>EA ← rA + rB</td>
<td>2.037</td>
</tr>
</tbody>
</table>
Table 5.2: Store Instructions

<table>
<thead>
<tr>
<th>Instruction Mnemonic</th>
<th>Operand Format</th>
<th>Description</th>
<th>Current</th>
</tr>
</thead>
<tbody>
<tr>
<td>stb</td>
<td>rD,d(rA)</td>
<td>EA ← d + rA</td>
<td>1.936</td>
</tr>
<tr>
<td>stbu</td>
<td>rD,d(rA)</td>
<td>EA ← d + rA</td>
<td>1.936</td>
</tr>
<tr>
<td>stbx</td>
<td>rD,rA,rB</td>
<td>EA ← rA + rB</td>
<td>1.981</td>
</tr>
<tr>
<td>stbux</td>
<td>rD,rA,rB</td>
<td>EA ← rA + rB</td>
<td>1.980</td>
</tr>
<tr>
<td>sth</td>
<td>rD,d(rA)</td>
<td>EA ← d + rA</td>
<td>1.938</td>
</tr>
<tr>
<td>sthu</td>
<td>rD,d(rA)</td>
<td>EA ← d + rA</td>
<td>1.938</td>
</tr>
<tr>
<td>sthx</td>
<td>rD,rA,rB</td>
<td>EA ← rA + rB</td>
<td>1.983</td>
</tr>
<tr>
<td>sthux</td>
<td>rD,rA,rB</td>
<td>EA ← rA + rB</td>
<td>1.983</td>
</tr>
<tr>
<td>stw</td>
<td>rD,d(rA)</td>
<td>EA ← d + rA</td>
<td>1.946</td>
</tr>
<tr>
<td>stwu</td>
<td>rD,d(rA)</td>
<td>EA ← d + rA</td>
<td>1.946</td>
</tr>
<tr>
<td>stwx</td>
<td>rD,rA,rB</td>
<td>EA ← rA + rB</td>
<td>1.991</td>
</tr>
<tr>
<td>stwux</td>
<td>rD,rA,rB</td>
<td>EA ← rA + rB</td>
<td>1.990</td>
</tr>
</tbody>
</table>

which are first seperated into two categories: loads and stores. Furthermore each of these categories can be further split into the number of bytes they are transacting. Each of these categories can be executed in two addressing modes: direct and indirect. Direct address translation calculates the effective address (EA) by adding the value specified by $d$ to the value in $rA$. Indirect address translation requires referencing two values from $rA$ and $rB$ and adding them together to get the EA. Furthermore, the instruction can either update $rA$ to reflect the result of the EA caculation or just leave $rA$ as it is. For analysis, the energy consumption for loads and stores can be split into the following four factors:

1. Whether it is a load or store
2. The number of bytes required
3. Address translation mode
4. Whether the EA is placed in $rA$ or not

Figures 5.17 - 5.19 show the current draw trend for these four factors.

In general, loads consume more energy than stores. However, this behavior is dependent on other factors such as whether the cache is operating in write-back or write-through mode. Additionally, the test bench is set up only to measure the current drawn solely by the processor. Thus, the addition of an L2 cache is ignored at this time. It can also be observed from Figure 5.17 that
current increases when more bytes of data are required. This can be explained by the number of bit lines required for use on the data bus.

Figure 5.18 shows the energy consumption differences for the two addressing modes. Indirect address translation consumes more energy than direct address translation. Utilization of one more register tends to increase the energy consumption.

Lastly, placing the EA in $rA$ appears to increase the energy consumption as shown by Figure 5.19. Most of the observations can be generalized by the following: the more work required by an instruction, the more energy consumed. Although this seems obvious, the programmer is usually not aware of what is happening inside a computer. Additionally, some functional units in the processor do not follow this generalization, i.e., the ALU.

### 5.3.4 Cache hits and misses

To test the differences in energy consumed during a cache miss, tests were performed to exercise both cache hits and misses. The results of such tests are shown in Figures 5.20 and 5.21.

In Figure 5.20 it can be seen that the current required to operate the processor during a cache miss is lower than a cache hit. However, this measurement is not indicative of the overall work
Figure 5.18: Current Draw versus Address Translation Mode

Figure 5.19: Current Draw versus Updating Effective Address Relation
needed. Cache accesses that hit in the level 1 cache take only one processor cycle to execute, whereas a miss in the level 1 cache takes significantly more cycles. Therefore, calculating the energy consumed to execute each instruction requires multiplying the power that the processor is operating at by the time it takes to complete a cache miss.

Figure 5.21 shows a significant difference in the amount of energy consumed during a cache hit and a cache miss. A cache miss, though it may require less power, consumes much more energy to complete than a cache hit. Furthermore, these results are indicative of a cache hit in the level 2 cache if there was a miss in the level 1 cache. The number of cycles required for a memory access that misses in both caches is significantly higher. Thus, the energy consumption of memory accesses appears to be dominated by the number of processor cycles required to execute the access.

### 5.4 Testing the Floating-Point Unit

The floating-point unit (FPU) on the PowerPC 750 is split into a three stage pipeline to increase throughput. Effectively, up to three floating-point instructions may execute simultaneously.
Figure 5.21: Cache Energy Consumption

However, not all floating-point instructions can be pipelined nor do all having the same timing characteristics in each pipeline of the FPU. For example, executing the “fres” instruction takes 10 clock cycles and cannot be pipelined. Therefore, if subsequent floating-point instructions exist in the instruction dispatch queue, a stall will occur that lasts at least 10 cycles. Table 5.3 shows the timing characteristics for floating-point instructions.

Note that in the timing column, if the number is by itself, it represents nonpipelinable instructions. The numbers separated by dashes show the latency of each stage of the FPU’s pipeline.

Table 5.3: Floating-Point Unit Timing

<table>
<thead>
<tr>
<th>Timing</th>
<th>Instructions</th>
</tr>
</thead>
<tbody>
<tr>
<td>3</td>
<td>mtfsb0[1], mtfsb1[1], mtfsi[1], mtfsf[1]</td>
</tr>
<tr>
<td>10</td>
<td>fres[1]</td>
</tr>
<tr>
<td>17</td>
<td>fdivs[1]</td>
</tr>
<tr>
<td>33</td>
<td>fdiv[1]</td>
</tr>
<tr>
<td>2-1-1</td>
<td>fnadd[1], fnsub[1], fnmul[1], fnmadd[1], fnmsub[1]</td>
</tr>
<tr>
<td>1-1-1</td>
<td>all other floating-point instructions</td>
</tr>
</tbody>
</table>
An important note to remember is that the FPU operates on 64-bit values. The floating-point registers are 64 bits wide and the operations available operate on 64 bits. There are two formats for floating-point operations: single-precision and double-precision. Single-precision utilizes 32 bits to represent floating-point numbers, whereas double-precision utilizes 64 bits. The PowerPC has the capability of operating on both precision formats, however, single-precision data is converted to double-precision when used in an operation.

5.4.1 Floating-point registers

Since the registers that a floating-point instruction uses are different than the general purpose registers, the same operand tests were run again. However, testing of the data values held in these registers was left out due to time considerations. The instruction used for these batches of tests were the “fadd” instruction which can be pipelined and which exhibits no stalls once the instruction has filled the FPU, as well as one other factor discussed in the next paragraph.

Looking at Figures 5.22 - 5.28, it can be seen that the floating-point registers behaved in the exact same way as the general purpose registers. Any dependencies between the instructions decreased the amount of current drawn. However, extra stall cycles were incurred, and thus, the total energy consumed per dependent instruction was much higher than for a independent instruction.

5.4.2 Floating-point loads and stores

Floating-point loads and stores are similar to integer loads and stores. The difference is that floating-point registers are 64 bits wide. Since floating-point loads and stores are similar to integer loads and stores, the behavior was assumed to also be similar. A small set of tests were performed to confirm this assumption. Figure 5.29 shows the general behavior of floating point loads and stores.

The instruction used for loads was “lfs,” which loads a word from memory and converts it from 32-bit floating-point format to 64-bit floating-point format, before placing the result in a register. Figure 5.29 shows the linear behavior, which is similar to integer loads and stores, As was the case for integer loads and stores, the current draw was dependent on the number of 0 bits in the data.
Figure 5.22: Current Draw When Destination Register is F0

Figure 5.23: Current Draw When Destination Register is F1
Figure 5.24: Current Draw When Destination Register is F3

Figure 5.25: Current Draw When Destination Register is F7
<table>
<thead>
<tr>
<th>Number of Bits in Source Register</th>
<th>Current (A)</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>1.700</td>
</tr>
<tr>
<td>0.5</td>
<td>1.750</td>
</tr>
<tr>
<td>1</td>
<td>1.800</td>
</tr>
<tr>
<td>1.5</td>
<td>1.850</td>
</tr>
<tr>
<td>2</td>
<td>1.900</td>
</tr>
<tr>
<td>2.5</td>
<td>1.950</td>
</tr>
<tr>
<td>3</td>
<td>2.000</td>
</tr>
</tbody>
</table>

**Figure 5.26: Current Draw When Destination Register is F15**

**Figure 5.27: Current Draw When Destination Register is F31**
Figure 5.28: Average Current Draw for Floating-Point Registers

Figure 5.29: Floating-Point Loads and Stores
being transferred. The same basic result was also found for floating-point stores. The instruction used for stores was “stfs”, which first converts the value in the floating-point register into a 32-bit format and then loads the word into memory. One interesting point that can be observed between Figures 5.29 and 5.16 is that the curves are nearly identical. This makes sense, since the instruction “lfs” only loads a 32-bit value from memory, which is the same thing that the integer load instruction “lwz” does.

The surprising result was when comparing floating-point loads to floating-point stores. When there are fewer 0 bits, the floating-point stores draw more current than floating-point loads. On the other end of the results, when there are more 0 bits in the floating-point register, the floating-point loads draw more current than the floating-point stores. The explanation for this result is in due to a slight oversight when developing the tests. The IEEE 754 standard for floating-point calculations defines the special numbers \( \infty \), \(-\infty\), and \( \text{NAN} \) (Not-A-Number). It turns out that the tests developed were not designed with specific floating-point numbers in mind, but rather the number of 0 bits loaded into the registers. The surprising result is an effect of performing floating-point loads and stores on these special floating-point numbers. The PowerPC 750 creates an exception to these special numbers and the execution becomes different. Therefore, the results shown are a combination of results including normal floating-point loads and stores, and floating-point load and store exceptions.

5.4.3 Fixed-point versus floating-point

Since the data referenced in memory is a maximum of 32 bits, the data loaded must undergo conversion from single precision to double precision. Though the decimal value does not hold much relevance in relating energy consumption with instructions, the comparison of fixed-point versus floating-point has more of a dependence on the value. The reason lies within the bit-level representation of the value. Floating-point calculations follow the IEEE 754 standard, which interprets floating-point numbers in terms of a sign bit, a set of exponent bits, and a set of mantissa bits. The floating-point format can be seen in here: \( SE_0...E_{7}M_{0}...M_{22} \) (32-bit representation) and \( SE_0...E_{10}M_{0}...M_{51} \) (64-bit representation).

The single precision format requires 32 bits, while the double-precision requires 64 bits. Con-
version from single-precision to double-precision does not require any 1 bits, but since there are 64
bits instead of 32, there are more 0 bits. Since all floating-point operations in the PowerPC 750 use
double-precision, it is not important to compare the single-precision format with double-precision.
However, the comparison between fixed-point and floating-point is affected by the different represen-
tations. First, the 32-bit representation of an integer value is different than its floating-point
representation. Therefore, the number of 0 and 1 bits required to represent an integer value is
different for integers and floating-points. Second, the use of 64-bits instead of 32-bits means, that
there are twice as many bit lines that contribute to the energy consumed by a floating-point in-
struction. The addition of pipeline stages within the FPU was the last factor to be considered
when comparing fixed-point and floating-point instructions. The IUs spit out two instructions
per clock cycle assuming no dependencies and nonmultiply or nondivide instructions. The FPU
has a throughput of one instruction per clock cycle once the pipeline has been filled with basic
floating-point instructions.

For the comparison test, the “add” and “fadd” instructions were used. Both general purpose
and floating-point register files were all initialized to zero. The decimal values of the operands were
the same to provide a proper comparison of the different instructions providing the same output
desired. Figure 5.30 shows the results of the tests.

The results show the current draw comparison for fixed and floating-point calculations of the
same numbers. The current draw for fixed-point calculations appears to be higher than floating-
point calculations. However, remember that there are two integer units, so two fixed-point calcula-
tions are executed per cycle. Since there is a difference in instructions per cycle (IPC), average
power needs to be translated to energy consumption for proper analysis. Figure 5.31 shows the
energy consumption differences for fixed-point and floating-point calculations.

In terms of energy consumption per calculation, fixed-point calculations consume almost half
the energy consumed by the FPU to perform the same function. However, an interesting point can
be made here. Disregarding the initial load time for the FPU to fill its specific pipeline stages, the
energy consumed by the FPU and one of the IUs is about the same.
Figure 5.30: Fixed-Point versus Floating-Point (Current Draw)

Figure 5.31: Fixed-Point versus Floating-Point (Energy Consumption)
Table 5.4: Integer Unit Timing

<table>
<thead>
<tr>
<th>Timing</th>
<th>Instructions</th>
</tr>
</thead>
<tbody>
<tr>
<td>19</td>
<td>divw</td>
</tr>
<tr>
<td>2,3,4,5,6</td>
<td>mullw</td>
</tr>
<tr>
<td>2,3,4,5</td>
<td>mullw</td>
</tr>
<tr>
<td>2,3</td>
<td>mulli</td>
</tr>
<tr>
<td>2</td>
<td>tw, twi</td>
</tr>
<tr>
<td>1</td>
<td>all other integer instructions</td>
</tr>
</tbody>
</table>

The only reason the fixed-point calculations consumed less energy than the floating-point calculations is that there is extra hardware to accommodate higher instruction throughput for integer instructions. Thus, comparing one FPU to one IU would produce different results.

5.5 Miscellaneous Tests

This section discusses some other tests that were performed. Though they have not been categorized, they are tests of interest and for future work should be expanded upon.

5.5.1 Stalls

A factor that was controlled in the previous tests was the number of stalls required by the multiply instruction. As can be seen from Table 5.4, the “mullw” instruction can take up at least 2 cycles minimum and 5 cycles maximum, depending on the value in the second source register.

Due to the variant number of stalls that can be incurred by the multiply instruction, tests were performed to observe the behavior of the multiply instruction and its dependency on its operand values. Figure 5.32 shows the average current draw for different values in the second source register used in the “mullw” instruction. The sawtooth shape displays the effects of the different stalls. Each “tooth” in the figure shows how the energy consumption is dependent on the number of 0 bits in the source register. The effect of having multiple teeth shows that each tooth corresponds to the number of stalls. The smallest tooth corresponds to the “mullw” instruction being stalled five processor cycles every time, whereas the largest tooth corresponds to only two stall cycles each time the “mullw” instruction is executed. To calculate when the “mullw” instruction consumes...
more energy, the average power must be multiplied by the number of stall cycles. Thus, the energy consumption characteristic of the “mullw” instruction in relation to the operand values can be seen in Figure 5.33.

When relating the energy consumption to the “mullw” instruction, the only factor that makes a substantial difference is the number of stalls. The more stalls, the more energy consumed, even though the average power is lower.

5.5.2 Multiple loads and stores

The PowerPC 750 provides special load and store instructions, which can transfer more than one word of data. The “lhw” and “stmw” specify the number of words they wish to transfer by choosing the register number that represents the starting word. Each register number sequentially higher in number either holds or will hold each sequential word in memory. Therefore, to transfer five words, R27 is specified as the starting register for the data transfer. The rest of the registers up to R31 are then chosen automatically for the rest of the transfer.

The tests tried to characterize the energy consumption dependent on how many words were

Figure 5.32: Current Profile of Multiply Instruction
being transferred. The number of processor cycles could not be measured. It was assumed that every word transferred required an additional processor cycle. This assumption was made to draw useful results from the measurements. Figure 5.34 shows the actual current measurements for both “lmw” and “stmw.” Figure 5.35 shows the same results, but as an energy consumption value. The difference in energy consumption with respect to the number of words can be seen to be, in general, linear. The difference between whether the instruction is performing a read or write from memory agrees with the results from integer stores and loads: loads consume more energy than stores.
Figure 5.34: Current Draw for Multiple Load and Store Instructions

Figure 5.35: Energy Consumption of Multiple Load and Store Instructions
CHAPTER 6

CONCLUSIONS

6.1 Summary

This paper presented a power measurement and analysis methodology that removes a fixed level of abstraction, e.g., transistor-level, gate-level, or instruction-level. The concept of an event was presented and is the basis for characterizing the power consumption behavior of a processor and application. The techniques used to measure power were derived from a previous study [1]. Analyzing the results of the tests, it was concluded that the power consumed by a processor varies according to certain factors. The most dominant factors, which dictate the amount of power consumed, were stalls and data values that were used for operands. In all the integer instruction tests, the number of 0-bits in the source operands was directly correlated with average power: the more 0-bits, the higher the average power. This is caused by the use of precharged bit lines and dynamic logic in the PowerPC 750.

In the case of stalls, the more stalls an instruction required, the more energy the instruction consumed. Using average power as a metric to characterize stalls was not very indicative, therefore, the amount of energy consumed for an instruction was calculated to see the effect of a stall.

Memory accesses were also examined, and the general conclusion was that cache hits consume less energy than cache misses. Again, average power was regarded as an irrelevant metric, since they were not indicative of the total energy required for a cache event.

The distinction between power and consumed energy allowed different comparisons in the analysis. Tests that had no stalls or consistent number of consumed cycles per instruction were compared with the average power metric. Tests that had varying numbers of stalls required the use of the
energy consumed metric to gain a relevant measurement.

The FPU was also tested and found to have similar results to IU testing. One test sought to see if there was any difference when computing fixed-point calculations and floating-point calculations that output the same results. The results of this test showed inconsistent results due to the different encoding of decimal numbers in 32-bit integer format and 64-bit floating-point format. It was concluded that the energy consumed differed with respect to the number of 0-bits in the source operands.

Data forwarding was tested to see how much energy was consumed to utilize that logic. It turns out that data forwarding consumes less power than accessing the register file.

6.2 Future Outlook

Though this study covered many different units of the processor, there is much more work to be done. Other units that should be tested are the branch prediction unit, the level 2 cache, I/O, and several other units. Fully testing the processor is vital to creating an accurate power modeler. Future research could apply this analysis and measurement methodology to other processors.

The next step after exhaustively testing the processor would be to develop an event-level power model of the processor. The model would produce estimates of power consumption and verify these through the test bench. Eventually, the power model will be fine-tuned so that fast and accurate estimates could be made and new ideas in optimization could be tested.

Other future possibilities would be a real-time power profiler that would run alongside an application and profile the amount of power consumed through real-time measurements. Since power can be modeled by specific events occurring, the profiler could take measurements on each event and create a profile of the application’s power consumption behavior. Stepping up a notch in execution, the profile of several applications could be run, which would be invaluable for scheduling applications to fit within a power budget.

Besides being a preliminary step to future research, this study serves as an insight into the behavior of a processor and its power consumption characteristics. Hopefully, with more research and development, an event-level power estimation tool can be developed that produces fast and accurate results.
REFERENCES


