UNIVERSITY OF SOUTHAMPTON

FACULTY OF PHYSICAL SCIENCES AND ENGINEERING

Electronics and Computer Science

An Improved Instruction-Level Power and Energy Model for RISC Microprocessors

by

Wei Wang

Thesis for the degree of Doctor of Philosophy

March 2017

UNIVERSITY OF SOUTHAMPTON

ABSTRACT

FACULTY OF PHYSICAL SCIENCES AND ENGINEERING
Electronics and Computer Science

Doctor of Philosophy

AN IMPROVED INSTRUCTION-LEVEL POWER AND ENERGY MODEL FOR RISC MICROPROCESSORS

by Wei Wang

Recently, the power and energy consumed by a chip has become a primary design constraint for embedded systems and is largely affected by software. Because aims vary with the application domain, the best program is sometimes the most power or energy efficient one rather than the fastest. However, there is a gap between software and hardware that makes it hard to predict which code consumes the least power without measurement. Therefore, it is vital to discover which factors can affect a program's power and energy consumption.

In this thesis we present an instruction level model to estimate the power and energy consumed by a program. Firstly, instead of studying the different instructions individually, we cluster instructions into three groups: ALU, load and store. The power is affected by the percentage of each group in the program. Secondly, the power is affected by the instructions per cycle (IPC) of the program since IPC can reflect how fast the processor runs.

There are three advantages of this method, and the first one is conciseness. The reason is that it does not consider the overhead energy as an independent factor or the operand Hamming distance of two consecutive instructions.

The second one is accuracy. For example, the errors of our method across different benchmarks with different processors on the development boards are all less than 10%.

The last and the most important advantage of this method is that it can apply to different processors, such as OpenRISC processor, ARM11, ARM Cortex-A8, and a dual-core ARM Cortex-A9 processor. We have demonstrated that the previous instruction level power/energe model cannot be extended to superscalar processors and multi-core processors.

Contents

List of Figures

List of Tables

Acknowledgements

I would like to thank all those who helped me during my studies. I am very thankful to my supervisor, Prof. Mark Zwolinski. During the study, Mark gives me his unconditional counsel and support, and encouraged me to overcome the problems I face.

I also want to thank my colleagues who helped me with this project, in particular Jeff Hooker and Sheng Ye, who helped me modify the board.

I would also like to express my thanks to my parents who have loved me, supported me and sponsored me during my studies. I would like to thank my girlfriend for her support and loyalty since we lived in different countries for a long time.

Finally, I want to thank my friends and house mates, especially Meng Tian, and we have a lot of good memories during the four years of our studies.

Abbreviations

BTAC	**B**ranch **T**arget **A**ddress **C**ache
BTB	**B**ranch **T**arget **B**uffers
CISC	**C**omplex **I**nstruction **S**et **C**omputer
CPI	**C**yscle **P**er **I**nstruction
DVFS	**D**ynamic **V**oltage and **F**requency **S**caling
CPU	**C**entral **P**rocessing **U**nit
EPI	**E**nergy **P**er **I**nstruction
GHB	**G**lobal **H**istory **B**uffers
GPGPU	**G**eneral-**p**urpose computing on **G**raphics **P**rocessing **U**nit
ILP	**I**nstruction **L**evel **P**arallelism
IPC	**I**nstruction **P**er **C**lockcycle
ISA	**I**nstruction **S**et **A**rchitecture
MMU	**M**emory **M**anagement **U**nit
OoO	**O**ut of **O**rder
OS	**O**perating of **S**ystem
SDF	**S**tandard **D**elay **F**ormat
TLBs	**T**ranslation **L**ook-aside **B**uffers
TLP	**T**hread **L**evel **P**arallelism

Nomenclature

A Ampere
C Capacitance
F Frequency
J Joule
P Power
p Percentage of
V Volt W Watt

Chapter 1

Introduction

Power/energy consumption is a crucial aspect of modern digital design because a lot of mobile devices, such as mobile phones, use batteries as energy sources; this makes power/energy consumption a key issue. To make a low power processor, a number of hardware techniques have been developed, such as clock gating and power gating.

On the other hand, it is not only the architecture of the processor that can affect the power/energy consumption but the program as well. This means that a microprocessor system can have different power/energy consumptions when running different programs. Therefore, in order to reduce the power/energy usage of a program, it is vital to discover the significant factors which affect a program's power/energy consumption.

However, there is a gap between software and hardware that makes it hard to predict which code consumes the least average power/energy. Tiwari and Lee *et al.* state that one of the benefits of an instruction-level analysis is that it provides clues about how to write effective power-saving software applications [1]. An instruction level energy study provides a way to determine how software affects energy consumption, and thereby allows low energy software to be written.

1.1 Motivation and Objects

A number of models for estimating energy consumption at the instruction level have previously been proposed such as data-dependent models and cycle-accurate models [2, 3]. However, there are some disadvantages to these methods.

Firstly, the fabrication technology of CMOS and the density of integration of chips are improving every year. Figure 1.1 shows the transistor counts for microprocessors over 40 years.

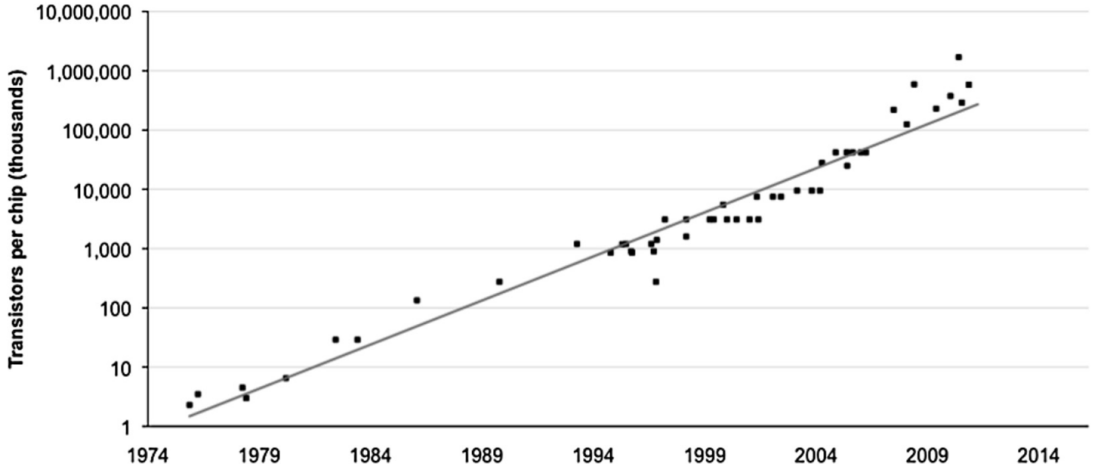


Figure 1.1: Transistor counts for microprocessors over time (thousands) [4].

However, a lot of models and methods are created based on old fabrication technologies. For example, Nikolaos *et al.* used the ARM7TDMI embedded processor ($0.35\mu m$) as the target processor and created an energy model [5]. There is no guarantee that these models can work with modern processors. The reason is that the technology is improving, but the percentage of the static power consumption also increases and may become the major part of the power consumption [6].

Secondly, the computer architecture technology has improved and the pipeline has become deeper and deeper. However, a lot of models are based on old structures such as the Intel 486DX2 [7, 8] and ARM7TDMI [9]. The architectures of modern processors are not the same as old processors. For example, a superscalar processor can fetch more than one instructions in a single clock cycle, thus is more powerful than a single scalar processor. They are already commonly used in modern mobile phones such as iphone 4 and 5. On the other hand, the design of the instruction pre-fetch unit, the cache, the out-of-order pipeline, and the branch prediction unit have improved as well. Therefore, the old models may not apply to the structure of modern processors and new simple models may suit better.

Moreover, multi-core processors are also widely used in desktop, laptop and mobile systems. Amdahl's law shows that the maximum speedup of a program is limited by the percentage of time spent in the sequential fraction [10]. Thus, the power and energy consumption of multi-core processors is affected by the application. However, the power/energy model of a multi-core processor has not been well studied at instruction level.

Thirdly, even though some of the previous work could be extended to modern processors, some is hard to use, such as data dependency models. For example, a lot of models are concerned with the types and operand values of each different instruction individually, such as the energy models of the Intel 486DX2 [7, 8] and ARM7TDMI processors [5]. If

a program contains billions of instructions, the models have to trace what these billions of instructions are and sum the different energy of each instruction. However, some factors of these old models may not be important any more or could be replaced by some easily gathered factors. Thus, a new concise model which needs fewer inputs and easily collected inputs would be more convenient to use.

The objectives of this thesis are listed below:

1. Find methods to create instruction level power models for RISC processors. The method can be extended for different RISC processors, such as scalar and super-scalar processors. The accuracy should be reasonable, less than 10%.
2. The methods should be simple and not need a lot of data for creating the model.
3. The created power models should be concise and easily used. For example, the input values should be few and also easy to collect.
4. Find an easy method for estimating the energy of programs with different RISC processors. The estimation error should be reasonable, better than 10%.

1.2 Contribution

In this thesis, we present a new instruction-level power model and we prove this method works in a number of different types of RISC processors including a scalar processor (OpenRISC), an Out of Order (OoO) scalar processor (ARM11), a superscalar processor (ARM Cortex-A8), and a dual-core processor (each core is an ARM Cortex-A9). On top of this, we demonstrate that the power model can be extended to an energy model easily in each case. Compared with previous methods, our method has several advantages:

1. It provides equal or better accuracy compared with [5, 11, 12]. The errors of all of benchmarks in different development boards are less than 10%.
2. It does not need a cycle-accurate simulation and so improves the simulation speed. A lot of work needs cycle-accurate analyses, such as [13, 14], and if the model takes pipeline stalls into consideration, a cycle-accurate simulation of the program may be needed [15].
3. Except for the case study of the OpenRISC processor, all of the other processor case studies, including ARM11, ARM Cortex-A8, and ARM Cortex-A9 Dual-core system, are validated by physical hardware measurements.
4. The model is more concise. A lot of the previous work concerns each instruction individually, which makes the models hard to use [1, 15]. Some work takes too many different aspects into consideration and needs a lot of effort to gather these

input data before using the model. For example, Steinke *et al*. created a model which uses 12-32 parameters for each instruction [16] and the model proposed by Bazzaz *et al*. needs 35 different variables. Our method does not need to identify each instruction individually because we cluster instructions into several different groups. On top of this, there are only three input variables for our model: the IPC (instruction per clock cycle), the distribution of instruction types and the speedup ratio, which is needed by the dual-core power model.

5. Moreover, some previous models use simple or too few benchmarks to test the performance of the model [17, 18]. The benchmarks we choose include several well-known mathematical functions, Mibench, and SPLASH2.

Besides these, we also did some research which has not been well studied by others:

1. We did detailed analysis on the power consumption of the superscalar processor, ARM Cortex A8, at the instruction level. Furthermore, the aspects we have studied include: how the power consumption of a processor is affected by L1/L2 instruction and data cache misses; by different instruction types; by dual-issue restriction; by instruction operands; and by the overhead power cost of two adjacent instructions.

2. We present a concise instruction level power model for the ARM Cortex A8 with good performance. The power consumption varies in different instructions but instead of studying each instruction individually, we have classified different instructions into three classes: arithmetic/logic, load and store. Therefore, only the distribution of each class is considered by the model rather than which instructions they are. On the other hand, the factors which can make the pipeline stall, such as cache misses, are considered as instructions per clock cycle (IPC).

3. We extend this method and create an instruction-level energy model for a dual-core system processor. The speedup ratio is used to show the percentage of runtime for which both cores run and this model shows good performance. Furthermore, we test nine benchmarks from SPLASH2 and the worst errors are 10.83% and 9.18% with the single thread and two thread tests, respectively. The average error of the single thread and two thread tests are 4.6% and 5.95%, respectively.

We have extended the power model to estimate the energy consumed by the processor. Comparing this with other energy models and methods, the advantage is accuracy and ease of use. The reason is that instead of creating an energy model, we divide this complex problem into two simple questions: a power model and the runtime of a program. A power model is easier to create than an energy model and the runtime of a program is one of the easiest variables to measure. Thus, the energy is simply equal to the average power times the runtime of the program.

1.3 Thesis Organization

This thesis is organized into seven chapters.

Chapter 2

Chapter 2 is the literature review of previous work. It presents different models from different aspects, such as the basic models, data-dependent models, and functional level models. Different models are concerned with different factors. For example, the basic energy model covers the base power/energy and overhead energy, which is discussed in Section 2.1. On the other hand, functional level models split the processors into different blocks and study each block separately, which is discussed in Section 2.5. We try to find which factors are the most significant and can be used in our method.

Chapter 3

In this chapter, we choose the OpenRISC OR1200 as the target processor (a 32-bit Harvard architecture scalar RISC processor with a five stage integer pipeline [19]) and present an instruction-level power and energy model. The OR1200 is synthesized using Design Compiler and the power is measured by Primetime. We find that the power of the processor does not change much for different operations and operand switch rates. The IO port power is related to the percentage of store instructions and the cache miss rate. Using linear regression, an accurate IO port power equation is derived. Finally, the total energy cost of a program is estimated from average power and runtime.

Chapter 4

In this chapter, a new instruction-level power/energy model is created for an ARM1176JZF-S (a 32-bit Out of Order scalar RISC processor with an eight stage pipeline, and 32kB data and instruction caches [20]). The instructions are classified into three groups: ALU, load and store, based on the different power costs. The power model includes two factors: the components and the instructions per cycle (IPC) of the program. The energy consumed by the processor is estimated by the average power multiplied by the runtime, which is the same as the energy model of the OpenRISC. Moreover, we prove that the Energy per Instruction (EPI) is inversely proportional to the Instructions per Clock cycle (IPC).

Chapter 5

The method presented in Chapter 4 is extended to an ARM Cortex-A8 (a super scalar processor with a 13 stage pipeline, two ALU pipelines, and one load/store pipeline, and 32kB for both instruction and data caches). Firstly, the power consumption is analysed under different conditions, such as the effect of L1/L2 cache misses, the effect of instruction types, and dual-issue restriction. We find that the previous basic power/energy model (presented in Equation 2.2) does not work. Secondly, we extend the power model

of the ARM11 into the Cortex-A8 and the power is estimated by the components and the IPC. The energy is estimated from runtime and power. The power model is verified with ten benchmarks. Finally, for a superscalar processor, we prove that increasing the IPC without changing the components of a program much may reduce the energy per instruction (EPI).

Chapter 6

This chapter focuses on the power consumption of a dual-core processer, and each core is an ARM Cortex-A9 (an Out of Order (OoO) superscalar processor with dynamic multiple issue technology, an efficient 8-stage pipeline, 32kB L1 instruction and data caches, and 1-MB L2 cache). We assume that the power is affected by three factors: IPC, speedup ratio and the components of a program. Linear regression is used to create a power model. Nine benchmarks from SPLASH2 are used to test the performance of the power model and they are tested in single thread and two thread modes. Moreover, the energy consumed by a program with a single thread is the same as or less than that with multiple-threads. The reason is that although more work needs to be done with multi-threads, such as operating system (OS) schedules, the runtime reduces significantly and the hardware usage is more efficient. Therefore, multi-threading can reduce the runtime without sacrificing energy.

Chapter 7

In this chapter, we discuss the limitations of our method and summarise the steps about how to extend the model to a new RISC processor. This method can only work for RISC processors, but not CISC processors since one RISC instruction only does one thing but one CISC instruction may do a lot of work.

Chapter 8

The final chapter summarises our method and discuss the future work. There are three potential future work ideas. The first one is to extend the method to a more complex system. The second one is to reduce the energy consumption of the system based on the prediction of the power model. The third one is to combine the static program analysis technology with the power model to estimate the energy more quickly.

1.4 List of Publications

1. Wei Wang and Mark Zwolinski, “An improved instruction -level energy model for RISC microprocessors” In, *9th Conference on Ph.D. Research in Microelectronics and Electronics (PRIME 2013) , 24 - 27 Jun 2013.* Villach, AT, , 349-352.

2. Wei Wang and Mark Zwolinski, Mark "An improved instruction-level power model for ARM11 microprocessor" In, *High Performance Energy Efficient Embedded Systems (HIP3ES),* Vienna, Austria, 21 Jan 2014.

Chapter 2

Literature Review

Computer architecture is improving every year. For example, Table 2.1 shows the development of Intel processers. It is clear that the number of transistors is increasing, and the clock rate is faster and faster.

Table 2.1: Performance milestones for Intel processors [21].

microprocessor	80486	Pentium	Pentium Pro	Pentium 4	Core i7
year	1989	1993	1997	2001	2010
Die size(mm^2)	81	90	308	217	240
transistors	1,200,000	3,100,000	5,500,000	42,000,000	1,170,000,000
clock rate(MHz)	25	66	200	150	3333

Moreover, RISC processors are more and more widely used, especially in embedded systems. For example, ARM sold 10 billion units in 2013 [22]. A lot of tablets and smart phones use ARM processors, such as iPad, iPhone4 and 5.

A number of models for estimating the power/energy consumption have previously been proposed. However, since the performance of the processors is improving every year, the fabrication technology, the clock speed, and other factors may affect the power consumption of a processor. Thus, the previous models may not apply to modern processors. The reason is that the power consumption can be divided into two parts: dynamic power and static power. Furthermore, the dynamic power is related to the switching rate in CMOS and the static power is related to the fabrication technology. This will be discussed in Chapter 3.

Although the previous models are based on old technologies and architectures, such as ARM7TDMI, they give clues about which factors can affect the power/energy of a microprocessor and they are listed below [23], [24], [25] [9], [7], [26], [27], [28]:

1. The effect of the instruction base cost.
   The base power/ energy cost of an instruction is the minimum average power/energy cost to finish that instruction.

2. The effect of adjacent instructions.
   Adjacent instructions affect the power/energy because they cause the state of a processor to change and this part is called the instruction overhead power/energy.

3. The effect of cache misses.
   Cache misses affect the power/energy of a program and when a cache read misses, it takes more time to load data from memory and the pipeline will stall.

4. The effect of resource constraints.
   Resource constraints can affect the speed of the processor by pipeline stalls, thus they will affect the runtime of a program.

5. The effect of the operand values.
   Operand values are taken into account because the operands in two consecutive instructions will influence the switching rate of CMOS and more CMOS switches will consume more power/energy.

6. The effect of the various addressing modes.
   Different addressing modes describe where the data comes from and the power/energy of the instruction may vary, depending on the different addressing modes.

However, some of these factors may not apply to modern processors. In this chapter, considering these factors, different models are analyzed including the advantages and disadvantages. Moreover, we do not only focus on the instruction-level power/energy model but also analyse some other well known methods and models.

2.1 Basic Model

2.1.1 The Base Power/Energy Cost

Different types of instructions have different execution times and use different parts of the processor, therefore the energy consumed by different instructions may be different. For example, instructions NOP, ADD and MUL use different units of a datapath and the energy consumption should not be the same. The base power/energy cost is used to describe the fundamental cost of an instruction and it can be thought of as the cost related to the basic processing to run the instruction [7]. For example, assuming the base energy cost of an ADD is 1 nJ, it means the energy cost of the processor to execute an ADD, from start (fetch) to finish (Register update), is 1 nJ. This idea works for both pipeline and non-pipeline processors.

The base power cost of an instruction, i, can be measured by running a loop, which only contains the instruction i. Then, the average power consumption of this loop is the basic power cost of instruction i. However, the size of this loop must be neither too big nor too small. If the size is too big, it may bring cache misses and if the size is too small, it cannot fully fill the pipeline. On the other hand, some load/store and branch instructions also exist in the loop. Thus, a big loop size can reduce the effect of these instructions.

The base energy cost is the energy consumed by an instruction going through the datapath. The base energy cost of an instruction equals the base power cost times the number of non-overlapping cycles to execute that instruction. In other words, the base energy cost of an instruction = the base power cost $\times$ the cycle per instruction (CPI) $\times \frac{1}{clock\ frequency}$. For example, if the base power cost of instruction i is $5\mu W$ and 1000 instructions need 1000 clock cycles to finish, the CPI will be 1, and the base energy of i will be $5\mu W \times \frac{1}{clock\ frequency}$. On the other hand, if it takes 2000 clock cycles to finish, the CPI will be 2 and the base energy of i will be $5\mu W \times 2 \times \frac{1}{clock\ frequency}$, respectively.

However, the idea of base power and energy cannot be used in a superscalar processor and this will be discussed in Chapter 5.

2.1.2 The effect of two adjacent instructions

The power (energy) cost of a pair of instructions is bigger than the average (sum) of the base power (energy) cost of each single instruction. This extra cost is called overhead power (energy) or circuit state overhead [8]. Tiwari *et al.* introduced an example of overhead current (because Power= Current $\times$ Voltage and Voltage is a constant in this example.) [7]. The base current costs of XOR and ADD are $319.2mA$ and $313.6mA$, respectively. Therefore, the expected base cost current of this pair should be their average current, which is $316.4mA$. However, the actual cost is $323.2mA$ which is $6.8mA$ more and this extra current is the overhead current.

The reason for the overhead power (energy) is that the base cost is determined when the same instructions are executed again and again, which means the context of each instruction changes the least. However, when a pair of different instructions is investigated, the context changes more than before. Therefore, the power/energy overhead is non-zero [7].

A matrix can record the overhead power/energy cost of different pairs of instructions. Table 2.2 is an example of an overhead matrix. The first column gives the first instruction of each pair and the first row gives the second instruction of the pair. Therefore the table can store the overhead energy or the average energy of the different pairs.

Table 2.2: The energy matrix (nJ) [29].

	LDI	LAB	MOV1	MOV2	ASL	MAC
LDI	3.6	13.7	15.5	6.3	10.8	6.0
LAB		2.5	1.9	12.2	20.9	15.0
MOV1			4.0	18.3	10.5	3.8
MOV2				25.6	26.7	22.2
ASL					3.6	8.0
MAC						12.5

The overhead power/energy was first proposed by Tiwari *et al.* [8] and they assumed that the sequence of the instruction pairs will not affect the overhead power/energy. For example, the overhead between ADD and MOV is the same as MOV and ADD. Therefore, Table 2.2 is a triangular matrix.

The method to measure overhead power/energy is similar to measuring the base power/energy cost and is to execute the same pair of instructions in a loop. Overhead power is the difference between the average power of the loop and the average power of the base power cost of each instruction. Assuming there are 100 pairs of instructions in the loop, the overhead energy is presented by the following equation:

$$\begin{aligned} E_{overhead} &= \frac{E_{total} - N \times E_{base\_Int1} - N \times E_{base\_Int2}}{2N} \\ E_{overhead} &= \frac{E_{total} - 100 \times E_{base\_Int1} - 100 \times E_{base\_Int2}}{200}, \end{aligned} \tag{2.1}$$

where E_{total} is the total energy consumed by this loop, E_{base_Int1} and E_{base_Int2} are the base energy cost of the first and second instructions, respectively. There are 200 instructions in the program and the state of the circuit changes 200 times in total.

Sometimes, an overhead cost may appear although these two instructions are not adjacent. Tiwari *et al.* presented the example in Table 2.3 [7, 29].

Table 2.3: An example of a sequence of four instruction where the overhead cost between 1 and 3 cannot be ignored (mA) [29].

number	instruction	base	overhead	cycles
1	MUL:LAB(X0+1),(X1+1)	37.2	(1&2)18.4	1
2	NOP	14.4	(2&3)18.4	1
3	MUL:LAB(X0+1),(X1+1)	36.6	(3&4)18.4	1
4	NOP	14.4	(4&1)18.4	1
Total	102.6(37.2+18.4+36.6+18.4)+73.6(18.4 × 4)=176.2			

In this test, the target processor is a Fujitsu $3.3V$, $0.5\mu m$, 40 MHz CMOS processor. The estimated energy is $14.5365nJ$ ($176.2 \times \frac{1}{40MHz} \times 3.3V$) but in reality it is $16.83nJ$ ($204.0mA \times \frac{1}{40MHz} \times 3.3V$). The difference in energy consumption is $2.2935nJ$ ($27.8mA \times \frac{1}{40MHz} \times 3.3V$, and $27.8mA$ =204.0 mA-176.2mA) and it comes from the circuit

state overhead between nonadjacent instructions 1 and 3. The reason is the state of the multiplier only can be updated when it is used. Thus, the first instruction and the third one have an overhead energy, which is considered to be $1.14675nJ$ ($13.9\ mA \times \frac{1}{40MHz} \times 3.3V$, and $13.9\ mA = 27.8\ mA \div 2$). Considering the overhead of instruction 1 and 3 can lead to a better estimate.

However, it seems that the effect of circuit states varies in its impact. Sometimes, the overhead does not affect the total power very much but sometimes it does. For example, in the case of the 486DX2 ($40MHz$, $3.3V$) and the Fujitsu SPARClite MB86934 ($20MHz$, $3.3V$), the circuit state overhead varied in a small range and had a limited impact. For the 486DX2, the overhead current varied in the range 0-$30mA$ while most programs varied in the range of 300-$420mA$. In the case of the '934, the range of the overhead current and most programs are 0-$20mA$ and 250-$400mA$, respectively. On the other hand, in the case of the Fujitsu proprietary DSP ($40MHz$, $3.3V$), the average current for most programs was in the range 20-$60mA$ but the overhead for some pairs was significant, which is up to $26.7mA$ and the overhead matrix table is presented in Table 2.2 [29]. Tiwari *et al.* said the reason for this difference may be that the architecture of the 486DX2 is more complex than for a simple DSP. Thus, the major energy cost of the circuits in complex processors is common to all instructions, such as pre-fetch, pipeline control, clocks. The overhead of the circuit is swamped by these common costs [8].

Moreover, because of the different overhead energy, Tiwari *et al.* stated that reordering the instruction sequence to get a low overhead energy cost is a method to write a low energy program although it does not save the number of clock cycles [7].

2.1.3 The Basic Instruction Level Model

Based on the previous analysis, the first instruction level energy model was created by Tiwari *et al.* [7, 8, 29]. Although it was created 20 years ago, it is important because a lot of modern power/energy models are based on it and we name it the basic model. The model is described as

$$E = \sum_i (B_i \times N_i) + \sum_{i,j} (O_{i,j} \times N_{i,j}) + \sum_k E_k, \tag{2.2}$$

where E is the total energy consumed by the program and consists of three parts. The first part is the sum of the base energy costs of each instruction. B_i is the base cost of instruction i and N_i is the number of instruction i executed in the program. The second component is the overhead energy cost due to instruction switching. $O_{i,j}$ is the overhead cost due to the instruction sequence (i,j) and $N_{i,j}$ is the number of occurrences of the sequence (i,j). The last part E_k is any additional energy due to cache misses or resource constraints.

For example, assuming a program has the following instruction sequence: ADD, MOV, and SUB, the energy consumed by this program is

$E_{total} = E_{ADD} + E_{MOV} + E_{SUB} + E_{ADD,MOV} + E_{MOV,SUB} + \sum_k E_k$

However, if the instructions sequence is ADD, SUB, and MOV, the energy will be

$E_{total} = E_{ADD} + E_{MOV} + E_{SUB} + E_{ADD,SUB} + E_{SUB,MOV} + \sum_k E_k$

Thus, the model is highly related to the instruction sequence.

The main advantage of this model was that it showed how to estimate the energy of a program for the first time. However, the model has two disadvantages; the first is the computation cost to measure every possible overhead is too great. Generally, the measurement times are proportional to the square of the number of instruction types. For example a model based on a DSP 56K needs 1176 measurements in total [30], because there are 49 instructions in the instruction set architecture (ISA).

The second disadvantage is the estimation of the cache miss energy or resource constraints energy, E_k, because this model does not define any method to compute the relationship between cache miss rate or pipeline stall and energy.

2.1.4 The extension of the basic model

Bona *et al.* used a similar idea and extended the basic model to study a very long instruction word (VLIW) processor: a Lx 4-issue VLIW pipelined processor provided by STMicroelectronics [31–33], the results are based on gate-level simulations. Assume a program W has N very long instructions $W = \{w_1 \cdots, w_{n-1}, w_n, \cdots, w_N\}$, each instruction w_n has K parallel operations $w_n = [w_n^1 \cdots w_n^k \cdots w_n^K]$, where w_n^k is the $k-th$ operation (issued on the $k-th$ lane of the processor) of the $n-th$ bundle of the stream, and the processor has S pipeline stages. The energy consumed by this program can be calculated as:

$$E(W) \approx \sum_{1 \leqslant n \leqslant N} \sum_{\forall s \epsilon S} [U_s(0|0) + \sum_{\forall k \epsilon K} v_s(w_n^k | w_{n-1}^k) + m_s^n * p_s^n * S_s + l_s^n * q_s^n * M_s], \quad (2.3)$$

where $U_s(0|0)$ is the base energy cost of stage s during execution of a bundle constituted entirely of NOPs ($[NOP \cdots NOP]^T$). $v_s(w_n^k | w_{n-1}^k)$ is the energy cost due to the change of operation ($w_{n-1}^k \rightarrow w_n^k$) on the same lane k. $m(l)$ is the average number of additional cycles due to data (instruction) cache miss during the execution of the w_n in stage s. $p(q)$ is the average probability that a data(instruction) cache miss can affect one instruction. $S(M)$ is the average energy consumption of the whole processor during cache misses.

Equation 2.3 can fit with the basic model:

$$B_i = \sum_{s \epsilon S} [U_S(0|0) + v_s(w_n^i|w_n^i)] \tag{2.4}$$

$$O_{i,j} = \sum_{s \epsilon S} [v_s(w_n^i|w_n^j) - v_s(w_n^i|w_n^i)] \tag{2.5}$$

$$\begin{aligned} \sum_k E_k &= \sum_{1 \leqslant n \leqslant N, s \epsilon S} [\mu_s + \sigma_s], \\ \mu_s^n &= m_s^n * p_s^n * S_s, \\ \sigma_s^n &= l_s^n * q_s^n * M_s, \end{aligned} \tag{2.6}$$

where μ_s and σ_s are the data cache miss penalty and instruction cache miss penalty respectively.

This model shows good performance and the average error is 1.9% with several different benchmarks tests including Mediabench applications (namely: G721 encoder and decoder, EPIC encoder and de-coder, MPEG2), matrix elaboration algorithms and a set of finite impulse response filters [31]. The disadvantage is that it has to look up the energy of each operation change ($v_s(w_n^i|w_n^j)$) from this overhead table and sum them together. Thus it takes a big effort for a program which has millions of instructions. However, the basic model also has this disadvantage.

Ascian *et al.* created a power model for a ST20-C2P processor (two stage pipeline, synthesized in HCMOS7 technology, $0.25\mu m$, $2.5V$) [34]. The power is measured by Powermill simulation, and a VHDL simulation can provide a cycle accurate trace of the instruction flow.

Assume an instruction fragment is as follows:

...

1. I_{i-1}^k
2. I_i^k
3. I_i^k
4. I_{i+1}^k

...

The sequence is repeated N times ($k \in 1, 2, ..., N$) and $c_i^{(k)}$ is the power used by the i-th instruction of the sequence at the k-th repetition. Thus, the backward and forward

costs are defined as

$$(I_{i-1} \rightarrow I_i)_{back} = \sum_{k=1}^{N} \frac{c_2^{(k)} - base(I_i)}{N} \tag{2.7}$$

$$(I_i \rightarrow I_{i+1})_{forw} = \sum_{k=1}^{N} \frac{c_3^{(k)} - base(I_i)}{N} \tag{2.8}$$

The model is

$$Cost(I_i) = base(I_i) + (I_{i-1} \rightarrow I_i)_{back} + (I_i \rightarrow I_{i+1})_{forw}, \tag{2.9}$$

where $Cost(I_i)$ is the base power cost of instruction i, $(I_{i-1} \rightarrow I_i)_{back}$ and $(I_i \rightarrow I_{i+1})_{forw}$ are the backward and forward costs respectively.

On the other hand, Equation 2.9 can fit with the basic model:

$$B_i = base(I_i) \tag{2.10}$$

$$O_{i,j} = (I_{i-1} \rightarrow I_i)_{back} + (I_i \rightarrow I_{i+1})_{forw}. \tag{2.11}$$

However, this model does not work well for the ST20-C2P processor. For example, for the test, which is a sequence of about 500 random instructions from the ISA, there are about 140 instructions with an estimated error between -40% to -20%. The reason is the model does not consider the effect of the data but this affects the power of the processor very much. The estimated base power cost of each instruction uses zero as the operand, thus the energy is underestimated. In order to consider the effect of data, an improved model (a data dependency model) was created and it will be discussed in Section 2.4.3.

On top of this, this model makes the size problem of the overhead cost even worse. For the basic model, the overhead cost does not distinguish the sequence of a pair of instructions. For example, the overhead of the pair a and b is the same as the pair b and a. However, this model considers the overhead cost in two different cases: $(I_{i-1} \rightarrow I_i)_{back}$ and $(I_{i-1} \rightarrow I_i)_{forw}$. This makes the model harder to use and it needs a cycle trace simulation.

2.2 Nop Model

In order to solve the size problem of the overhead power/energy table, Klass *et al.* used another method to measure the base power/energy cost of the processor CMU 56000 DSP (a non pipeline, 24-bit, fixed-point DSP, produced by Motorola Semiconductor [35]) [30]. They calculated the base energy cost of each instruction by using loops which alternate the target instruction with NOP instructions. Thus, the base energy cost in the NOP

model can be expressed as

$$B_{NOP\_i} = B_i + O_i, \tag{2.12}$$

where B_i is the base cost of the target instruction i (the same definition as in the basic model) and O_i is the overhead cost between the target instruction and NOP.

Then, the energy consumption of a program is presented as

$$E = \sum_i B_{NOP\_i}, \tag{2.13}$$

where B_{NOP_i} is the base energy of each instruction i defined in the NOP model. The authors assumed that the overhead energy caused by the circuit state switching was considered in the base energy cost. Therefore, the measurement times will decrease from $O(N^2)$ to $O(N)$, where N is the number of instructions in the ISA. The error of this method is between 1% and 8% with four tests: fir4, fir64, fir4u and fft.

Nikolaidis *et al.* used this method to study the ARM7TDMI (a three-stage pipeline, 32-bit RISC CPU designed by ARM [36]). On top of this, they presented a better definition of the base energy cost in the NOP model – the amount of energy which is consumed for the execution of an instruction after the execution of a reference instruction (NOP) [9]. Moreover they analysed the base cost of the NOP model in detail and presented the relationship of the base cost between the basic model and NOP model.

Firstly, assuming a program which has n instructions and needs n clock cycles to finish, the energy consumption of the program is the sum of the energy costs of each clock cycle, and can be presented as:

$$E\_M(Instr) = E_{cycle\_1} + E_{cycle\_2} + \cdots + E_{cycle\_n} \tag{2.14}$$

If that program contains only one test instruction and $n-1$ reference instructions (NOP), then the energy consumed by the test instruction is

$$E(Instr) = E\_M(Instr) - (n-1)E_{NOP} \tag{2.15}$$

In order to calculate the base cost, a loop which contains one test instruction and several NOP instructions is executed. Assuming the pipeline has three stages and the test instruction is executed in three cycles, Table 2.4 shows each pipeline stage.

Based on Equation 2.14, the consumed energy will be

$$E\_M(Instr) = E_{cycle\_n+1} + E_{cycle\_n+2} + E_{cycle\_n+3} \tag{2.16}$$

Table 2.4: Pipeline stages during the execution of instruction [9].

pipeline stages	3-stage pipeline operation				
IF	NOP	NOP	Instr	NOP	NOP
ID	NOP	NOP	NOP	Instr	NOP
EX	NOP	NOP	NOP	NOP	Instr
clock cycles	n-1	n	n+1	n+2	n+3

On the other hand, if we consider the circuit state effects due to the change of the pipeline stages (overhead cost), the energy can be described as:

$$E\_M(Instr) = E_{Instr} + 2E_{NOP} + E_{NOP,Instr} + E_{Instr,NOP} + E_{NOP,NOP}, \tag{2.17}$$

where E_{Instr} is the real energy of the instruction and $E_{NOP,Instr}$, $E_{Instr,NOP}$, $E_{NOP,NOP}$ are the inter cycle costs $(n,n+1)$, $(n+1,n+2)$, and $(n+2,n+3)$ respectively. Here, E_{Instr} is the base energy cost defined in Section 2.1. Moreover, based on Equation 2.15, the energy can be expressed as

$$E\_M(Instr) = E(Instr) + 2E_{NOP} \tag{2.18}$$

Combining Equation 2.17 and 2.18 leads to the following equation:

$$E(Instr) = E_{Instr} + E_{NOP,Instr} + E_{Instr,NOP} + E_{NOP,NOP}. \tag{2.19}$$

For example, the base energy cost of ADD in NOP model can be presented as: $E(ADD) = E_{ADD}+E_{NOP,ADD}+E_{ADD,NOP}+E_{NOP,NOP}$ (E_{ADD} and $E_{NOP,ADD}$ are the base energy cost and overhead energy defined in the basic model in Section 2.1 respectively).

Similarly, this method can be extended to study the inter-instruction effect of a pair of instructions. The state of the pipeline is shown in Table 2.5.

Table 2.5: Pipeline states during the execution of instructions for measuring inter-instruction costs [9].

pipeline stages	3-stage pipeline operation				
IF	NOP	Instr1	Instr2	NOP	NOP
ID	NOP	NOP	Instr1	Instr2	NOP
EX	NOP	NOP	NOP	Instr1	Instr2
clock cycles	n	n+1	n+2	n+3	n+4

Thus, the energy consumption from clock cycle $n+1$ to $n+4$ is

$$\begin{aligned} E\_M(Instr1, Instr2) =& E_{Instr1} + E_{Instr2} + 2E_{NOP} + E_{NOP,Instr1} + E_{Instr1,Instr2} + \\ &+ E_{Instr2,NOP} + E_{NOP,NOP} \end{aligned} \tag{2.20}$$

The inter-instruction effect can be described as

$$E(Instr1, Instr2) = E\_M(Instr1, Instr2) - E(Instr1) - E(Instr2) - 2E_{NOP} \quad (2.21)$$

Combining Equation 2.19, Equation 2.20, and Equation 2.21 yields the following equation

$$E(Instr1, Instr2) = E_{Instr1,Instr2} - E_{Instr1,NOP} - E_{NOP,Instr2} - E_{NOP,NOP}, \quad (2.22)$$

where $E(Instr1, Instr2)$ is the inter-cycle cost in NOP model ($E_{Instr1,Instr2}$ is the overhead cost defined in Section 2.1).

For example, the overhead energy cost of ADD and MOV in the NOP model can be presented as: $E(ADD, MOV) = E_{ADD,MOV} - E_{ADD,NOP} - E_{NOP,MOV} - E_{NOP,NOP}$.

Based on Equation 2.19 and Equation 2.22, it is observed that when these two equations are used to estimate the energy of a program, the inter-cycle costs (such as $E_{NOP,Instr}$ and $E_{Instr,NOP}$) will cancel each other. For example, assuming a program has three instructions ADD, MOV, and CMP, the base energy cost of ADD, MOV, and CMP and the overhead cost of $E(ADD, MOV)$ and $E(MOV, CMP)$ are shown in Figure 2.1.

$$\begin{aligned}
&E(ADD) = E_{ADD} + E_{NOP,ADD} + E_{ADD,NOP} + E_{NOP,NOP} \\
&E(MOV) = E_{MOV} + E_{NOP,MOV} + E_{MOV,NOP} + E_{NOP,NOP} \\
&E(CMP) = E_{CMP} + E_{NOP,CMP} + E_{CMP,NOP} + E_{NOP,NOP} \\
&E(ADD,MOV) = E_{ADD,MOV} - E_{ADD,NOP} - E_{NOP,MOV} - E_{NOP,NOP} \\
+\quad &E(MOV,CMP) = E_{MOV,CMP} - E_{MOV,NOP} - E_{NOP,CMP} - E_{NOP,NOP} \\
\hline
&E(\mathrm{Program}) = E_{ADD} + E_{MOV} + E_{CMP} + E_{ADD,MOV} + E_{MOV,CMP} \\
&\qquad + E_{NOP,ADD} + E_{CMP,NOP} + E_{NOP,NOP}
\end{aligned}$$

Figure 2.1: Estimation of the energy of a program consisting by three instructions [9].

Figure 2.1 shows that the energy consumed by a program is related to E_{Instr} and $E_{Instr1,Instr2}$ and they are the base energy cost and the overhead energy cost defined in Section 2.1, respectively. Thus, Figure 2.1 shows the relation between the NOP model and the basic model, and the following equation can be derived:

$$E = \sum_i (B_i \times N_i) + \sum_{i,j} (E_{i,j} \times N_{i,j}) + E_{NOP,NOP}, \quad (2.23)$$

where E_i and $E_{Instr1,Instr2}$ are the base energy cost and overhead energy cost defined in Section 2.1, respectively. Thus, this result is the same as Equation 2.2 in the basic model.

Nikolaidis *et al.* used this method to create a data dependency energy model for the ARM7TDMI and the detail about this model will be explained in Section 2.4.3. Table 2.6 is a summary of relationship between the basic model and the NOP model.

Table 2.6: The summary of the relationship between the basic model and the NOP model. The number of NOP varies depending on the processor and it has to be more than the number of pipeline stages.

Name	The basic model [7]	The NOP model [9]
Base energy cost measurement method	Int_1,Int_1,Int_1...	NOP,NOP,Int_1, NOP, NOP ...
Overhead energy cost measurement method	Int_1,Int_2,Int_1,Int2...	NOP,NOP,Int_1,Int_2, NOP, NOP ...
The value of the base energy cost	B_i	$B_i + E_{NOP,i} + E_{i,NOP} + E_{NOP,NOP}$
The value of the overhead energy cost	$E_{i,j}$	$E_{i,j} - E_{i,NOP} - E_{NOP,j} - E_{NOP,NOP}$
Energy of a program	$E = \sum_i (E_i \times N_i) + \sum_{i,j} (E_{i,j} \times N_{i,j})$	$E = \sum_i (B_i \times N_i) + \sum_{i,j} (B_{i,j} \times N_{i,j}) + E_{NOP,NOP}$

Penolazzi *et al.* extended this method to the SPARC LEON3 processor (a 32-bit CPU microprocessor core, based on the SPARC-V8 RISC architecture and instruction set, with a 7-stage pipeline) [11]. The method they used is to run $100 \times (5 \cdot NOP, IUT, 5 \cdot NOP)$ and $100 \times (5 \cdot NOP, IUT_1, IUT_2, 5 \cdot NOP)$ for an individual instruction energy test and for a pair of instructions, respectively. Thus the energy consumed by each instruction and by a pair of instructions will be

$$\begin{aligned} E_{IUT} &= \frac{E_{total} - 1000 \cdot E_{NOP}}{100} \\ E_{IUT_1,IUT_2} &= \frac{E_{total} - 1000 \cdot E_{NOP}}{100}, \end{aligned} \tag{2.24}$$

where E_{total} is the energy consumed by the program. Table 2.7 shows the results for several pairs of instructions.

Moreover, they found the overhead energy was negative in 90% of cases, which means that $E_{IUT_1} + E_{IUT_2} > E_{IUT_1,IUT_2}$. However, they did not give a explanation for this. We think the reason can be explained by Equation 2.22, and the difference between $E_{IUT_1} + E_{IUT_2}$ and E_{IUT_1,IUT_2} is $E_{Instr1,Instr2} - E_{Instr1,NOP} - E_{NOP,Instr2} - E_{NOP,NOP}$, which is likely to be negative.

In the NOP model, the base power/energy cost has a new definition [9]. However, the definition in the basic model is more widely used. Therefore, after this section, both the base cost and overhead cost assume the definition in Section 2.1 unless otherwise

Table 2.7: Energy models: 1-instr. Vs. 2-instr.Source: [11].

1-instr-model			2-instr-model		
IUT	E[pJ]	Sum	IUT_1-2	E[pJ]	Diff.[%]
add	92.75	182.39	add_and	120.65	-33.85
and	89.64				
and	89.64	188.93	and_or	127.72	-32.4
or	99.29				
ld	75.17	168.3	ld_xor	128.24	-23.81
xor	93.13				

stated. On the other hand, based on the result of Figure 2.1, these two methods are very similar.

2.3 Clustering Instructions Model

To reduce the complexity of the overhead energy model, another method called clustering is introduced [7]. The idea of the method is to cluster instructions into several different groups. Instead of considering the overhead of every possible instruction pair individually, the overhead cost comes from the different pairs of clusters. This idea can be presented by the following equation:

$$O_{i,j} = \begin{cases} E_i & \text{if } i,j \in C_i \\ E_{i,j} & \text{if } i \in C_i, j \in C_j \end{cases} \tag{2.25}$$

where $O_{i,j}$ is the overhead cost, C_i is the i-th class. Thus, the number of overhead measurements will decrease from O(ISA^2) to O(C^2).

Lee *et al*. studied a Fujitsu processor ($3.3V$, $0.5\mu m$, 40 MHz CMOS, two stage pipeline) and clustered instructions into six classes as shown in Table 2.8 [29].

Table 2.8: Six instruction classes [29].

class	addressing
LDI	immed → reg (load immediate data to a register)
LAB	mem1 → reg A and mem2 → reg B (transfer memory data to registerss A,B)
MOV1	reg1 → reg2 (move data from one register to another)
MOV2	mem→ reg, or reg→ mem (move data from memory to a register, or from a register to memory)
ASL	reg specified implicitly (add/sub, shift, logi operation sin ALU)
MAC	reg specified implicitly (multiplyy and accumulated in ALU)
String	transfer,compare,search

It is clear that the clustering is based on the different addressing modes. Moreover, instructions in the same class use similar parts of the CPU and consume similar power. Table 2.9 and Table 2.10 show the base cost of each class and the average overhead cost, respectively.

Table 2.9: The average base cost for unpacked instruction (nJ) [29].

	LDI	LAB	MOV1	MOV2	ASL	MAC
range	15.8-22.9	34.6-38.5	18.8-20.7	17.6-19.2	15.8-17.2	17-17.4
average base	19.4	36.5	19.88	18.4	16.5	17.2

Table 2.10: The average overhead cost for unpacked instruction (nJ) [29].

	LDI	LAB	MOV1	MOV2	ASL	MAC
LDI	3.6	13.7	15.5	6.3	10.8	6.0
LAB		2.5	1.9	12.2	20.9	15.0
MOV1			4.0	18.3	10.5	3.8
MOV2				25.6	26.7	22.2
ASL					3.6	8.0
MAC						12.5

Bona *et al.* extended the VLIW basic model in Section 2.1.3 and clustered the instructions into 11 classes based on the energy of each operation [33, 37]. Moreover, they presented a method (k-mean clustering algorithm) to decide how many classes should be created. The idea of this method is to minimize the following equation:

$$\sum_{j=1}^{C}\sum_{i=1}^{n_j}(x_{i,j}-c_j)^2, \tag{2.26}$$

where the instructions are clustered into j different classes, n_j is the number for elements in the j-th class, $x_{i,j}$ is the i-th element of class j and c_j is the center of gravity of the j-th cluster. When the value of Equation 2.26 is a minimum, the maximum accuracy is achieved.

Rong *et al.* studied the ADSP-2189 processor (13.3 ns instruction cycle time, 2.5 V, single-cycle instruction execution [38]) and the instructions are split into five families: ALU, MAC, Shift, MOVE and other instructions [39]. Clustering in this way is appropriate because the processor contains three independent computation units: the ALU, the multiplier– accumulator (MAC), and the Barrel shifter. The first three classes focus on the different arithmetic and logic calculations and the class MOVE focuses on data transfer, such as load/store.

Although the NOP model and the Clustering method both reduce the complexity of the overhead energy matrix, they still do not consider cache misses and pipeline stalls. Moreover, they all have to identify and count the number of instructions of each class. On top of this, it takes a lot of effort to record this information and look up the overhead table when running a big program that includes billions of instructions.

2.4 Linear Regression Method to Analyze the Power/Energy

This section will introduce the linear regression method to analyse the power and energy. This method can be used by different types of instruction-level power/energy analysis and also at other levels, such as functional-level power analysis. In this section, we focus on the models which widely use the linear regression method.

2.4.1 Introduction to linear regression

The power/energy consumed by the processor is dominated not by one variable but by many. Thus a power model should consider all of these variables and choose those that are important while ignoring those less important. Linear regression can be used to address this problem. Specifically, linear regression analysis estimates the relationship between the response and a set of variables [40].

Firstly, some input values are chosen as the potential variables, such as cache miss rate, operand Hamming distance. Then, training tests are designed that have different behaviour and the potential variables, such as different instruction distributions, pipeline stall rates, and cache miss rates.

Assuming, in test i, the value of each potential variable is $X_{i,1}, X_{i,2}, ..., X_{i,k}$, and the power/energt consumption is Y_i, the linear regression model can be presented as

$$Y_i = \alpha + \beta_1 X_{i,1} + \beta_2 X_{i,2} + ... + \beta_k X_{i,k} + \epsilon_i, \tag{2.27}$$

where k is the number of variables which can affect the power consumption and ϵ_i is the error [41]. The idea is to use least squares to find β which can minimize the following expression:

$$\sum_{i=1}^{i=n} (Y_i - \alpha - \beta_1 X_{i,1} - ... - \beta_k X_{i,k}), \tag{2.28}$$

where n is the number of the tests.

Let $\mathbf{Y}=\begin{pmatrix} Y_1 \\ Y_2 \\ ... \\ Y_n \end{pmatrix}$, $\mathbf{X}=\begin{pmatrix} 1 & X_{11} & X_{12} & ... & X_{1k} \\ 1 & X_{21} & X_{22} & ... & X_{2k} \\ ... & ... & ... & ... & ... \\ 1 & X_{n1} & X_{n2} & ... & X_{np} \end{pmatrix}$, $\beta=\begin{pmatrix} \beta_1 \\ \beta_2 \\ ... \\ \beta_n \end{pmatrix}$, $\epsilon=\begin{pmatrix} \epsilon_1 \epsilon_2 ... \epsilon_n \end{pmatrix}$.

Equation 2.28 can be rewritten as

$$\mathbf{Y} = \mathbf{X}\beta + \epsilon \tag{2.29}$$

The goal of the linear regression can be represented to minimize the following expression [41]:

$$\widehat{\beta} = (\mathbf{X}'\mathbf{X}^{-1})\mathbf{X}'\mathbf{Y}. \tag{2.30}$$

There are several mathematical algorithms to solve this, such as normal equations [41]. Finally, when the β is determined, the power/energy model with different parameters is created. Once the input values are given, the corresponding power can be estimated.

2.4.2 Linear Regression Model

The parameters which are considered vary for different processors. For example, Bircher *et al.* generated the following linear regression model for the Pentium 4 processor [42]:

$$Power = \alpha_0 + \alpha_1 \cdot metric_1 + ... + \alpha_n \cdot metric_n, \tag{2.31}$$

where α_0 and α_n are a constant number and the coefficients of the corresponding metrics, respectively. $metric_n$ are the variables they consider and are shown in Table 2.11 and Table 2.12, However, the abbreviations are not clearly explained by the authors [42]. The correlations were created based on 23 benchmarks from the SPEC2000 benchmarks. They show whether the effects of these factors are positive or negative and how important they are. For example, Instr Total/Cycle has a big positive effect (0.84), which means if a program has a bigger Instr Total/Cycle, the average power will be bigger. However, L2 Miss/Cycle has a negative effect (-0.333), which means a greater L2 Miss/Cycle will decrease the average power consumption. Based on these factors, seven different models are created.

Table 2.11: High Correlation($|correlation|$ >0.32) [42].

metric	correlation
Spec Del/Cycle	0.898
Fetched Uop/Cycle	0.84
Instr Total/Cycle	0.84
Completed Uop/Cycle	0.83
Load /Cycle	0.8
Uop /Cycle	0.79
Branch /Cycle	0.78
Stores /Cycle	0.64
Mispred Branch /Cycle	0.41
L2 Miss/Cycle	-0.33
Cancelled /Cycle	0.33

Table 2.12: Low correlation($|correlation|$ <0.32) [42].

metric	correlation
L2 Hit/Cycle	0.31
Buss Access/Cycle	-0.31
TC Del /Cycle	0.32
Bus Util	-0.31
Fp Op/Uop	-0.22
Prefetch Rate	0.17
TC Build//Cycle	-0.15
ITLB Hit/Cycle	-0.09
TC Miss/Cycle	-0.09
ITLB Miss/Cycle	-0.04
L2 Hits/Cycle	-0.03
L2 Access/Cycle	-0.02

The properties that they were concerned with are the performance and the speed of the program, which are internal properties of the program. However, there are two disadvantages to this method. The first is that it takes too many factors into consideration, thus it takes too much effort to measure the weight of each factor with different tests. Secondly, the weights of these factors are extracted from testing SPEC2000 benchmarks. However, a lot of these benchmarks (10/22) are used for validation again and the reported error is 2.6% [42]. Thus, it has not been clearly proven that the model can work well for other tests.

Fei *et al*. studied the energy consumption of a state-of-the-art extensible processor Tensilica's Xtensa (a 32-bit RISC microprocessor core, five-stage pipeline [43]) [44]. The results are based on RTL simulation. The technology is the NEC RTL cell library CB11 for the 0.18μm technology. The processor is extensible and the variables have to cover the energy usage of both the original basic design and the custom design. The basic idea is still linear regression and the model is

$$\begin{aligned} Energy =& E_0 + E_1X_1 + E_2X_2 + \cdots + E_nX_n \\ =& E_{ins}(X_1, \cdots, X_m) + E_{struc}(X_{m+1}, \cdots, X_n), \end{aligned} \tag{2.32}$$

where E_0, E_1, $\cdots$, E_n are the energy coefficients. Variables X_1,X_2,$\cdots$,X_n are chosen from both the instruction level and structural domains. E_{ins} and E_{struc} are the energies due to the original basic processor and custom hardware extensions.

The original basic processor energy, E_{ins}, is described as

$$\begin{aligned} E_{ins} =& E_{arith} \cdot Cyc_{arith} + E_{ld} \cdot Cyc_{ld} + E_{st} \cdot Cyc_{st} + E_j \cdot Cyc_j + E_{br\_tk} \cdot Cyc_{br\_tk} \\ &+ E_{br\_utk} \cdot Cyc_{br\_utk} + E_i \cdot Num_i + E_d \cdot Cyc_d + E_{uncache} \cdot Num_{uncache} \\ &+ E_{interlock} \cdot Num_{interlock} + E_{side\_tie} \cdot Cyc_{side\_tie}, \end{aligned} \tag{2.33}$$

where the variables, E_n, are shown in Table 2.13. Cyc_n corresponds to the number of cycles taken by each factor, and Num is the number of times the corresponding factor occurs.

However, the base instruction may also use custom functional blocks. Thus, for each base instruction, the activated custom hardware modules have to be chased. The custom hardware modules are divided into ten different categories, as shown in Table 2.13. Thus, the energy of the extension part, E_{struc}, is described as

$$E_{struc} = E_1 \cdot \sum_j C_{1,j}Cyc_{1,j} + E_2 \cdot \sum_j C_{2,j}Cyc_{2,j} + \cdots + E_{10} \cdot \sum_j C_{10,j}Cyc_{10,j}, \tag{2.34}$$

where $Cyc_{i,j}$(i=1,2,...,10) are the number of cycles in which the jth functional block belonging to component category i is active. There may be more than one functional block in one category. For example, E_2 contains three blocks "+", "-", and "comp". $C_{i,j}$

is the energy complexity of this functional block. E_i(i=1,2,...,10) is the average energy computation per bit (per entry for the table) per cycle for each kind of resource module category and they are listed in Table 2.13 . TIE means Tensilica instruction extension, which is used to customize the instruction set.

Table 2.13: Energy coefficients of the characterized Xtensa processor [44].

coefficient	description	value(nJ)	coefficient	description	value(fJ)
E_{arith}	arithmetic instruction	0.638	E_1	*	152
E_{ld}	load instruction	0.512	E_2	+/-/comp	70
E_{st}	store instruction	1.092	E_3	log/red/mux	12
E_j	jump instruction	0.603	E_4	shifter	377
E_{br_tk}	branch taken	0.504	E_5	custom register	177
E_{br_uk}	branch untaken	0.345	E_6	TIE_mult	165
E_{side_tie}	side effects	0.616	E_7	TIE_mac	190
E_i	instruction cache miss	2.93	E_8	TIE_add	69
E_d	data cache miss	4.34	E_9	TIE_{csa}	37
$E_{uncache}$	uncached instruction fetch	2.42	E_{10}	table	27
$E_{interlock}$	processor interlock	0.98			

The benefits of this model are its accuracy (the maximum error of ten benchmarks is -8.5%) and detailed analysis, because it shows how the energy is consumed by different functions of the processor. For example, a data cache miss will consume $4.34nJ$ [44]. However, one big disadvantage of this model is its complexity and so it is hard to create and use.

2.4.3 Data Dependent Model

2.4.3.1 The Effect of Data

The operand values will affect the energy of a program in two different aspects. Firstly, the base power/energy cost is related to the operand values [1, 7, 25, 45]. For example, Nikolaidis *et al.* studied an ARM7TDMI (supply voltage 2.5V) and found that when the operands of ADD were (0,0) and (55555555,AAAAAAAA), the energy consumption would be $0.95nJ$ and $1.12nJ$ respectively. Furthermore, the growth of energy is proportional to the number of '1's in the operand for ARM7TDMI processors [45].

Figure 2.2 shows the test results for an ARM7TDMI, and Nikolaidis *et al.* claimed that although the measurements were not finished completely, there was a relation between the energy and the number of '1's in the values of their addresses and the operands [45]. The relation is close to linear and the energy is proportional to the number of '1's in the operand [45].

On the other hand, Tiwari *et al.* showed that, for the Intel 486DX2, more '1's consume less energy and the number of '1's was inversely proportional to energy [8]. Table 2.14

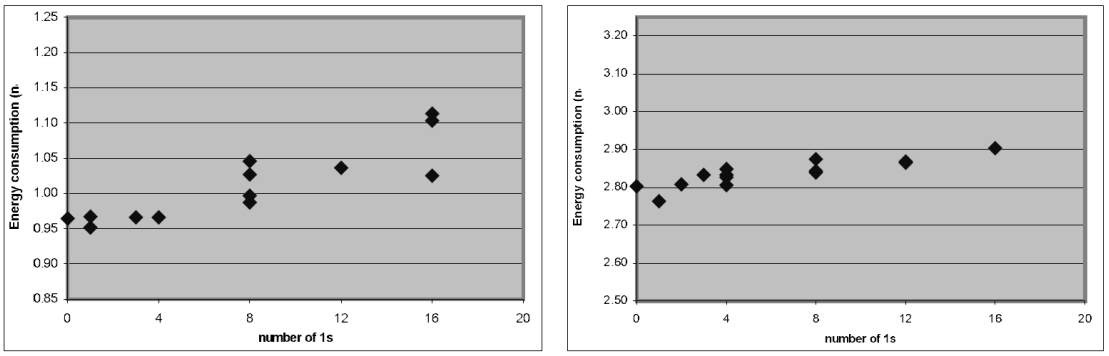


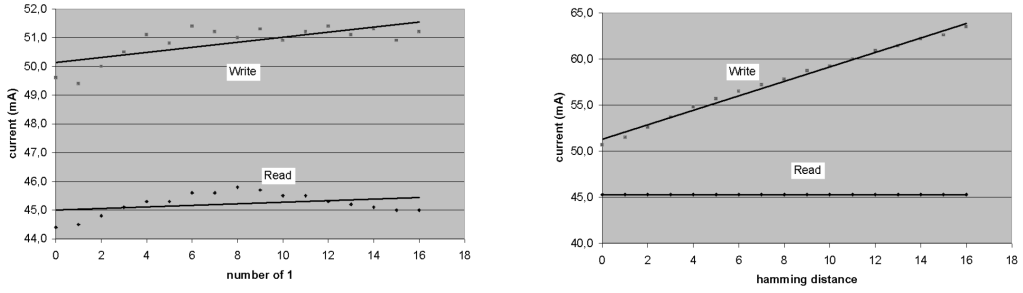
Figure 2.2: Energy consumption of (a) ADD and (b) LDR as a function of the number of '1's in the operand [45].

shows how operands may affect the base cost of MOV, BX, DATA in the Intel 486DX2 and it is clear that the more '1's exist, the less power is consumed [8]. For example, when the operand is 0x, the current is 309.5 mA. However, when the operand is 0xFFFF, the current is 288.5 mA. Compared with the results of an ARM7TDMI, in Figure 2.2, the effect of the number of '1' is opposite.

Table 2.14: Base cost of MOV BX,DATA.

data	0x0	0xF	0xFF	0xFFF	0xFFFF
No. of 1's	0	4	8	12	16
Current(mA)	309.5	305.2	300.1	294.2	288.5

Moreover, the more changes between the operands of two consecutive instructions, the more power/energy will be consumed [2, 11]. The reason is that the bigger Hamming distance makes the CMOS in the processor switch more often and consume more dynamic power.



(a) CPU current depending on number of '1's on data bus.

(b) CPU current depending on Hamming distance on data bus.

Figure 2.3: The operand can affect the current of the ARM7TDMI [16].

Steinke *et al.* tested an ARM7TDMI processor AT91M40400 and got a similar result to Figure 2.2. Moreover, they also tested how the Hamming distance may affect the power. Compared with Figure 2.3(a) and Figure 2.3(b), the effect of different Hamming distances on the data bus is stronger than the effect of number of the '1's on the data bus [16].

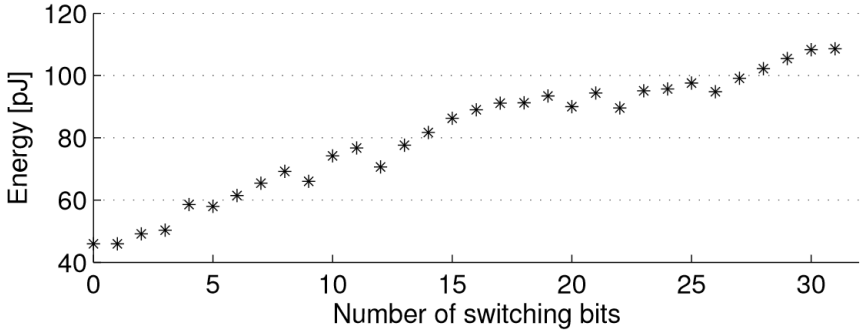


Figure 2.4: Energy change vs. data switching activity [11].

Figure 2.4 shows the more switching bits there are, the more energy the processor will consume. The target processor is a SPARC Leon3 which is synthesized to run at $400MHz$ with TSMC90nm technology [11].

2.4.3.2 Different Data Dependent Models

Because the power/energy is strongly affected by the data in some cases, a data dependent model was created. The main idea of this method is that the base power/energy cost of each instruction is not a constant but related to the data of operands. Linear regression is used to analyse how the data may affect the base power/energy cost.

One of the first data dependent models was created by Ascian *et al.* for a ST20-C2P processor (synthesized in HCMOS7 technology, $0.25\mu m$, $2.5V$) [34]. The model is presented as

$$Cost(I_i) = base^{(0)}(I_i) + (I_{i-1} \rightarrow I_i)^{(0)}_{back} + (I_i \rightarrow I_{i+1})^{(0)}_{forw} + f(data), \tag{2.35}$$

where the variables $base^{(0)}(I_i)$, $(I_{i-1} \rightarrow I_i)^{(0)}_{back}$, and $(I_i \rightarrow I_{i+1})^{(0)}_{forw}$ are the base cost, and the backward and forward costs when the activity on the registers and buses remains null between two instructions, respectively. $f(data)$ is the power which is affected by the data. The definitions of $(I_{i-1} \rightarrow I_i)_{back}$ and $(I_i \rightarrow I_{i+1})_{forw}$ were explained in Equation 2.7 and Equation 2.8.

Moreover, the $f(data)$ part can be defined as

$$\begin{aligned} f(data) = & \\ & Hamm^{0\rightarrow1}(xbus_{i-1}, xbus_i) \times weight^{0\rightarrow1}_{xbus} + \\ & Hamm^{1\rightarrow0}(xbus_{i-1}, xbus_i) \times weight^{1\rightarrow0}_{xbus} + \\ & Hamm^{0\rightarrow1}(ybus_{i-1}, ybus_i) \times weight^{0\rightarrow1}_{ybus} + \\ & Hamm^{1\rightarrow0}(ybus_{i-1}, xbus_i) \times weight^{1\rightarrow0}_{ybus} + \\ & Hamm(memaddr_{i-1}, memaddr_i) \times weight_{memaddr}, \end{aligned} \tag{2.36}$$

where the function $Hamm^{x \to y}(a, b)$ gives the number of bits switching $x \to y$ when moving from state a to b. $Hamm(memaddr_{i-1}, memaddr_i)$ shows the bit switching between the previous memory address and the current address, and does not distinguish whether it is from 0 to 1 or 1 to 0.

Six benchmarks (sum_low, sum_high, rand_inst, mtx_sum, mtx_mul, dft) are used to test the performance of the model and the maximum error is less than 6%. However, some of these benchmarks are too simple, thus the test result is weak for validation. For example, rand_inst is just a group of random instructions. It should be tested by more rigorous benchmarks. On the other hand, this model makes the problem of the size of the overhead cost worse (twice as big as the basic model). Thus, it has to trace the instructions of the program in detail and is harder to use.

Kavvadias *et al.* created a data-dependency model for an ARM7TDMI processor [5]. The base energy cost of each instruction is

$$E_i = b_i + \sum_j a_{i,j} N_{i,j}, \tag{2.37}$$

where E_i is the total base cost of instruction i, which contains two parts b_i and $\sum_j a_{i,j} N_{i,j}$. b_i is the pure base cost (the methods in the NOP model in Section 2.2) of the instruction i (the operands are zero). $a_{i,j}$ and $N_{i,j}$ are the coefficient and the number of '1's of the energy sensitive factors of the instruction i, respectively. The energy sensitive factor coefficients are listed in Table 2.15.

Table 2.15: Energy-sensitive factor coefficients [5].

Energy-sensitive factor	coefficient
register number	$a_{i,1}$
register value	$a_{i,2}$
immediate value	$a_{i,3}$
operand value	$a_{i,4}$
operand address	$a_{i,5}$
fetch address	$a_{i,6}$

From Equation 2.37, it is clear that even if two instructions have the same opcode, the energy will be different depending on their operands. Based on the factors in Table 2.15, Kavvadias *et al.* analysed each factor separately because they thought the correlation between each factor to be insignificant. Finally, the energy of a program which has n instructions can be estimated as:

$$E = \sum_1^n E_i + \sum_1^{n-1} O_{i,i+1} + \sum \epsilon, \tag{2.38}$$

where the first part is the sum of E_i of each instruction, which has already been presented in Equation 2.37. $O_{i,j}$ is the inter-instruction cost of instructions i and j (the methods in the NOP model in Section 2.2), and ϵ is the cost of a pipeline stall [5].

Although this model takes the effect of the operand value into consideration and estimates the energy very accurately based on testing a real kernel, there are several disadvantages. Firstly, the benchmarks used to validate the model are too few (only one), and it is not a common, well-known benchmark. Secondly, the model is hard to use because it has to consider the overhead energy of two instructions. Moreover, it is also difficult to get the number of '1's in each operand. Thus, it is hard to predict the energy of a program which has billions of instructions.

Sarta *et al.* created a power model for a ST20-C1 processor produced by STMicroelectronics [2]. It has a two stage pipeline and synthesized using a 0.35 μm, 3.3V library. The test results are based on VHDL simulations and Powermill to analyse the power of the CPU. Powermill is a simulation tool [46]. The model is described as

$$\begin{aligned} power_{average} &= P_{data} + P_j + P_{i,j} \\ &= K_1 \cdot n_1 + \cdots + K_n \cdot n_n + K_0 + C_{i,j}, \end{aligned} \tag{2.39}$$

where $power_{average}$ consists of three part: the power related with data, the base power cost of each instructions, and the overhead power cost of instruction pair i and j. K_i and n_i are the weights and the numbers of transitions of the elements which can influence the power consumption, respectively. Furthermore, the vector K is called the activity index and includes memory address, data write, x_bus, y_bus and a fixed cost. K_0 is the minimum cost for an instruction. $C_{i,j}$ is the changing-instruction cost between instructions i and j, like the overhead cost discussed earlier.

However, this model does not consider cache misses or pipeline stall penalties. Therefore, if the cache miss rate is high, the model may not be able to accurately predict the energy. Additionally, the $C_{i,j}$ term introduces the same problem as the overhead energy, $O_{i,j}$, whereby it is too complex to compute every possible combination. Moreover, the authors did not use enough benchmarks to prove the performance of the model.

Steinke *et al.* studied an ARM7TDMI processor, AT91M40400, and created a model which considers both the number of '1's and the Hamming distance of two operands in two adjacent instructions [16]. The model includes four parts: the instruction-dependent cost inside the CPU(E_{cpu_instr}), the data-dependent cost inside the CPU(E_{cpu_data}), the instruction-dependent costs in the instruction memory(E_{mem_instr}) and the data-dependent costs in the data memory(E_{mem_data}). Therefore, the total energy consumed by a processor is

$$E_{total} = E_{cpu\_instr} + E_{cpu\_data} + E_{mem\_instr} + E_{mem\_data} \tag{2.40}$$

If a program has m instructions and each instruction has s immediate values and t registers, the energy of the CPU is described as

$$\begin{aligned} E_{CPU\_instr} = & \sum_{i=1}^{m}(\\ & BaseCPU(Opcode_i)+ \\ & \sum_{j=1}^{s}(\alpha_1 * \omega(Imm_{i,j} + \beta_1 * h(Imm_{i-1,k}, Imm_{i,j}))+ \\ & \sum_{k=1}^{t}(\alpha_2 * \omega(Reg_{i,k} + \beta_2 * h(Reg_{i-1,k}, Reg_{i,k}))+ \\ & \sum_{k=1}^{t}(\alpha_3 * \omega(RegVal_{i,k} + \beta_3 * h(RegVal_{i-1,k}, RegVal_{i,k}))+ \\ & \alpha_4 * \omega(IAddr_i) + \beta_4 * h(IAddr_{i-1}, IAddr_i)+ \\ & FUChange(Instr_{i-1}, Instr_i)), \end{aligned} \tag{2.41}$$

where Imm, Reg, $RegVal$, and $IAddr$ are the immediate value, the register number, values within the registers, and the instruction address, respectively. ω is a function used to model the energy affected by '0's and '1's of the data, such as immediate values ($\omega(Imm_{i,j} + \beta_1 * h(Imm_{i-1,k}, Imm_{i,j}))$) and register data ($\omega(Reg_{i,k} + \beta_2 * h(Reg_{i-1,k}, Reg_{i,k}))$). Function h is used to model the changing bits of two consecutive instructions.

Moreover, the data dependent costs inside the CPU for n data accesses are related to the data address $DAddr$, the $Data$ itself and the direction dir (read/write). Therefore, the energy of this part is described as

$$\begin{aligned} E_{cpu\_data} = & \sum_{i=1}^{n}(\\ & \alpha_5 * \omega(DAddr_i) + \beta_5 * h(DAddr_{i-1}, DAddr_i)+ \\ & \alpha_{6,dir} * \omega(Data_i) + \beta_{6,dir} * h(Data_{i-1}, Data_i)) \end{aligned} \tag{2.42}$$

We concentrate on the average power and energy cost of the CPU, thus the memory energy consumption is not related to our work and we do not introduce it here. Based on Equations 2.40, 2.41, and 2.42, a model to estimate the energy consumption of a program is created [16]. The advantage of this model is that it takes memory access into consideration(E_{cpu_data}). Memory access normally happens when cache misses occur. Thus, this model considers the cache miss effect to some extent. However, the disadvantage of this model is its complexity. It has 13 variables and several internal functions.

Moreover, it has to trace the bit changes of two consecutive instructions and considers the overhead effect ($FUChange(Instr_{i-1}, Instr_i)$). Thus, it is difficult to use for a big program which has billions of instructions. On top of this, this model is not validated by any benchmarks.

Bazzaz *et al.* created a model for the AT91SAM7X256 which uses the ARM7TDMI as the core [15]. 60 specialized training tests are used to analyse the coefficients of each energy sensitive factor. The energy model is described as

$$E_{app} = \sum_{p \in P} N_p \times coeff(p), \tag{2.43}$$

where N_p is the total number of occurrences of P during execution, P is the full set of the factors which can affect the energy, and $coeff(p)$ is the energy coefficient of p. The set P includes 35 parameters and they are listed in Table 2.16 [15]. To compute the coefficients of the model, linear regression is used.

Table 2.16: Final results of regression of ARM7TDMI [15].

Instruction parameters (related to Opcode)							
Param	Energy(nJ)	Param	Energy(nJ)	Param	Energy(nJ)	Param	Energy(nJ)
ADD	0.89	MVN	1.13	ADC	1.127	ORR	1.131
RSB	1.153	AND	1.178	RSC	1.119	B	0.79
BIC	1.049	SMLAL	4.391	BL	6.023	SMULL	4.29
STM	1.449	CMP	0.978	STR	1.343	EOR	1.167
LDM	1.94	SWP	3.593	LDR	1.84	TEQS	1.003
TSTS	1.006	MLA	3.423	UMLAL	4.878	MOVS	1.021
MUL	2.931	Shift	0.288	ADDS	1.481	SBC	1.113
CMN	0.976	SUB	1.143	MOV	1.284	UMULL	4.254
Inter instruction parameters (related to data)				Memory parameters			
Parameter		Energy (pJ)		Parameter		Energy (nJ)	
Hamming distance		9.23		Flash loads		2.25	
Instruction weight		23.6		SRAM loads		0.72	
Regbank bit flip		2.81		SRAM stores		0.98	

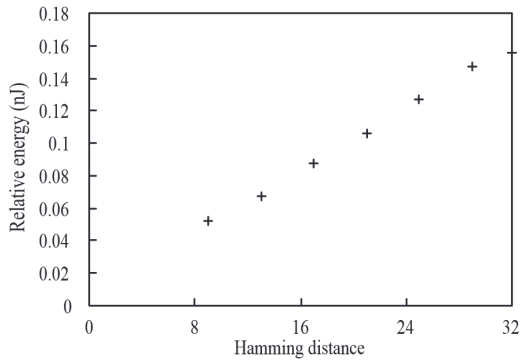
This table merges the basic model and the data dependency model. For example, the base cost of an instruction comes from the basic model but the Hamming distance and instruction weight come from the variables in the data dependency model. Seven benchmarks, which come from MiBench, are used to test the performance of the model and the results show the worst estimation error is 6% [15]. In this model, the pipeline stalls are grouped into two cases: fixed length and variable length. The fixed length pipeline stall is caused by multi-cycle instructions, such as SWAP which takes four cycles to execute. The variable length stalls are caused for several different reasons, such as operand values, previously accessed memory address and correct prediction of a jump. However, this model presents a solution to the fixed pipeline stalls but ignores

the variable length pipeline stalls. The reason is that estimating the variable length pipeline stall needs cycle-accurate simulation of the program [15].

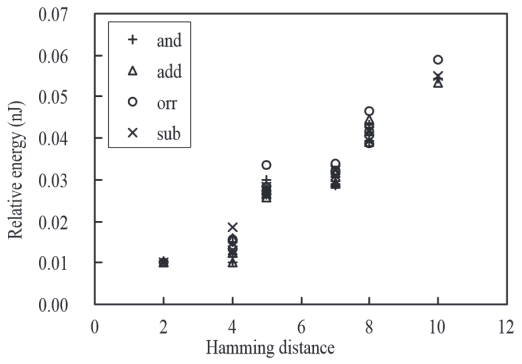
Based on the analysis of the previous data dependency models, it is clear that this kind of model can predict the power/energy accurately. However these models have a common problem, which is that the model needs a lot of input data. Furthermore, the number of bit changes of two consecutive instructions is needed, which is hard to trace when the program is big. Thus, if the model could avoid data dependency, it would be very concise and convenient to use.

2.4.4 Cycle-accurate Model

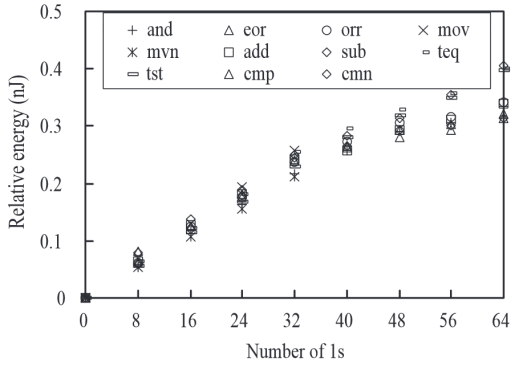
Different instructions may use different parts of the pipeline and each block of a processor consumes different power. Thus, linear regression can also be extended to study the power/energy consumption of each different pipeline stage, rather than the whole processor or the system, to generate a cycle-accurate energy model. On the other hand, in order to create a cycle-accurate model, the effect of data dependence has to be considered [13, 17].



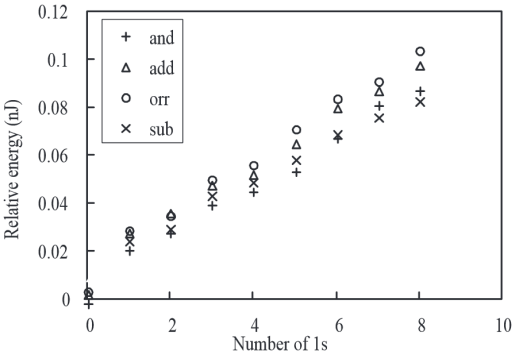
(a) Energy consumption by the instruction fetch address (PC stage).



(b) Energy consumption by the register number (EX stage).



(c) Energy consumption by the operand value (EX stage).



(d) Energy consumption by the immediate value (EX stage).

Figure 2.5: The energy consumption affected by data in different stages [13].

Chang *et al.* tested the energy consumed by each stage of a processor ARM7TDMI [13]. They showed that the energy consumption of each pipeline stage was affected by the opcode, the instruction fetch address, the register number, the register value, the data fetch address, and the immediate operand. For example, Figure 2.5 shows that the Hamming distance and number of '1's may affect the energy consumption [13]. Then, how these factors affect the different pipeline stages: PC stage, EX stage, ID stage, and IF stage were studied individually. They found that the power is dominated by two factors: the number of '1's which are in the current binary number (w) and Hamming distance between current and previous values (h). Thus, a power consumption model of each pipeline is given:

$$P = \alpha * h + \beta * w + r, \tag{2.44}$$

where α and β are the weights of h and w respectively, and r is a constant. However, they concentrated on analysing how data can affect the power/energy rather then creating an accurate model, thus they do not prove this hypothesized power consumption model with benchmarks.

Lee *et al.* studied an ARM7TDMI processor and the variables they were concerned with were the instruction fetch address, the register numbers, the immediate operand, and the data values [14]. How different instructions affect the energy of each pipeline stage are studied individually and if V is the set of all the model variables, the energy of each pipeline stage $e_s(X, Y)$ is calculated as

$$e_s(X, Y) = B_s^X + \sum_{v \epsilon V} f_s^X(vX, vY), \tag{2.45}$$

where X and Y stand for different two instructions and B_s^X is the base cost for instruction X at pipeline stage s. $f_s^X(vX, vY)$ is the energy variation affected by the model variable v, such as the instruction fetch address and the register numbers.

This model considers the data dependency effect. The total energy of the processor is the sum of the energy of each stage. However, the pipeline stall was not considered in this model and needs further investigation. Moreover, the validation benchmark is a sample program which contains a random mixture of ARM data-processing instructions, thus it does not prove that this model can be extended to big programs.

Instead of analyzing each stage of the processor, Abrar *et al.* studied the broad overview of the ARM7TDMI with gate-level simulation [3]. A cycle accurate energy model is described as

$$E_i = Base(I_i) + \sum_{s \epsilon S} a^s A_i^s, \tag{2.46}$$

where E_i is the energy consumed in cycle i, $Base(I_i)$ is the base cost of instruction I executed in cycle i, S is the set of signals in the model, a^s is the energy dissipated for unit activity on signal s, and A_i^s is the total activity on signal s in cycle i. They

assumed that the power consumption mainly comes from two parts: execution of the current instruction ($Base(I_i)$), and charging/discharging of the internal nodes such as the data/address buses ($\sum_{s \epsilon S} a^s A_i^s$) [3].

Compared with the previous model in Equation 2.45, this model does not analyse each stage of the pipeline individually but together, which makes the model more concise. However, this model lacks validation in detail, since Abrar *et al.* only explained that the benchmarks were related with mathematical operations and control but without introducing any benchmarks names [3]. It does not consider the effect of pipeline stalls.

Based on the study of the previous cycle accurate models, it is clear that a common method of creating a cycle-accurate model is to study each pipeline stage individually and to find out what can affect the power/energy of each stage. However, sometimes what the software engineers want is a low energy program. Thus, they do not care about the instant power consumption, and this cycle-accurate model is not very useful for this aim. In other words, this model costs too much effort for this aim.

2.5 Functional-level Power Model

The main idea of functional-level power analysis is to spilt the processor into different parts or blocks such as the instruction management unit (IMU), the processing unit (PU), and others. Then, factors which can affect the power/energy of each block are pre-defined. Instead of studying the whole processor, this method studies each block separately. Furthermore, linear regression is used to analyse the relation between these factors and the power/energy of each block. For example, the IMU power may be affected by the rate of instruction dispatching and CPU stall rate [47]. Then, a power/energy model for each block can be generated and the total power consumption is just the sum of each block.

The functional-level power model was firstly introduced by Laurent *et al.* and the target processor is the TMS320C6021 DSP [48]. It has an 11 stage pipeline and supports VLIW instructions (256 bits instruction) and parallelism (up to eight instructions in parallel).

Figure 2.6 shows the components of the processor TMS320C6201. The processor is split into four parts: the Memory Management Unit (MMU), the Instruction Management Unit (IMU), the Processing Unit (PU), and the External Memory InterFace (EMIF). Moreover the factors which can affect the power are the frequency(F), the average number of processing units used per cycle (β), the cache miss rate (γ), the in-external instructions (program) read rate (ϵ) and the in-external data access rate (τ). The IMU can work in four different modes: the MAPPED mode, the CACHE mode, the FREEZE

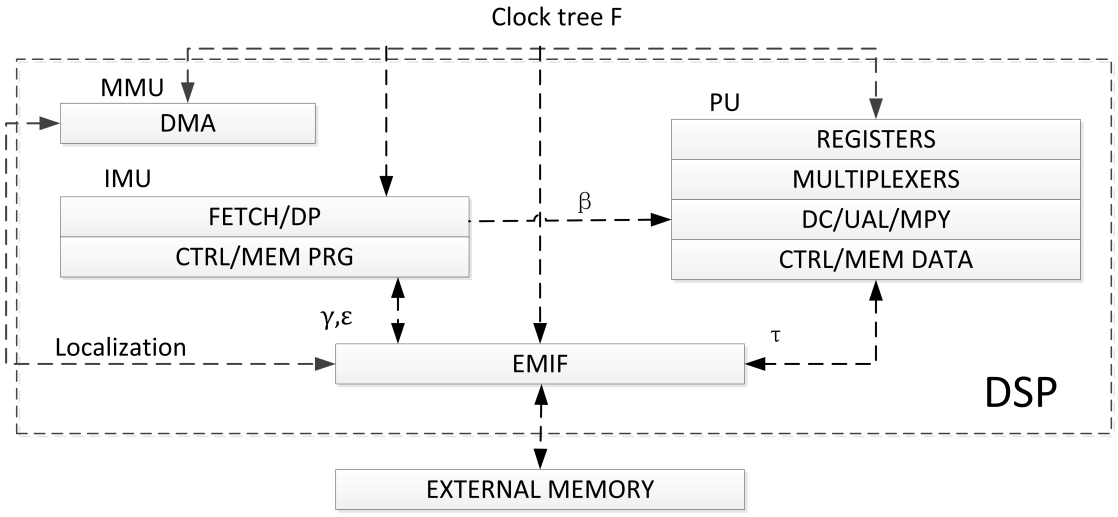


Figure 2.6: Functional analysis for the TMS320C6201 [48].

mode, and the BYPASS mode. The power models of each mode are presented in Table 2.17.

Table 2.17: Consumption rules of IMU for different memory modes of TMS320C6201 [48].

memory modes	consumption rules	
Mapped	I=(aα+b)F+cα +d	F:MHz a=5.21mA/MHz;b=4.19mA/MHz c = 42.4mA; d = 7.6mA
Bypass	I=(a+b)F+c	a=5.68mA/MHz;b=4.19mA/MHz c=38.4mA
Cache	I=$S\alpha F\gamma + T\alpha F + UF\gamma + VF + W\alpha\gamma + X\alpha + Y\gamma + Z$	
Freeze	I=$S\alpha F\epsilon + T\alpha F + UF\epsilon + VF + W\alpha\epsilon + X\alpha + Y\gamma + Z$	

For the CACHE and FREEZE mode, the coefficients S, T, U, V, W, X, Y, and Z are related to the ranges of ϵ or γ. Based on the ranges, the FREEZE mode and the CACHE mode are divided into two cases($\epsilon \leqslant 25\%$ or $\epsilon > 25\%$) and three cases ($\gamma < 50\%$, $50\% \leqslant \gamma \leqslant 75\%$,$\gamma > 75\%$), respectively [48].

Similarly, the power model of the PU is

$$I_{PU} = a\beta F, \tag{2.47}$$

where a equals $0.64mA/MHz$.

Laurent *et al.* presented two models for the IMU and the PU, but the MMU block is not studied. The model works well and the test results of the benchmark FIR 16 shows that the error is less than 7.4%. However, the relationship between S, T, U, V, W, X, Y, Z and ϵ ,γ is not very clearly presented. Moreover, this model only has been proved using FIR 16 and needs more benchmarks to prove the validity of the method.

Senn *et al.* extended this method to a TI C6X processor, a VLIW processor which has a deep pipeline (up to 11 stages) and parallelism capabilities (up to eight operations in parallel) [49].

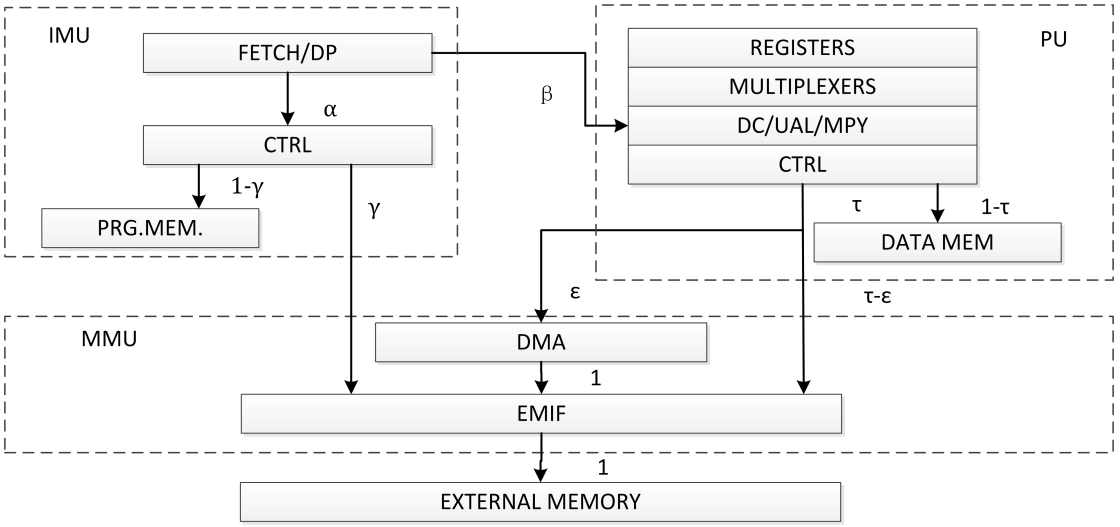


Figure 2.7: Functional analysis for the C6X [49].

Figure 2.7 shows the functional level blocks of the TI C6x. It includes three parts: IMU, PU, MMU. There are five parameters which can affect the power of each block: the parallelism rate (α), the processing rate (β), the cache miss rate (γ), the external data memory access rate (τ), and the activity rate between the data memory controller and DMA (ϵ). They created a model based on four fundamental assumptions:

$$\alpha = \frac{NFP}{NEP} \times (11 - PSR), \tag{2.48}$$

where NFP, NFP, and PSR are the number of fetch packets, the number of execution packets, and pipeline stall rate respectively.

$$\beta = \frac{1}{NPU_{MAX}} \frac{NPU}{NEP} \times (1 - PSR), \tag{2.49}$$

where NPU and NPU_{MAX} are the average number of processing units used per cycle and the maximum number of processing units (NPU_{MAX}=8 for the C6x), respectively.

The pipeline stall rate (PSR) is related to the number of pipeline stall cycles (NPS) and the total number of cycles for execution (NTC). Thus, the following equation can be stated:

$$PSR = \frac{NPS}{NTC}. \tag{2.50}$$

Moreover, the pipeline stalls can be split into three cases: 1. the external data access, which is related to τ; 2. the instruction memory access, which is related to γ; 3. internal

data bank conflicts. Therefore, the total NPS is the sum of these three cases:

$$NPS = NPS_{\gamma} + NPS_{\tau} + NPS_{BS}. \tag{2.51}$$

Senn *et al.* analysed the factors that can affect the power of each block in detail and the model shows good performance with nine benchmarks (the average error is less than 5%) [49]. Moreover, this method has been extend to several different processors: C67, C64, C62, C55, and ARM7 [50].

However there is not a very clear model for the whole CPU system and they only presented α and β for each test [49]. Furthermore, how to use these parameters to estimate the power is not shown clearly. Moreover, the benchmarks do not cover several important cases. For example, they assume the cache miss rate is always zero (γ=0) and the DMA is not used(ϵ=0) [49].

Julien *et al.* extended this work and generate a full power model for TMS320C6201 [51]. The method of dividing the processor into blocks is the same as that of Senn *et al.*, which is shown in Figure 2.7. The parameters and the power model of each block in different modes are shown in Table 2.18 and Table 2.19, respectively.

Table 2.18: Sensitive factors for TMS320C6021 [51].

α	parallelism rate
β	the utilization rate of the processing units
PSR	pipeline stall rate
α'	$\alpha(1 - PSR)$
β'	$\beta(1 - PSR)$
W	data width transferred
γ	the program cache miss rate
τ	the external data memory access rate
ϵ	the activity level between the data memory controller and the direct memory access DMA
F	clock frequency

Compared with previous work done by Laurent *et al.* and Senn *et al.*, the work done by Julien *et al.* has several advantages. Firstly, the model is more complete because it covers every block in different modes, such as IMU, PU, and DMA. Moreover, it shows a clear model that estimates the power based on the pre-defined parameters. Secondly, the performance of this method has been proved by seven benchmarks, such as FIR, FFT, and LMS. Moreover, this method works well in several different processors including C62, C67, C55 and ARM7.

Mostafa *et al.* studied a TMS320C6416T processor, which is a VLIW processor [47]. Figure 2.8 shows the blocks of the processor. This processor is split into three parts: the IMU, processing unit and L1 cache memory. The parameters which can affect the power

Table 2.19: The power models for each block of TMS320C6021 [51].

Block name	current for each block (mA and MHz)
Instruction management unit	$I_{mapped} = 5.21\alpha F + 4.19F + 42.4\alpha + 7.6$
Memory modes_ bypass	$I_{bypass} = 9.87F + 38.4$
Memory modes_freeze	$(9.07F + 118)\alpha'^{[-0.14\log(\gamma)-0.0011]}$
Memory modes_cache	$(8.55F + 184)\alpha'^{[-0.1249\log(\gamma)-0.002276]}$
Processing unit	$I_{PU} = 55.12\beta F$
DMA(F is greater than external memory frequency)	$I_{DMA} = (-0.083WF + 4.9F + 24.93W - 476.16)\epsilon$
DMA(F is less than external memory frequency)	$I_{DMA} = (0.077WF + 2.12F + 2.05W + 94.72)\epsilon$
total current	$I_{total} = I_{IMU} + I_{PU} + I_{DMA}$ where I_{IMU} is one of the current: I_{mapped}, I_{bypass}, I_{freeze}, and I_{cache}

of each block are shown in Table 2.20. Each block is studied separately and Table 2.21 shows the power model for each block.

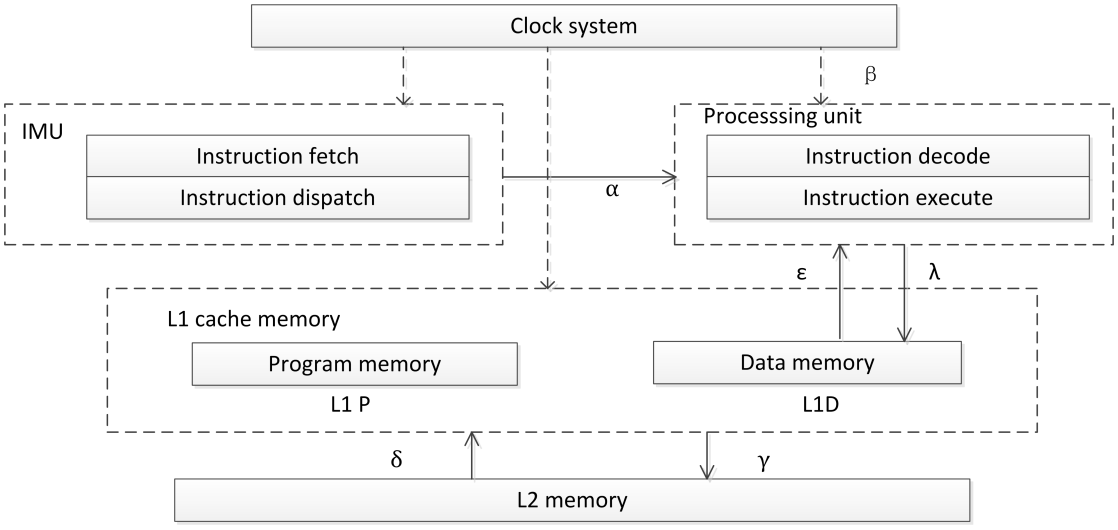


Figure 2.8: Functional analysis for the C6416T [47].

Table 2.20: Methodology for computing algorithmic parameters for C6416T [47].

Parameter	computation methodology
α	No. of fetch packets/No. of execution packets
β	(No. of executed instructions-do not include NOP instructions)/Total code cycles
ε	(No. of L1D read hits/Total code cycles).100
λ	(No. of L1D write hits/Total code cycles).101
δ	((No. of L1D read misses+No. of L1D write misses)/No. of L1D reference).100
γ	(No. of L1P misses/No. of L1P references).100
PSR	No. of CPU stall cycles/Total code cycles

Table 2.21: Complete power consumption model for C6416T DSP at F=1000$MMHz$ [47].

Functional unti	Functional unit power consumption submodel
Clock distribution	$P_{Clock\_Disribution} = (0.0006F + 0.0574) \times V_{core}$
IMU	$P_{IMU} = (-0.0918\alpha^2 + 0.284\alpha + 0.0603)(1 - PSR) \times V_{core}$
Processing units	$P_{PU} = (-0.0049\beta + 0.0065)(1 - PSR) \times V_{core}$
Memory read	$P_{Mem\_Read} = (-2 \cdot 10^{-6}\varepsilon^2 + 0.0012\varepsilon)(1 - PSR) \times V_{core}$
Memory write	$P_{Mem\_Write} = (-10^{-5}\lambda^2 + 0.0049\lambda)(1 - PSR) \times V_{core}$
L1D cache	$P_{L1D} = (-2 \cdot 10^{-5}\gamma^2 + 0.0041\gamma)(1 - PSR) \times V_{core}$
L1P cache	$P_{L1P} = 0.0011\delta(1 - PSR) \times V_{core}$

In this model, there are seven factors and each one may affect some parts of the processor. Compared with the previous functional level models, this model takes the instruction NOP into consideration. Furthermore, it does not use the parallelism factor as the parameter to describe the usage of the processing units (PUs). The reason is that NOP does not require any PUs for its execution. Thus a different parameter (β) to describe the usage of the processor is used. On top of this, the analysis of the effect of cache misses is more detailed. This model also shows that the PSR may reduce the power consumption of each block and we also show similar results in Chapter 4. Nine benchmarks are used to test the performance of the model and the maximum error is 3.3%.

So far, all of the discussed functional level models divide the processor into blocks based on real hardware units. However, based on the different instruction operations, Brandolese *et al*. divided the processor into five functionalities: fetch and decode (F&D), arithmetic and logic operation (A&L), register write operations (WrReg), load and store (Ld&St) and branch (Br) [52,53]. Table 2.22 shows the relationship between operations and related factors.

Different instructions use these functionalities differently. For example, ADD $r1$, $r2$, $r3$ may not use the LD&St block but uses A&L. Then, based on the sensitivity factors,

Table 2.22: The operation class of assembly languages and functionalities [52].

class	operation	functionalities
Arithmetic&logic	add, subtract, and, or, not, exor, multiply, divide, compare ,shift	F&D, A&L
Data transfer	registers, memory, stack	F&D
Control	unconditional jumps, conditional jumps, calls and returns	F&D, Br
System	exception handling, interrupts, system calls	F&D, Br
Floating-point	add,subtract, compare, multiply	F&D, A&L
Decimal	divide, BCD arithmetic conversion	F&D, Br
String	transfer, compare, search	F&D,A&L, WrReg

the energy of different instructions is estimated by the sum of the factors. The energy model is created from sum of the energy of each instruction in an application. This kind of method has been extended to the ARM7TDMI, Intel i960JF, Intel i960HD and SPARClite MB86934, and a very long instruction word (VLIW) processor [33].

One of the advantages of the functional level power model is accuracy because it comprehensively analyses the problem. Moreover, it describes how a program affects the power in detail and software engineers may get help from this to write low power or energy code. However, this method has several disadvantages:

1. To design a functional level power model requires a clear understanding about the chip architecture, to divide it into different blocks. Which parameters can affect each block and how each block affects each other has to be predefined carefully. Otherwise there can be either too few factors for an accurate model or too many redundant factors for a concise model.

2. It takes time to create all of the models for the whole CPU system, because each block needs an individual power model. For example, if the processor is split into four different blocks, at least four power models are required.

3. This method needs a lot of parameters and it takes time to measure all of them. On top of this, it also takes time to use each model to predict the full CPU system power. For example, considering Table 2.20 and Table 2.21, seven parameters need to be measured and seven different individual models have to be used together to generate a full CPU model.

2.6 Architecture-level Estimation

Although gate level simulation and RTL simulation can provide a good result, the simulation speed is a bottleneck. Thus, they are hard to use to study big applications and

software. Some architecture level simulators are designed to address this problem, such as Wattch which is 1000x faster than the existing layout-level power tools [54].

These simulators analyse how the program changes the circuit activity in each clock cycle and use capacitive models to estimate the power. For example, the simulator SimplePower divides the functional unit into two classes: bit-independent functional units (the operation for each slice has no relation with other bit slices, such as the logic unit in the ALU) and bit-dependent functional units (the operation for each slice is related to other bit slices, such as the 32-bit adder) [55]. For a bit-dependent functional unit, the energy characterization is based on a lookup table which restores all of the switch possibilities and capacitances (as shown in Table 2.23).

Table 2.23: Switch Capacitance Table [55].

index		switch Capacitance
previous input vector	current input vector	(pF)
$0_1...0_n$	$0_1...0_n$	cap_0
$0_1...0_n$	$0_1...1_n$	cap_1
$0_1...0_n$	$0_1...10_n$	cap_2
$0_1...0_n$	$0_1...11_n$	cap_3
...	...	cap_0
$1_1...1_n$	$1_1...10_n$	$cap_0 2^n - 2$
$1_1...1_n$	$1_1...11_n$	cap_{2^n-1}

However, there are two challenges in this method. The first one is the size of the lookup table because it grows exponentially with the size of the inputs. The second is the performance cost of accessing the lookup table for each component in a cycle [47].

In order to solve these problems, several different approaches have been developed such as an uncompressed/compressed energy table [56]. Wattch used another method. Without changing the capacitance model of each block, a parameter α was defined and used to describe the switch frequency. They generate α as the input to the model based on the internal cycle level performance simulator [54].

Besides these two widely used simulators, there are several other architecture level simulators, such as EPS [57] and SoftWatt [58]. The common method of these simulators for estimating the power/energy of the processor is to analyze how applications affect the circuit activity during each cycle and to use the capacitive models to estimate the power [47]. Therefore, an architecture-level estimation may be also a cycle-accurate level estimation, such as SimplePower [55].

Architecture-level power simulators have some disadvantages. Firstly, the simulators only support limited RTL level processor models. It is difficult to add commercial modern processor models, as the details of these processors are not available [15]. On the other hand, the simulator has its own disadvantages. For example, SimplePower

simulates an in-order 5-stage pipeline, and some advanced technologies are not used, such as branch prediction and instruction pre-fetch. On top of this, the clock power and control logic power are not implemented [55]. Moreover, low power technologies are not used in Wattch such as clock-gating and dynamic voltage and frequency scaling (DVFS) [54]. Therefore, the power simulator cannot fully replace real measurements.

2.7 System-level Estimation

Sometimes, people more care about the power/energy of the whole system, thus a system-level model is created. The components of different systems vary but normally include the CPU, Memory subsystem, and others, such as LCD/backlight. The idea of the system-level model is to study each component individually, then add the power/energy consumption of each component together.

Nunez-Yanez *et al*. presented a system-level energy model based on the ARM Cortex-A9 dual core processor [59]. The subsystems they studied include CPU, L2 cache, memory controller, interconnect, and LPDDR2 memory device. The method to create the power model of each component is regression. Firstly, they created different test benches with different predefined factors. Then, based on the different energy consumption and factors of the test, the corresponding coefficients are modelled. For example, there are two primary types of activity metrics: state-like metrics, such as the time spent executing or not executing caused by waiting for external memory, and event-like metrics, such as the number of L1 data cache hits or misses. The following equation is the energy consumption model:

$$\sum_{j=1}^{J} P_j T_j + \sum_{k=1}^{K} E_k N_k = \sum_{l=1}^{J+K} m_l a_l = \mathbf{m^T a}, \tag{2.52}$$

where J and K are the number of state like metrics and event like metrics, respectively. P_j and T_j are the power consumption of state j and the time spent on it. Similarly, E_k and N_k are the energy cost of event k and the occurrences of that event. Finally, a simpler expression is used and the activity vector can be presented as: (T_1 T_2...T_j N_1 N_2...N_K)=(a_1 a_2...a_M)=$\mathbf{a}$. Similarly, the unknown power model coefficients, $\mathbf{m}$, can be presented as: (P_1 P_2...P_j E_1 E_2 E_3...E_K)^T=(m_1 m_2...m_M)=$\mathbf{m}^T$. Table 2.24 shows the factors and coefficients of the energy model.

The model shows good estimation over seven benchmarks: a2time, JPEG enc., JPEG dec., FFT, Matrix, Route Lookup, Viterbi, CPU stree. The maximum error is less than 7%. A contribution of this work is to provide a method to estimate the energy at system level at the pre-silicon stage when RTL code is available. On top of this, this method is tested in a state-of-the-art multiprocessor architecture with realistic benchmarks, such

Table 2.24: Power models activity counts [59].

Parameter	Model summary		
	Coefficient value	Type	Description
core_state_active	100	CPU state	Counts the number of cycles the core is in active state executing instructions
core_state_stall	112		Counts the number of cycles the core is in stall state waiting for some additional inputs before progressing
core_state_wfi	0.12		Counts the number of cycles the core is in standby mode with most of the clocks disable
intc_state_clock_enabled	82		Counts the number of cycles during which the integer clock is enabled
neon_state_clock_enabled	51		Counts the number of cycles the neon data engine unit is enabled
Integer_instruction_renaming	0.11	CPU event	Number of instructions going through the register renaming stage
floating_point_instrucion_renaming	0.19		Counts the number of floating point instructions going through the register rename stage
neon_instruction_renaming	0.05		Counts the number of neon instructions going through the register rename stage
d_cache_miss	0.87		Counts the number of L1 data cache accesses that resulted in a data cache miss
i_cache_miss	0.22		Counts the number of L1 instruction cache misses
data_read_hit	101.0/93.5/90.5/57.2	L2 cache event	Counts the number of L2 data cache read accesses that result in a cache hit
data_read_request	131.2/121.0/116.7/83.89		Counts the number of L2 data cache read accesses
data_write_hit	103.9/96.2/92.8/59.52		Counts the number of L2 data cache write accesses that result in cache hit.
data_write_request	131.2/121.0/116.7/83.89		Counts the number of L2 data cache write accesses
Instruction_read_hit	101.0/93.5/90.5/57.20		Counts the number of L2 instruction read accesses that result in a hit
Instruction_read_request	131.2/121.0/116.7/83.89		Counts the number of L2 instruction read accesses
write_channel_cpu_0	0.065	Interconnect event	Counts the number of write transfer in the cpu0/axi interface
read_channel_cpu_0	0.075		Counts the number of read transfer in the cpu0/axi interface
write_channel_cpu_1	0.073		Counts the number of write transfer in the cpu1/axi interface
read_channel_cpu_1	0.078		Counts the number of read transfer in the cpu1/axi interface
write_channel_tg	0.055		Counts the number of write transfer in the tg/axi interface
read_channel_tg	0.043		Counts the number of read transfer in the cpu/axi interface
read_channel_slave	0.055	Memory controller event	Counts the number of read transfer in the memory controller/axi interface
write_channel_slave	0.056		Counts the number of write transfer in the memory controller/axi interface
dfi_write_data_enable	112.5	PHY event	Counts the number of write enables in the dfi/memory chips interface
dfi_read_data_enable	112.5		Counts the number of read enables in the dfi/memory chips interface
activate_command	5.98	LPDDR2 event	Counts the number of activate commands in the memory chips
read_command	2.16		Counts the number of read commands in the memory chips
write_command	1.83		Counts the number of write commands in the memory chips
clock_enable_all_banks_precharge	24.97	LPDDR2 state	Counts the number of clock enable with all banks precharched events the memory chips
clock_disable_all_banks_precharge	0.85		Counts the number of clock disable with all banks precharched events the memory chips
clock_enable_some_banks_precharge	30.46		Counts the number of clock enable with some banks precharched events the memory chips

as internet browsing. However, the disadvantage is the energy model is too complicated and it highly depends on the performance counter. Thus, this method is hard to apply to a processor which does not provide a performance counter or can not supply this detailed information.

Carroll *et al*. analysed the energy usage of a smartphone: Freerunner [60]. The smartphone includes the following main components: CPU core, RAM (both banks), GSM, GPS, Bluetooth, LCD panel and touch-screen, LCD backlight, WiFi, audio (codec and amplifier), internal NAND flash, and SD card. Table 2.25 shows the hardware specifications in detail.

In this paper, Carroll *et al*. tested where the energy goes and the results show that the majority of energy is consumed by the GSM module and the display. Moreover, they

Table 2.25: Freerunner hardware specifications [60].

component	specification
SoC	Samsung S3C2442
CPU	ARM 920T@400MHz
RAM	128 MiB SDRAM
Flash	256 MiB NAND
Cellular radio	TI Calypso GSM+GPRS
GPS	u-blox ANTARIS 4
Graphics	Smedia Glamo 3362
LCD	Topploy 480*640
SD Card	SanDisk 2GB
Bluetooth	Delta DFBM-CS320
WiFi	Accton 3236AQ
Audio codec	Wolfson WM8753
Audio amplifier	National Semiconductor LM4853
Power controller	NXP PCF50633
Battery	1200 mAh, 3.7 V Li-Ion

presented an energy model under a number of typical usage scenarios as follow:

$$\begin{aligned} E_{audio}(t) &= 0.32W \times t \\ E_{video}(t) &= (0.45W + P_{BL}) \times t \\ E_{sms}(t) &= (0.3W + P_{BL}) \times t \\ E_{call}(t) &= 1.05W \times t \\ E_{web}(t) &= (0.43W + P_{BL}) \times t \\ E_{email}(t) &= (0.61W + P_{BL}) \times t \end{aligned} \tag{2.53}$$

where E_{audio}, E_{video}, E_{sms}, E_{call}, E_{web}, and E_{email} are the energy consumption when the phone is in audio playback, video playback, text messaging, phone call, web browsing, and emailing scenarios, respectively. In some conditions, the power is related to the backlight (P_{BL}), such as video.

The advantage of this work is it shows where the energy goes in detail on a smartphone. However, the energy model is the total energy consumption of the whole system, and how the different tests affect the power consumption of each component, such as CPU, and RAM, is not analysed in detail.

Lee *et al.* studied an ARM926EJ-S processor system which includes processors, bus fabrics, custom IP blocks, and memories [61]. In their model, the CPU core logic is considered in two cases: a busy state and an idle state (stalled by interlocks). Moreover, they found that for an ARM926EJ-S processor, the cache power consumption varies a lot, which is from 3% up to 60% of the total power. The advantage is the simulation speed of their method is 100 times faster than gate-level power estimation. However, the authors did not show a clear model.

Shye *et al* studied an HTC smartphone which used a Qualcomm MSM7201A chipset. The CPU is a 528 MHz ARM11 processor [62]. The system includes: CPU, screen, call, EDGE network, Wifi, SD card, DSP, and system (the power that is not considered in the hardware components listed above). Linear regression is used to create the power model. The details about the hardware components and corresponding coefficients are listed in Table 2.26.

Table 2.26: Parameters used for linear regression in the power estimation model [62].

HW unit	Parameter	Description	Range (of $\beta_{i,j}$)	Coefficient (c_j)unit
CPU	hi_CPU_util	Average CPU utilization while operating at 384 MHz	0−100	3.97 $mW/\%$
	med_CPU_util	Average CPU utilization while operating at 246 MHz	0−100	2.79 $mW/\%$
Screen	screen_on	Fraction of the time interval with the screen on	0−1	150.31 mW
	brightness	Screen brightness	0−255	2.07 mW/(step)
Call	call_ringing	Fraction of the time interval where the phone is ringing	0−1	761.70 mW
	call_off_hook	Fraction of time interval during a phone call	0−1	389.97 mW
EDGE	edge_has_traffic	Fraction of time inverval where there is EDGE traffic	0−1	522.67 mW
	edge_traffic	Number of bytes transferred with the EDGE network during time interval	0	3.47 mW/byte
Wifi	wifi_on	Fraction of time interval Wifi connection is on	0−1	1.77 mW
	wifi_has_traffic	Fraction of time interval where there is Wifi traffic	0−1	658.93 mW
	wifi_traffic	Count of bytes transferred with Wifi during interval	≥0	0.518 mW/byte
SD Card	sdcard_traffic	Number of sectors transferred to/from Micro SD card	≥0	0.0324mW/sector
DSP	music_on	Fraction of time interval music is on	0−1	275.65 mW
System	system_on	Fraction of time interval phone is not idle	0−1	169.08 mW

If a single measurement in a sample i is $\beta_{i,j}$, then the corresponding power consumption consumed by the hardware is $p_{i,j}$, which can be presented as:

$$p_{i,j} = \beta_{i,j} \cdot c_j, \tag{2.54}$$

where $\beta_{i,j}$ is the input of the model and c_j stands for the coefficients of each hardware component listed in Table 2.26. Thus, the power consumed by the whole system in sample i is the sum of the power of each component. Assuming the number of coefficients is n, the power consumption can be described as:

$$\begin{aligned} P_i &= k + (p_{i,0} + p_{i,1} + p_{i,2} + ... + p_{i,n}) \\ &= k + ((\beta_{i,0} \cdot c_0) + (\beta_{i,1} \cdot c_1) + (\beta_{i,2} \cdot c_2) + ... + (\beta_{i,n} \cdot c_n)), \end{aligned} \tag{2.55}$$

where k is a constant offset. If for each sample i, x_i=$(\beta_{i,0}, \beta_{i,1}, ..., \beta_{i,n})$, and c=$(c_0, c_1, ..., c_n)$, then Equation 2.55 can be represented as:

$$P_i = k + x_i \cdot c \tag{2.56}$$

If m samples are measured, the power model will become

$$\begin{pmatrix} P_0 \\ P_1 \\ ... \\ P_m \end{pmatrix} = k \cdot \begin{pmatrix} 1 \\ 1 \\ ... \\ 1 \end{pmatrix} + \begin{pmatrix} \beta_{0,0}...\beta_{0,n} \\ \beta_{1,0}...\beta_{1,n} \\ ... \\ \beta_{m,0}...\beta_{m,n} \end{pmatrix} \begin{pmatrix} c_0 \\ c_1 \\ ... \\ c_m \end{pmatrix}. \tag{2.57}$$

Letting P=$\begin{pmatrix} P_0 \\ P_1 \\ ... \\ P_m \end{pmatrix}$, X=$\begin{pmatrix} \beta_{0,0}...\beta_{0,n} \\ \beta_{1,0}...\beta_{1,n} \\ ... \\ \beta_{m,0}...\beta_{m,n} \end{pmatrix}$, and e=$(1...1)^T$, the final power model is

$$P = k \cdot e + Xc. \tag{2.58}$$

If k and c are determined, the whole system power consumption can be calculated from Equation 2.58, and the power consumption of each hardware component can be modelled by Equation 2.54.

Moreover, the total energy consumed by the system across a set of samples X with sampling period, t_s, can be presented as

$$E = t_s \cdot sum(P) = \sum_{i=0}^{m} t_s \cdot P_i = t_s \sum_{i=0}^{m} (k + x_i \cdot c) \tag{2.59}$$

However, the phone may be in an idle state and the power consumption becomes constant ($p_{idle} \approx 68.3mW$). The system_on ratio from Table 2.26 indicates the percent of time that the system is in an active state. Taking the active state and idle state into consideration, the power model becomes

$$Power_i = system\_on \cdot (P_i) + (1 - system\_on) \cdot (p_{idle}), \tag{2.60}$$

when the system is in the active mode, the power is calculated by the linear regression model presented in Equation 2.58. However, in the idle state, the constant P_{idle} is used as the approximation.

The power model has been tested with mobile phone users for a period of time. The test results shows that the power model has a good performance. Furthermore, 65% of the individual samples are estimated with less than 10% error, and 90% of the samples are within 20%. The test results shows that the energy consumption widely varies depending upon the different users. However, the screen and the CPU are the two largest power consuming components [62].

Antti *el al*. studied an ARM 1136 mobile system (a Nokia Internet Tablet N810) [63]. The considered components of the system are the processor, WLAN interface, and display. Different variables are chosen to reflect the different activity levels of each component. For example, hardware performance counters (HPCs) are used to study the processors, the downlink and uplink data rates are used for the WNI, and the brightness level is for the display. They use linear regression to create a full system power model and there are 21 factors which are considered in the model and they are listed in Table 2.27. More specifically, seventeen factors come from the processor, three factors are used to model the WLAN interface, and one is for the display.

Table 2.27: Description of regression variables [63].

Hardware rescource	Considered Factors
Processor	HPCs available on ARM 1136:CPU_CYCLES, DCACHE_MISS, TLB_MISS, ITLB_MISS, CYCLES_DATA_STALL, INSN_EXECUTED, DTLB_MISS, DCACHE_ACCESS, DCACHE_MISS, EXP_EXTERNAL, DCACHE_ACCESS_ALL, IFU_IFETCH_MISS, BR_INST_MISS_PRED, CYCLES_IFU_MEM_STALL, LSU_STALL, PC_CHANGE, BR_INST_EXECUTED.
WLAN Interface	CAM enable,network data rate (download or upload)
Display	Brightness levels on a Nokia N810

Five different categories of benchmarks are used: 1. idle with different brightness levels, 2. audio/video players, 3. audio/video recorders, 4. file download/upload at different network data rates, and 5. streaming. In each category, there may be more than one case to test, such as audio/video recorders, and they are described in Table 2.28.

Although 17 different factors (presented in Table 2.27) were studied for the processor, in order to make the power model concise, only the three most important factors are taken into consideration, and they are CPU_CYCLES, DCACHE_WB and TLB_MI. After analysing the relationship between the power and the predefined factors, the following power model is generated:

Table 2.28: Descriptions of the workloads used in our energy benchmark [63].

Category	Description	Test Case
Idle with Different Brightness Levels	CPU and memory workload: Low Wireless connection: No Brightness level: 0-5	Keep the system idle without running any applications and set the brightness level of the display to different values.
Audio/Video Players	CPU and memory workload: Low-High Wireless connection: No Brightness level: 0 for audio player; 5 for video player.	Media player on N810: mplayer Media file storage: Phone memory Audio format: MP3, OGG, RM Number of audio players in parallel: 1, 2, 3 Video format: AVI, MPEG Number of video players in parallel: 1, 2
Audio/ Video Recorders	CPU and memory workload: Medium Wireless connection: No Brightness level : 5	Run an embedded audio recorder to record an audio file played on a machine close to the experimental device. Use the embedded camera to record a video.
File Download/ Upload at Different Data Rate	CPU and memory workload: Low-High WLAN connection: On Network data rate: 16KB/s-400KB/s. Brightness level: 0	N810: netcat Linux Server: netcat, Trickle (bandwidth limiting utility) Data rate limit: 16, 32, 128, 256, and 400KB/s. CAM: On/off (data rate $<$ 32KB/s); Off (data rate$\geqslant$32KB/s) Download Storage: phone memory, /dev/null Upload storage: phone memory
Streaming	CPU and memory workload: High WLAN connection: On WLAN Power saving mode: Enabled Brightness level: 5	Watch online TV programs transferred from www.itv.com. Encoding rate: 16-72KB/s Listen to radio programs from three different radio websites. Download date rate: around 24KB/s. Use web browser to watch YouTube videos online. Download data rate: 46-136KB/s depending on the network conditions.

$$
\begin{aligned}
Power(W) =& 0.7655 + 0.2474 \times g_0(x_0) + 0.0815 \times g_1(x_1) + 0.0606 \times g_2(x_2) \\
&+ 0.0011 \times g_{17}(x_{17}) + 0.0015 \times g_{18}(x_{18}) + 0.3822 \times g_{19}(x_{19}) \\
&+ 0.1255 \times g_{20}(x_{20}) \\
g_0(x_0) =& \frac{x_0 - 1316.84}{1349.423}, x_0 = \frac{c_0}{d}, \\
g_1(x_1) =& \frac{x_1 - 0.000901}{0.00045}, x_1 = \frac{c_1}{c_0}, \\
g_2(x_2) =& \frac{x_2 - 1316.84}{1349.423}, x_2 = \frac{c_2}{c_0}, \\
g_{17}(x_{17}) =& x_{17},\ x_{17} : download\ data\ rate\ (KB/S), \\
g_{18}(x_{18}) =& x_{18},\ x_{18} : upload\ data\ rate\ (KB/S), \\
g_{19}(x_{19}) =& x_{19},\ x_{19} : CAM\ switch, \\
g_{20}(x_{20}) =& x_{20},\ x_{20} : brightnesslevel,
\end{aligned}
\tag{2.61}
$$

where c_0, c_1, c_2, d are the increments in CPU_CYCLES, DCACHE_WB, TLB_MISS, and monitoring period, respectively. Seven benchmarks are used to validate the performance of the power model: radio, liveTV, YouTube, audio recorder, video recorder, upload, and download. The biggest error comes from the video recorder, which is 13.7% and the least error is 0.2%.

The advantage of the model is conciseness and that real hardware is used. The accuracy is also good except the video recorder test. However, the system does not consider the energy cost of the memory, which is one of the most important subsystems of a mobile system.

2.8 Some Other Related Research

Russell *et al.* studied the 80960JF processor (one instruction/clock execution, 32-bit processor, 4kB instruction cache, 3.3V supply voltage [64]) and the 80960HD (32-bit parallel architecture processor, 16kB instruction cache and 8kB data cache, 3.3V supply voltage [65]) and found that most of the instructions consume similar power [66]. The power consumption of each instruction is listed in Table 2.29. Based on these results, they created a simple model in which the power is modelled with a constant parameter and is not related to the instruction types. Four benchmarks (psdes, heap, fft and moment) were used to test the performance of the model and the maximum error was -8.5%.

Table 2.29: Subset of power measurements of 80960JF and 80960HD [66].

	JF Processor			HD Processor		
Instr	P_{ave}	s	Cyc.	P_{ave}	s	Cyc.
add	1.77	0.03	1	2.87	0.03	0.5
sub	1.74	0.02	1	2.85	0.03	0.5
mul	1.72	0.003	5	2.80	0.02	2.5
div	1.62	0.02	35	2.43	0.02	17.5
mod	1.62	0.02	35	2.49	0.02	17.5
rotate	1.79	0.03	1	2.85	0.03	0.5
and	1.75	0.03	1	2.86	0.04	0.5
xor	1.76	0.02	1	2.87	0.03	0.5
setbit	1.75	0.03	1	2.87	0.03	0.5
bswap	1.62	0.02	10	2.81	0.04	0.5
mov	1.67	0.02	1	2.83	0.03	0.5
extern.ld	1.71	0.01	9	3.18	0.02	6
interrn.ld	1.75	0.02	2	3.40	0.03	0.5
extern.st	1.97	0.05	5	3.23	0.02	5
cmp	1.63	0.02	3	2.76	0.03	0.5
cmpdec	1.62	0.02	3	2.78	0.03	0.5
bswap	1.40	0.01	2	2.57	0.02	1
call/ret	1.56	0.01	14	2.95	0.02	5

However, all of these test results assume cache hits and the effect of cache misses is not considered. Therefore, a program which has a lot of cache misses may not be estimated correctly by this model. On the other hand, the 80960HD is a superscalar processor and the effect of dual-issue is not analyzed. In Chapter 5, we choose a superscalar processor, ARM Cortex-A8, as an example and analyze the instruction level power in detail.

Sultan *et al.* also found similar results for LEON3 which was implemented in an FPGA [67]. Furthermore, the range of the power consumption of Load/Store instructions is from 590 nW to 640 nW and the range of all of the other instructions (including arithmetic instructions, logical instructions, control instructions and shift instructions) is between $550nW$ and 568 nW.

Sinha *et al.* studied the StrongARM processor (a 32-bit scalar processor with a five-stage classic RISC pipeline, 16kB instruction/data cache. [68]) and they found that the current variation of six different benchmarks (fft, fir, log sort, dhry, dct) was less than 8% [69]. Therefore, they assumed that the current was only related to the supply voltage (V_{dd}) and clock frequency (f), and created a simple energy model:

$$E_{tot} = V_{dd} I_0(V_{dd}, f)\Delta t, \tag{2.62}$$

where V_{dd} is the supply voltage, and Δt is the runtime of the program. In this model, they assumed that the current, $I_0(V_{dd}, f)$, was only related to the power supply voltage and frequency of the processor.

However based on the instruction level current test, they found the maximum variation for different instructions was about 38%. Therefore, the constant model may not be accurate if a program contains a big percentage of instructions which consume more power. In order to avoid this problem, they generated another more accurate model:

$$I(V_{dd}, f) = I_0(V_{dd}, f) \sum_{k=0}^{K-1} w_k c_k, \tag{2.63}$$

where w_k is a set of weights and c_k is the fraction of total cycles of a program, i.e, $\sum c_k = 1$. The cycles are divided into four classes as shown in Table 2.30.

Table 2.30: Weighting factors for K=4 on the Strong ARM.

Class	Weight	Value
Instruction	w_1	2.1739
Sequential memory access	w_2	0.0311
Non-sequential memory access	w_3	1.2366
Internal cycles	w_4	0.8289

This model is concise because it only needs four inputs. The model is based on the six different benchmarks, thus it should be tested by other tests to prove the validation. However, there are not enough benchmarks to test the model. Moreover, how to get the fraction of each class is not described.

Konstantakos *et al.* clustered the instructions based on the instruction clock cycle using a Motorola HC908GP32 processor (a 8-bit, 4.9152 MHz, $3V/5V$ power supply, micro controller [70]) [18]. The energy consumption of each group is described in Table 2.31.

Table 2.31: Average energy consumption of Motorola HC908GP32 instructions.

instruction group	coefficient	average energy per cycle(μJ)	average deviation
1-cycle	cp1	0.05355	0.53%
2-cycles	cp2	0.0566	2.85%
3-cycles	cp3	0.0585	5.50%
4-cycles	cp4	0.0580	3.19%
5-cycles	cp5	0.0576	1.34%%

Konstantakos *et al*. created an energy model:

$$\begin{aligned} E_{microcontroller} =& cp1.\#instructions1 - cycle + cp2.\#instructions2 - cycles \\ &+ cp3.\#instructions3 - cycles + cp4.\#instructions4 - cycles \\ &+ cp5.\#instructions5 - cycles, \end{aligned} \tag{2.64}$$

where the variables $cp1, ..., cp5$ are the average energy for each instruction group [18]. This table also proves the results presented by Russell *et al*. in Section 2.8, because the energy per cycle is quite similar. However, this model does not consider the effect of cache misses and the benchmarks used to validate are inefficient (only one benchmark: a data-logging application).

Shao *et al*. created a new instruction-level energy model for Intel's Xeon Phi Processor ($22nm$ and containing 60 cores running at 1.09 GHz; each core contains a 512-bit vector processing unit, a 32 kB L1 instruction/data cache, and a 512kB private L2 cache) and the key factor is the energy per instruction EPI [71]. Firstly, they designed different tests to generate the EPI for different instructions with different numbers of cores and threads. Then, the total energy is calculated from the instruction counts multiplied by the corresponding EPI. This idea is similar to the basic model in Section 2.1.3. However, the advantage of this model is it considers multi-core and multi-thread programs. Moreover, this model is tested by five benchmarks (md, scan, reduction, stencil linpack_29k) from the SHOC benchmark suite and the error is less than 5% [71]. However, this method is highly dependent on the Intel performance counter and this method cannot be used, if another processor's performance counter cannot provide that data in detail.

2.8.1 The Effect of Cache

The cache is needed because it solves the speed mismatch problem between the CPU and main memory. If a cache miss happens, it is necessary to fetch the missing instruction from a lower level memory, such as main memory. However, visiting a lower level memory will cost a lot more energy [7, 8]. Moreover, cache misses will lead to longer times to execute the program.

When cache misses exist, for an in-order scalar processor, the pipeline will stall and the power consumption will be lower [8]. For example, the base power cost of instruction MOV in the 40MHz Intel 486DX2-S Series CPU (8kB cache) is $1021.68mW$ ($1021.68mW$=$309.6\ mA \times 3.3V$) but when cache misses, the average power is $712.8mW$ ($712.8mW$=$216\ mA \times 3.3V$) [8]. However, considering the time spent on fetching data from lower memory, the total energy will be more.

Sridhar *et al.* presented a method to calculate the average cache miss penalty [24]:

$$Ave.CacheMissPenalty = \frac{E_{code} - \sum_{i=0}^{Num.\ of\ Instructions} BaseCost_i}{Number\ of\ Cache\ Misses} \tag{2.65}$$

where E_{code} is the total energy consumption of the program and $BaseCost_i$ is the base cost of each instruction.

Based on Equation 2.65, the total cache penalty is given by the following equation :

$$\begin{pmatrix} Cache \\ Miss \\ Penalty \end{pmatrix} = \begin{pmatrix} Number \\ of \\ Memory \\ Access \end{pmatrix} \times \begin{pmatrix} Cache \\ Miss \\ Rate \end{pmatrix} \times \begin{pmatrix} Avg. \\ Cache \\ Miss \\ Penalty \end{pmatrix} \tag{2.66}$$

It is clear that the total cache miss penalty is related to three factors: the number of memory accesses, the cache miss rate, and the average cache miss penalty. However, this method is not validated by any benchmarks. Moreover, these three factors are not easy to get without a cycle accurate simulation, which takes a long time.

On top of this, the cache miss penalty is harder to measure for a modern processor, such as an out-of-order processor and non-blocking cache design. In this design, if a cache miss happens and the following instructions are not related to the missing data, the processor keeps running without waiting for the missing data. Thus the processor usage is more efficient. However, it is hard to tell for how many clock cycles the processor will stall because of a cache miss, since the penalty varies.

On the other hand, a lot of modern processors use a random replacement policy to decide which cache line is evicted, such as the TI AM3359 processor (it uses the ARM Cortex-A8 as the CPU) [72]. Because of this replacement policy, it is hard to measure the cache miss penalty and to calculate an accurate cache miss rate for a program.

Moreover the memory hierarchy becomes complex and a lot of modern processors have several different level caches. Thus, the miss penalty of a level one cache is very different from level two. However, cache misses coming from level one or level two are highly dependent on the program and cache design.

2.8.2 The Effect of The Different Hazards

There are three types of hazard: structural hazards, data hazards and control hazards [21]. These hazards may affect the performance of the processor by stalling the pipeline. If a pipeline stalls, it will consume extra energy because the runtime of the program will be longer. However, this part of the energy penalty is not considered in the previous work, since they do not show a clear method to estimate it.

Structural hazards occur when the resources are not replicated sufficiently and several instructions want to get data from the same resource at the same time. Data hazards occur when an instruction's input value depends on the result of a previously uncompleted instruction. Therefore, the pipeline has to wait until the required data is ready. Control hazards occur when a branch exists in the program and the pipeline has to flush if wrong instructions are executing.

There are some existing technologies to solve these problems, such as forwarding logic and pipeline scheduling, but they do not address them completely. Therefore, the pipeline may stall because of these hazards.

Penolazzi *et al*. introduced a data dependency test based on the simulation of a SPARC Leon3 processor and Table 2.32 shows the energy of the test pairs [11] .

Table 2.32: Registers conrrelation analysis [11].

Pair	With corr.		No corr.		cyclesDiff.
	E(pj)	cyc.	E(pj)	cyc.	
ld_add	269.66	4	125.53	**2**	2
smul_smul	332..46	11	495.41	10	1
xor_st	180.85	3	184	3	0
sll_add	136.2	2	141.34	2	0

Table 2.32 is split into two groups horizontally. The upper portion is for when the register correlation increases the number of cycles, such as the ld_add pair. The *add* has to wait until the ld instruction is finished, since one of the operands of *add* comes from the previous instruction, *ld*. Therefore, it will take two extra cycles and consume more energy. For the lower group, the data dependency does not affect the execution time. The authors explained that the first instruction in the pair only takes one cycle to finish which is too short to allow any level of parallelization.

Sridhar *et al*. explained how to calculate the pipeline stall penalty [24]. The pipeline stall is divided into two cases: conditional and unconditional stall. The following equations show how to calculate them, respectively:

$$\begin{pmatrix} Stall \\ Penalty \end{pmatrix} = \begin{pmatrix} Uncond \\ Stall \\ Penalty \end{pmatrix} + \begin{pmatrix} Cond \\ Stall \\ Penalty \end{pmatrix} \tag{2.67}$$

$$\begin{pmatrix} Uncond \\ Stall \\ Penalty \end{pmatrix} = \begin{pmatrix} Number \\ ofCalls \\ Uncond. \\ Branches \end{pmatrix} \times \begin{pmatrix} Instr \\ Stall \\ Penalty \end{pmatrix} \tag{2.68}$$

$$\begin{pmatrix} Cond \\ Stall \\ Penalty \end{pmatrix} = \begin{pmatrix} Number \\ ofCond \\ Branchs \end{pmatrix} \times \begin{pmatrix} Branch \\ Probability \end{pmatrix} \times \begin{pmatrix} Instr. \\ Stall \\ Penalty \end{pmatrix} \tag{2.69}$$

However, this method may not apply to modern processors because modern processors are able to predict the decision and the target address of a conditional branch. If a branch is predicted correctly, it will not cause a pipeline stall. In other words, only the branches which are predicted incorrectly by the branch predictor can cause a pipeline stall.

There are a lot of reasons for a pipeline stall, such as a multi-cycle instruction, data hazards, incorrect prediction of jumps, and cache misses [15]. However, when a pipeline stalls, the processor has to wait and do nothing until the problem is solved. Therefore, if a model can analyse these different reasons together rather than individually, the model will be concise.

2.8.3 The Effect of Memory and The Various Address Models

Tiwari *et al*. mentioned that the memory and the various addressing modes can affect the energy very much [7]. They studied the Intel x486 processor and found that instructions with memory operands consumed more energy compared to instructions with register operands. For example, instructions using only register operands cost about 300 mA. However, memory reads that hit the cache cost upwards of $430mA$ and memory writes cost upwards of 530 mA when the cache is a write-though cache. The difference is nearly double. Thus, the power/enenrgy model has to consider the various addressing modes [7].

2.9 Conclusion

2.9.1 Summary of The Previous Work

This chapter has discussed an overview of different power and energy models at various levels. The basic model (discussed in Equation 2.2 in Section 2.1) is the first instruction level model developed 20 years ago [8]. However, many other models are based on it, such as the idea of basic energy and overhead energy [15, 42, 73]. However, this model has several disadvantages. The first disadvantage is that the overhead energy needs too many experiments to cover all of the possibilities of different pairs. The second one is that the model does not consider the effect of cache misses.

In order to solve these problems, a lot of research has been done, such as the NOP model [30] (Section 2.2) and clustering instructions [33, 37] (Section 2.3). For the NOP model, a new definition of the base power/energ cost and overhead energy are created, which reduced the measurements from $O(N^2)$ to $O(N)$, where N is the number of instructions in the ISA [30]. On the other hand, for clustering instructions, the instructions are divided into different groups based on the power or energy consumption. Instructions that come from the same group do not have overhead power/energy in the power/energy model.

Linear regression is a common method to derive the power and energy model and a lot of models have been developed based on this method [5, 16, 47, 74]. Firstly, some variables are predefined, then different tests are run to gather the data about how these variable affect the power or energy. Finally, different coefficients are given to different parameters (Section 2.4).

A functional-level power model has been used to study VLIW processors [48,51]. Firstly, the processor is divided into different functional blocks. Then, what can affect the power of each block is studied separately. Finally, the total power of the processor is the sum of the power of each block.

Some architectural level power estimation software has been developed. The advantage of these tools are quick estimations and they can present cycle accurate power information. However, these tools are hard to use for commercial modern processor models since the details of these processors are not available.

2.9.2 Trends and Changes

Nowadays, processors have been well developed compared with 20 years ago and a lot of technologies which can make the processor have a better performance are used and further developed, such as out-of-order pipelines, instruction prefetch units, and branch

prediction. For example, the cache sizes are bigger and multi-level caches are widely used, such as in the Intel i7 (three level cache, L1 instruction/data cache is 32kB, L2 is 256 kB, and L3 is 2 MB per core) [21]. The pipeline becomes much deeper than before. For example, the ARM Cortex-A8 contains a three-stage instruction fetch pipeline, five-stage instruction decode pipeline and a five-stage instruction execute and load/store pipeline [75].

Besides the change to the computer architecture, the manufacturing technology has also improved. Figure 2.9 shows how fabrication technology has improved. It is clear that the size of the CMOS is smaller and smaller. A lot of previous models are based on old manufacturing technologies. For example, Tiwari *et al*. studied a Intel 486DX2 which used 1 micron technology [76]. However, nowadays, 65 *nm* and 45 *nm* are widely used for high performance processors.

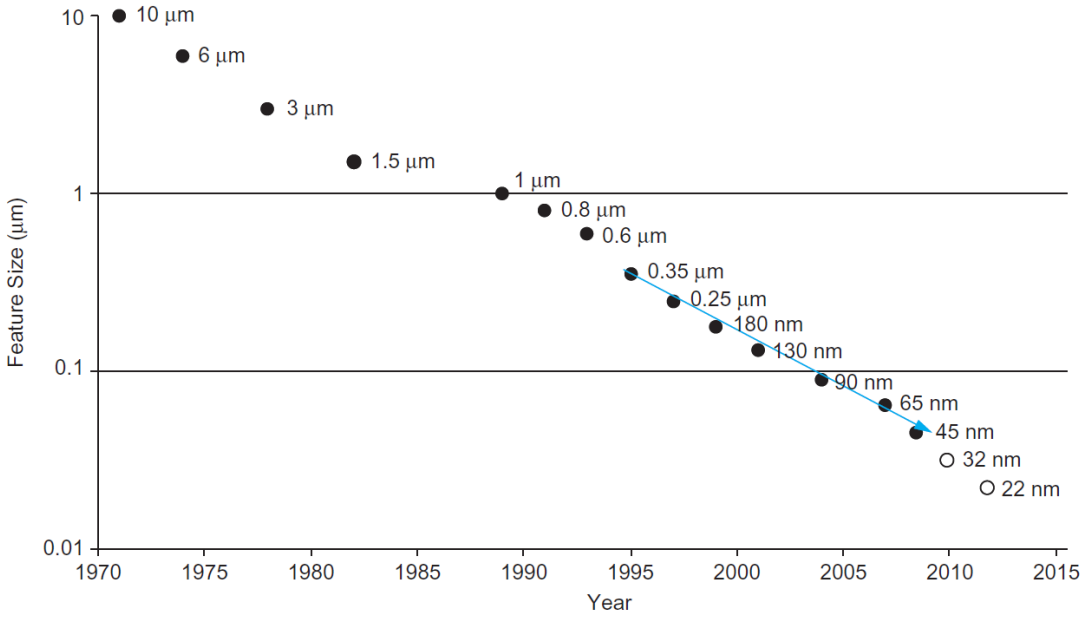


Figure 2.9: Process generations [77].

Because of the development of the manufacturing technology, the clock speed of the processor has increased. For example, Figure 2.10 shows the development of the Intel processors.

On the other hand, the RISC processor becomes more and more popular, especially in embedded systems.

The power/energy models have become more and more complex without improving accuracy much, such as functional level power models. Based on these changes, a new model or method to estimate the power and energy of a program is required. However, the energy model is hard to create because there are a lot of factors that affect the energy of a program. For example, the types of instructions, the Hamming distance of two adjacent instructions and the pipeline stall can all affect the energy consumption.

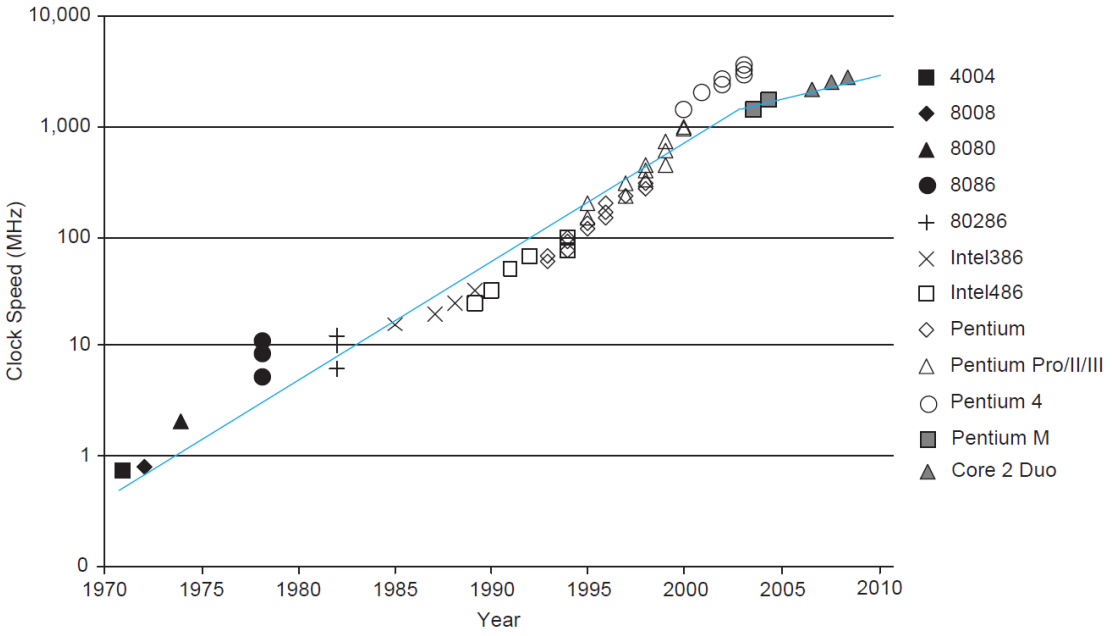


Figure 2.10: Clock frequencies of Intel microprocessors [77].

One of the most important factors is the pipeline stall, but there are also a lot of causes that can make the pipeline stall, such as cache misses, write buffer limitation, etc. Thus, an energy model which considers everything is hard to create.

Instead of establishing the energy model directly, we think it is easier to formulate the energy of a program in two steps: 1) creating the power model, and 2) measuring the runtime. The runtime of the program can be both measured easily, such as by the program counter, and simulated by modern instruction set simulators, such as gem5 [78]. Therefore, the power model can also be extended to study the energy easily.

Chapter 3

OpenRISC

3.1 Target Processor

We choose OpenRISC as our first target processor because it is open source, which can let us easily analyse the inside behaviour of the processor. The OpenRISC 1200 is a 32-bit Harvard architecture scalar RISC processor with a five stage integer pipeline and some basic DSP capabilities. It supports both instruction and data caches with the inclusion of an MMU [19]. The outside data bus and address bus use the WISHBONE standard. OpenRISC 1200 can be changed by the users, for example, to delete or reduce the area of cache. Therefore, OR1200 is a high performance, low power, extensible RISC processor.

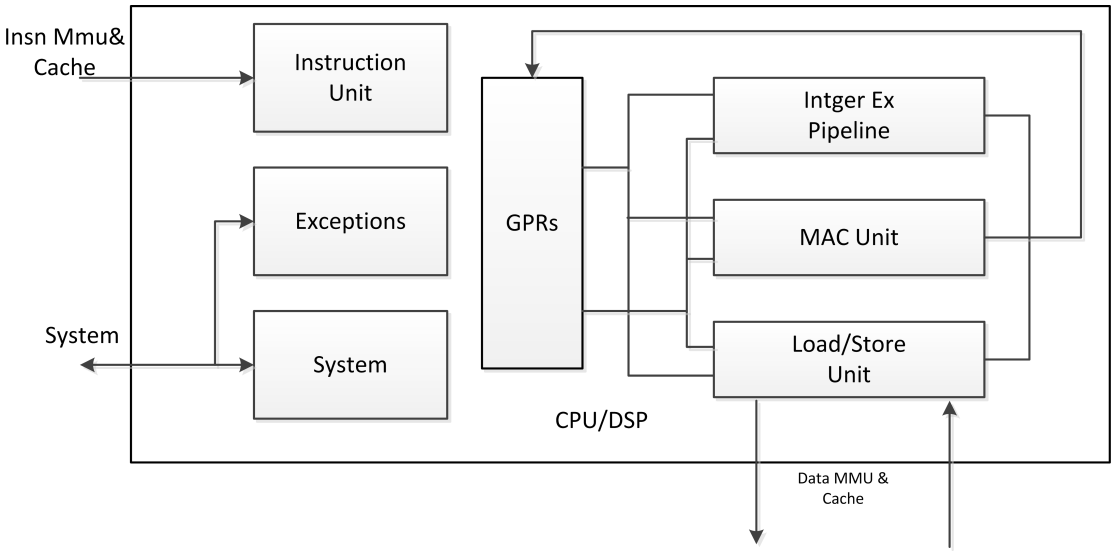


Figure 3.1: The architecture of CPU/DSP [79].

Figure 3.1 shows the architecture of the CPU/DSP which is 32-bit and the central unit of the OR1200 processor. This CPU consists of seven parts: instruction unit, exceptions,

system, GPRs(General Purpose Registers), Integer Execution Pipeline, MAC Unit, and Load/Store Unit. [79]:

- Instruction Unit
  The instruction unit implements the basic instruction pipeline. It is used to fetch instructions, dispatch and record the history to make sure that the operations finish in the right order. Moreover, it also executes the jump instructions, including conditional jump and unconditional jump.

- Exceptions
  The exceptions unit is used to handle the exceptions which come from external interrupt requests, certain memory access conditions, internal errors, system calls, and internal exceptions.

- System Unit
  The System Unit is used to link all other signals of the CPU/DSP that are not connected through instruction and data interfaces. It implements all system special-purpose registers, such as the supervisor register.

- General-Purpose Registers
  There are 32 general-purpose 32-bit registers in the OpenRISC processor and each general-purpose register file has as two synchronous dual-port memories.

- Integer Execution Pipeline
  Integer execution pipeline is the core of the pipeline which is used to implement arithmetic instructions, compare instructions, logic instructions, and rotate and shift instructions.

- MAC Unit
  The MAC unit is used to execute DSP MAC operations and is fully pipelined. Thus, it can accept one new MAC operation (32×32 with 48-bit accumulator) in each new clock cycle.

- Load/Store Unit
  The Load/Store unit is used to transfer data between the GPRs and the CPU internal bus. When load/store instructions are issued, the LSU needs to check if all of the operands (including the address register operand, the source data register operand (for store), and the destination data register operand (for load)) are available.

3.2 Experimental Tools

3.2.1 Synthesis tool: Design Compiler

A standard RTL synthesis tool, Synopsys Design Compiler, was used to implement the OpenRISC to a seven layer metal, $1.05V$, low-power, high-threshold-voltage $65nm$ process. A maximum clock speed of 111MHz was achieved. Two files are generated: a netlist file and a Standard Delay Format file (SDF), which is used to record the delay of each gate and pin.

3.2.2 CMOS power dissipation and power analysis tool: Primetime

3.2.2.1 CMOS power dissipation

For a CMOS circuit, the power consumption can be divided into two main categories: dynamic power and static power. The dynamic power results from transistor are switching. The static power is the power consumed when the transistors are stable [80].

Static Power

The static power includes three parts: sub-threshold leakage current when the transistors are off, tunnelling current through the gate oxide, and leakage current through reverse biased diodes. However, the most significant part is the first one: source-to-drain sub-threshold leakage current. The sub-threshold leakage occurs because when the transistors are off, there will still be a little current which prevents the gate from completely turning off. The following equation shows how to calculate the static power [81]:

$$LeakagePower = V \times I_{leagage} \tag{3.1}$$

Dynamic Power

Yip et al. explained that dynamic power is caused by changing the voltage on a net due to some stimulus [82]. Furthermore, dynamic power consumes the most significant part of the power consumption and the CMOS power analysis tool, Primetime, divides it into two parts: the switching power and internal power.

Switching Power

When a net is switching, the CMOS circuit needs to charge the various capacitive loads of outputs and this part of the power consumption is called the switching power. The following equation shows how to calculate the switching power:

$$SwitchingPower = \frac{1}{2} \cdot C_{load} \cdot V^2 \cdot f, \tag{3.2}$$

where C_{load} is the sum of the net and gate capacitances on the driving outputs and the frequency. f is the rate of state transitions [82].

Internal Power

Internal power is caused by the charging of the internal loads. For example, for a NAND gate, the input A equals one, the input B equals zero and the output is one. However, if the input A becomes zero and the input B becomes one, the output is the still same but the internal signals have switched and the gate will consume power from charging internal capacitances. Therefore, this part of the power consumption inside the gate belongs to internal power.

The other part of internal power is short circuit power. When a CMOS gate is switching, both the NMOS and PMOS transistors may conduct for a very short time and this is called short circuit power. The following equation shows how to calculate interrnal power:

$$InternalPower = \frac{1}{2} \cdot C_{int} \cdot V^2 \cdot f + V \cdot I_{sc} \tag{3.3}$$

The diagram below shows how these different power figures relate to a simple buffer cell.

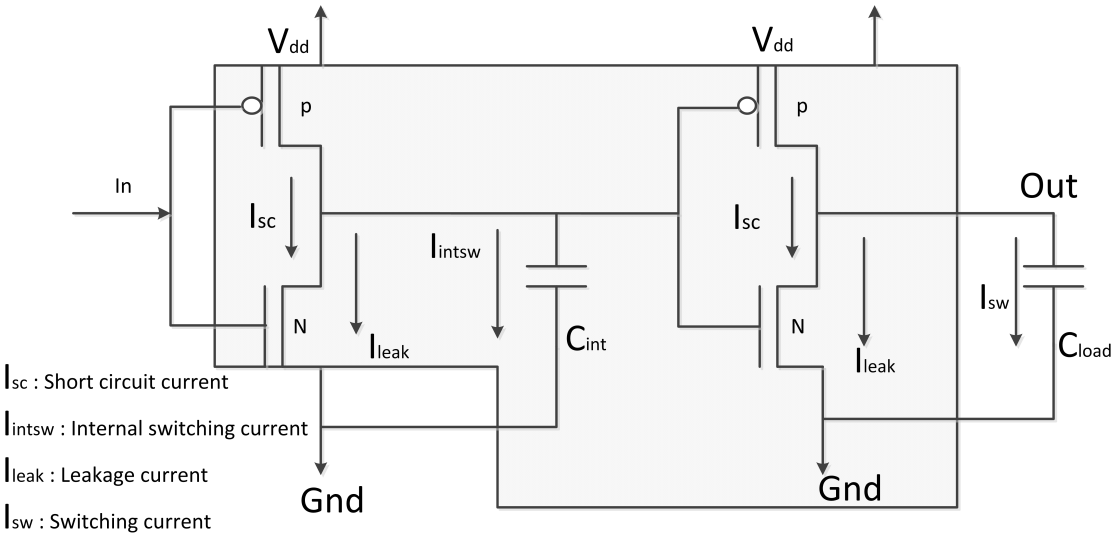


Figure 3.2: Components of power dissipation [82].

From Figure 3.2, the leakage current I_{leak} varies depending on the transistor states. For example, the leakage will be different when the N transistor is on or off. When the input signal In changes from low to high, the I_{sc} of the left inverter will change because the N transistor turns on and the P transistor turns off. Thus, the internal power is consumed due to the switching of I_{sc} and I_{intsw}, and charging and discharging C_{int}. Additionally, the switching current on the Out net charges and discharges C_{load}.

3.2.2.2 Power analysis tool: Primetime

Primetime is designed by Synopsys and provides users a single convenient platform to perform full-chip, power analysis, concurrent timing, and signal integrity. Primetime can analyse all of the power consumption discussed above very accurately. On top of this, the power analysis engine of Primetime supports composite current source (CCS) power models, which are used to model the CMOS cell library very accurately and also supports multiple signal activity formats [82].

The following are the requirements for using Primetime:

• Netlist Data: the Netlist data includes the information about the connectivity and types of the cells.

• Cell library power models: the cell library is provided by library vendors and it contains the cell models for each cell, such as the information about the static and dynamic power consumption.

• Signal activity: The dynamic power is directly proportional to the switching rate of CMOS. The Value Change Dump file (VCD)records how signals switch.

3.3 Experimental Methodology

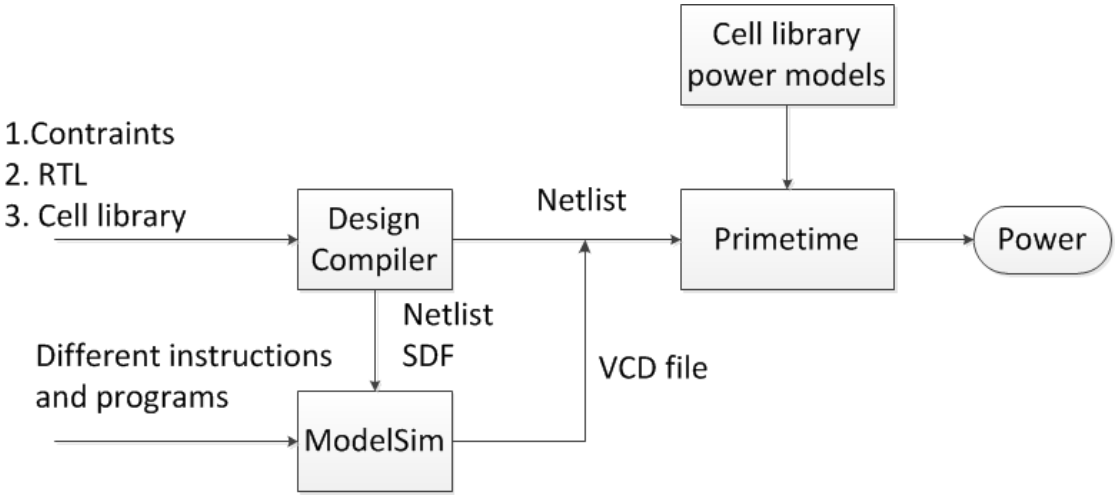


Figure 3.3: The test flow of OR1200.

Figure 3.3 shows the main test flow:

Firstly, based on the discussion in Section 3.2.1, the RTL of OpenRISC is synthesized by Design Compiler, and a Netlist and an SDF file are generated.

Secondly, in order to load the tests into the memory and debug the system, there are several files that need to be generated including: input.c, input.elf (the executable and

linkable format), input.bin (the binary code), input.asm (the assembly code), and input.vmem (verilog memory). The relationship between each file is shown below in Figure 3.4, and the OpenRISC tool chain supply all of the compiling and linking tools (the compile commands are shown in Appendix A.1.1).

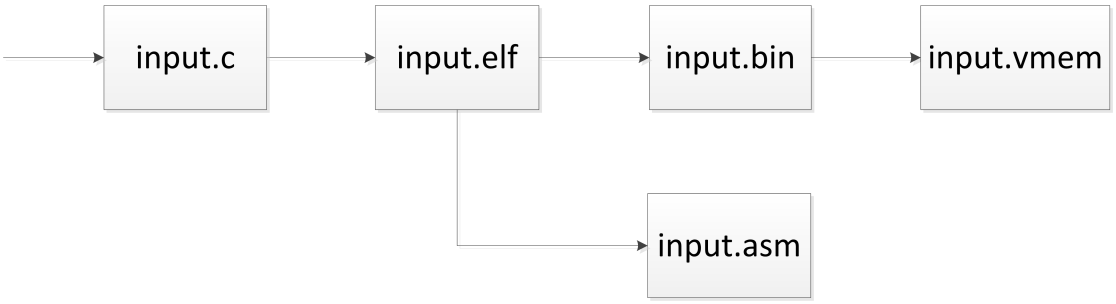


Figure 3.4: OR1200 tool chain.

In Figure 3.4, the input.c file is the source code which can be compiled into an input.elf file. The input.elf can be compiled into an input.bin code and an input.asm code. The input.asm code is the assembly code which includes the machine code and mnemonic and can be used to debug. Then, the input.bin file can be compiled into a input.vmem file which is our target file and it records all of the machine code.

The following is an example of the compiled input.asm code and the corresponding input.vmem code.

The assembly code

```
00002250 <main>:
    2250:   d7 e1 17 fc     l.sw 0xfffffffc(r1),r2
    2254:   9c 41 00 00     l.addi r2,r1,0x0
    2258:   9c 21 ff e0     l.addi r1,r1,0xffffffe0
    225c:   9c 60 00 00     l.addi r3,r0,0x0
    2260:   d7 e2 1f f4     l.sw 0xfffffff4(r2),r3
    2264:   9c 60 00 00     l.addi r3,r0,0x0
    2268:   d7 e2 1f f0     l.sw 0xfffffff0(r2),r3
    226c:   9c 60 00 00     l.addi r3,r0,0x0
    2270:   d7 e2 1f ec     l.sw 0xffffffec(r2),r3
    2274:   9c 60 00 00     l.addi r3,r0,0x0
    2278:   d7 e2 1f e8     l.sw 0xffffffe8(r2),r3
    227c:   9c 60 00 00     l.addi r3,r0,0x0
    2280:   d7 e2 1f e4     l.sw 0xffffffe4(r2),r3
    2284:   9c 60 00 00     l.addi r3,r0,0x0
    2288:   d7 e2 1f e0     l.sw 0xffffffe0(r2),r3
    228c:   9c 60 00 00     l.addi r3,r0,0x0
    2290:   d7 e2 1f f8     l.sw 0xfffffff8(r2),r3
    2294:   00 00 00 8c     l.j 24c4 <main+0x274>
    2298:   15 00 00 00     l.nop 0x0
    229c:   84 62 ff f8     l.lwz r3,0xfffffff8(r2)
```

The machine code

```
@0000088c 44004800 15000000 d7e14ffc 9c21fffc
@00000890 9c210004 8521fffc 44004800 15000000
@00000894 d7e117fc 9c410000 9c21ffe0 9c600000
@00000898 d7e21ff4 9c600000 d7e21ff0 9c600000
@0000089c d7e21fec 9c600000 d7e21fe8 9c600000
@000008a0 d7e21fe4 9c600000 d7e21fe0 9c600000
@000008a4 d7e21ff8 0000008c 15000000 8462fff8
@000008a8 9c630001 d7e21ff8 8482fff4 8462fff0
@000008ac e0641800 d7e21ff4 8482fff0 8462ffec
@000008b0 e0641800 d7e21ff0 8482ffec 8462ffe8
@000008b4 e0641800 d7e21fec 8482ffe8 8462ffe4
@000008b8 e0641800 d7e21fe8 8482ffe4 8462ffe0
@000008bc e0641800 d7e21fe4 8482ffe0 8462fff8
@000008c0 e0641800 d7e21fe0 8462fff8 9c630001
@000008c4 d7e21ff8 8482fff4 8462fff0 e0641800
@000008c8 d7e21ff4 8482fff0 8462ffec e0641800
@000008cc d7e21ff0 8482ffec 8462ffe8 e0641800
@000008d0 d7e21fec 8482ffe8 8462ffe4 e0641800
@000008d4 d7e21fe8 8482ffe4 8462ffe0 e0641800
@000008d8 d7e21fe4 8482ffe0 8462fff8 e0641800
@000008dc d7e21fe0 8462fff8 9c630001 d7e21ff8
```

Figure 3.5: An example of the generated assembly code and machine code.

The first column of the assembly code is the address of each instruction, and the first column of the machine code is the number of each instruction. Because each instruction takes four memory spaces (the instruction is 32 bit, 4 bytes), the ratio between them is four. They are both displayed in hexadecimal. For example, the first instruction of

"main" is d7e117fc whose address is 2250 in the memory and it is the 894th instruction in the machine code.

Thirdly, the ModelSim simulator is used to simulate the different instructions and programs. For different tests, we need to generate all of these files and load the final input.vmem file into the modelled RAM file (more specifically, the RAM file is "ram_wb_b3.v"). After the simulation, all of the signal switching information is stored into a VCD file.

Lastly the Netlist and VCD file are used by the Synopsys Primetime simulator to analyze the power for each test.

3.4 Power Analysis of Basic Test

3.4.1 Design Of The Tests

As all the models showed in Chapter 2, one of the most significant components is the base power/energy cost of each instruction. The method to test the base power cost is to run the test instructions in an infinite loop. This method has already been discussed in Section 2.1. However, on top of this, there are three factors should be taken into consideration together.

In order to measure the base power cost, the first factor is how big the loop size should be. The loop size may be either too big or too small. If the loop size is too big, it may be bigger than the cache size and cause some cache misses. On the other hand, if the loop size is too small, it may reduce accuracy.

The second factor is the cache miss, and this should be considered for measuring the effect of cache misses. The reason is the cache misses may affect the power of the processor and also the runtime of a program. However, this effect was neglected in some of the previous work. A lot of models did not take the cache miss effect into consideration at all or did not show a clear method to calculate this effect.

Considering these two factors together, for measuring the base power cost, the size of the loop of the tests is configured as 100 instances of an individual instruction type. The reason is the cache size is 4kB, the loop size has to be less than 4kB. For measuring the effect of cache misses, in order to get cache misses as many as possible, we configured another tests whose loop size is close to 8kB (7912B, double the cache size). It does not have to be exactly 8kB but should be more than 6kB at least. The reason is OpenRISC uses a direct-map cache.

An important thing is that the contents of these two group of tests are the same and the only difference is the size of the loop. For example, the test for measuring the

base power cost of “ADD” is “ADD, ADD.....ADD”, and there are 100 instances. The corresponding test to measure the effect of cache misses is still “ADD, ADD.....ADD”, but there are 1978 instances (one instruction takes four bytes).

The third factor is the switching rate of the operands between two consecutive instructions since it may affect the power; [2,11,34], it is also necessary to evaluate this precisely.

After analyzing the energy and power usage of the OR1200 processor, we modified the design a little because the IO ports consumed a lot of power, at times even more than the core itself. The reason for this was that even in the case of a cache hit, the IO port interfaces were used unnecessarily in logic instruction execution. Thus, preventing IO ports from sending useless data reduces the power consumption significantly. Therefore, we modified the design to filter data, and the processor sends data only when it needs to communicate with memory.

In order to generate the input.vmem file of each test, we created a very simple C program (for(i=1;i<100;i++)i=i+1;) and modified the contents of the resulting input.vmem file. The reason we do this is there are a lot of initialization codes before running the main function in C, such as enable cache. Thus, after the OpenRISC goes into the main function, we modify the input.vmem and make the processor jump to an empty space where we can write our own code. Because we just need the initialization code, the C code is not important and should be simple.

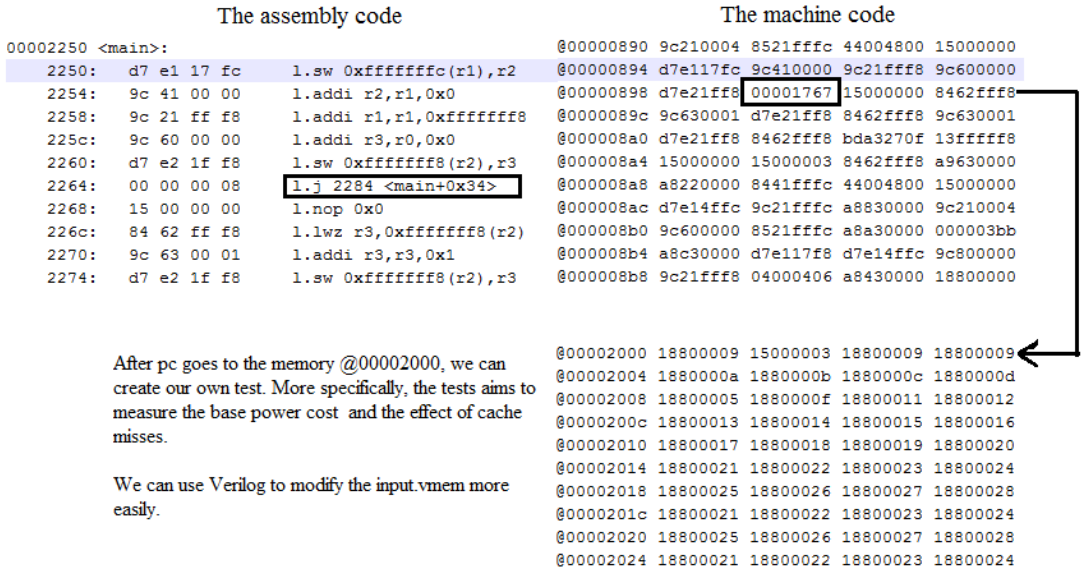


Figure 3.6: The method of creating our own tests.

Figure 3.6 shows an example of how to reuse the code generated from the OpenRISC tool chain. We modified the code “00 00 00 08” to “00 00 17 67” which is an unconditional jump to an empty space (@00002000 in memory). Verilog can be used to modify the

code more easily compared with writing the machine code in input.vmem directly and this is demonstrated in Appendix A.1.2.

Considering the three factors discussed above: the cache can either hit or miss, meanwhile the operand switch rate can either be high or low, there are four cases in each type of instruction. For the low operand switch rate, the Hamming distance of the operands of two consecutive instructions is less than four, but it is more than twelve for a high operand switch rate. The following is an example of the assembly code of the tests.

```
{
// For base power cost, the loop size is 400B (100*4);
// For the effect of cache misses, the loop size is close 8kB (7912B).
// For low operand switch rate, the Hamming distance of r2 and 5, and r3 and
    r6 are both less than 4.
// For high operand switch rate, the Hamming distance of r2 and 5, and r3 and
    r6 are both more than 12.
LOOP: add r1, r2, r3;
      add r4, r5, r6;
      add r1, r2, r3;
      add r4, r5, r6;
      ..............
      add r1, r2, r3;
      add r4, r5, r6;
      J LOOP
}
```

Table 3.1: The opcode and operand of the basic test.

	low switch	high switch
initial	r5:0x3r6: 0xcr7:0xf	r5:0x30aar6: 0x355r7:0xc55
movhi	movhi r7, 0xamovhi r6, 0x5	movhi r7, 0x1555movhi r6, 0x2aaa
add	add r3, r6, r5add r2, r5, r7	add r3, r6, r5add r2, r5, r7
addi	addi r2, r5, 0x5addi r3, r6, 0xf	addi r2, r5, 0x0addi r3, r6, 0x3fff
mul	mul r3, r6, r5mul r2, r5, r7	mul r2, r6, r5mul r3, r5, r7
muli	muli r6, r7, 0x6muli r3, r5, 0xf	muli r6, r7, 0x1fffmuli r3, r5, 0x0
and	and r2, r6, r5,r3, r5, r7	and r2, r6, r5,r3, r5, r7
andi	andi r2, r6, 0xaandi r3, r5, 0x1f	andi r2, r6, 0x39a7andi r3, r5, 0x638
or	or r2, r6, r5r3, r5, r7	or r2, r6, r5r3, r5, r7
ori	ori r2, r6, 0x1fori r3, r5, 0x10	ori r2, r6, 0x38ffori r3, r5, 0x700
xor	xor r2, r6, r5xor r3, r5,r7	xor r2, r6, r5xor r3, r5,r7
xori	xori r2, r6, 0xexori r3, r5, 0x5	xori r2, r6, 0x7f0xori r3, r5, 0x380f
sub	sub r3, r7, r6sub r2, r6,r5	sub r3, r6, r5sub r2, r5, r7
lbs	lbs r2, r6, 0x7lbs r3, r5, 0xa	lbs r2, r6, 0x7lbs r3, r5, 0xa
sb	sb 0x0(r5), r5sb 0xf(r7), r5	sb 0xff(r5), r5sb 0x0(r7), r5
sh	sh 0x5(r5),r6sh 0x7(r7),r5	sh 0x2807(r7),r5sh 0x3007(r6),r6

Table 3.1 shows the opcode and operand for the basic test. In order to have different cache miss rate, the instruction numbers in the loop can be varies and the full details are presented in Appendix A.1.2.

3.4.2 Test Results

Figure 3.7 shows the power consumption measurement of the OR1200. In this figure, "CacheH", "LowS","CacheM","HighS", "Core" and "IO" mean a cache hit, operand low switch, cache misses, operand high switch, the power consumption of the core and the IO respectively. For each group of tests, the first two bars show the average power consumption when the cache hits and the last two bars show when the cache misses. The top part of each column is the IO power consumption and the bottom part is the core.

The test result is from simulation and because there is no randomness included in the model, the measurement results will be always be the same if the test is the same. Therefore, there is no need to run each test more than once and so there is no variation and no margin of error.

For different instructions, the core power consumption is quite similar. For a cache hit, the maximum difference is 15.4% which comes from "and" and "sh" in the high switching case. For a cache miss, the maximum difference is only 6.7% which comes from "xor" and "sh" in the high switching case. The reason for this is that nearly all of the instructions need five pipeline stages. The load/store instructions use the same first three pipeline stages as the arithmetic and logic instructions. On the other hand, no matter what the instruction is, the ALU always performs all the different operations and only chooses a specific one as an output. Therefore, all of the instructions use quite similar hardware, especially the arithmetic and logic instructions. Similar results are also presented in [66, 67, 69].

Data switching can affect the core power, but in a very limited way. For example, the maximum difference is 4.46%, which comes from "addi" with a cache hit. For the high switching case, the core power is slightly higher than for low switching. The reason is that the high switching case has a bigger CMOS signal switching rate than low switching, which will mean more CMOS dynamic power. On the other hand, the average number of switching bits of a register is about seven [11]. Therefore, the average power should be not affected by the data Hamming distance and the power found for "low switching" can be used as the standard power consumption of the core in the model.

A cache miss does not affect the core power very much either. For example, the maximum difference is 10.7%, which comes from "and" in the high switching group. Therefore, when analysing the core power consumption, it is not necessary to consider the difference between a cache hit and a cache miss. The effect of cache misses can be modelled at

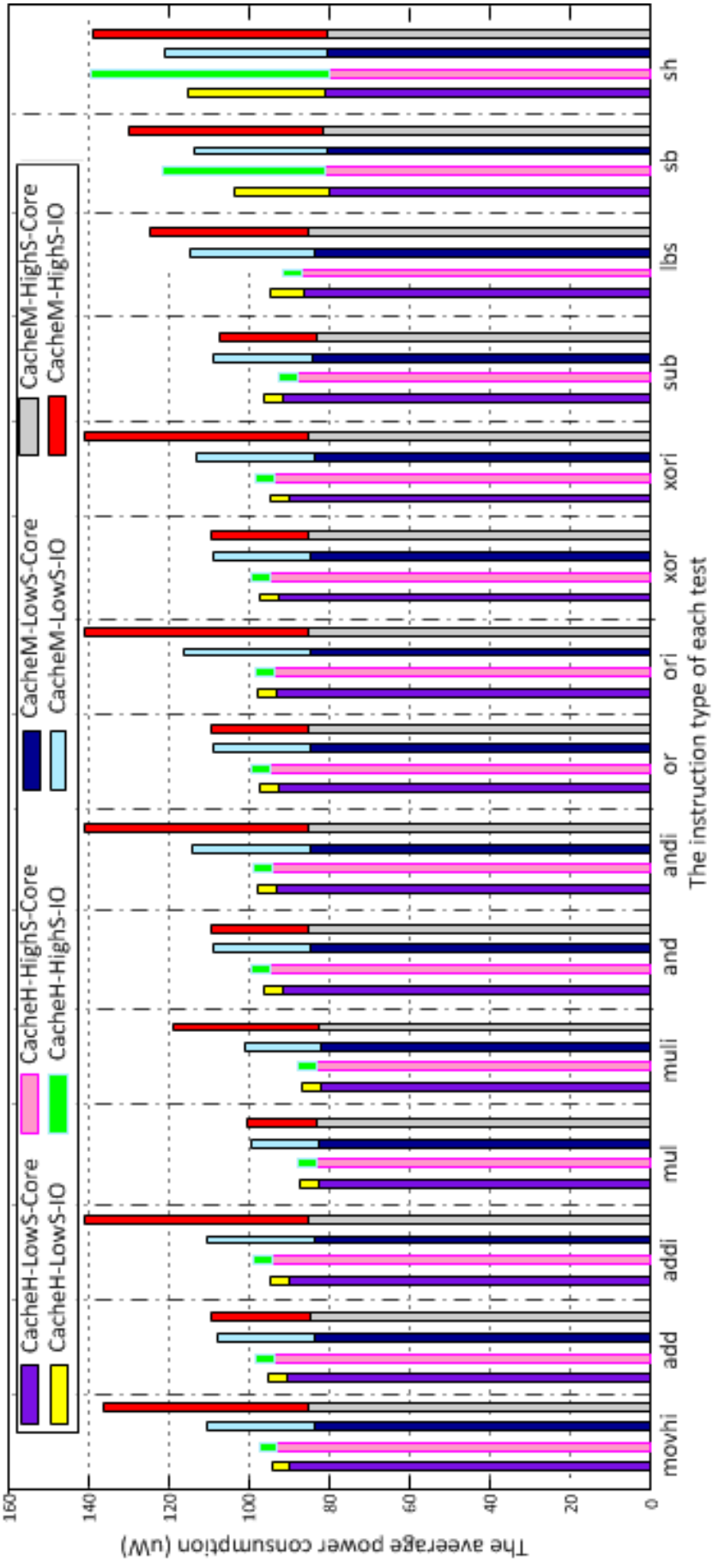


Figure 3.7: The Power consumption of OR1200 with 4kB cache.

runtime, since cache misses make the pipeline stall. Thus, a constant number can be used to estimate the base power of the core. On the other hand, a cache hit happens more frequently than a cache miss. This method makes the model more concise.

In the case of a cache hit, the IO ports consume similar power to the load, the arithmetic, and logic instructions. The data switching rate of the operands of two adjacent instructions does not affect the power much, and the maximum difference is 4.46%. However, for a store, the IO consumes a lot more power. For example, 42.6% of the energy cost of "sh" in the high switching case is from IO. The reason is that the cache is a write-though cache without a write buffer. Thus, the write-though cache will write data to main memory and consume more power whenever store instructions execute.

For the cache miss case, the IO port's power is considerably affected by data switching, especially for immediate addressing mode instructions. The reason is that two consecutive immediate operands may have a large Hamming distance, while the range of possible indirect operands is smaller.

Basically, the IO ports consume more power only when communication happens between processor and memory. This will only occur with a cache miss, or execution of store instructions. Therefore, our hypothesis is that the IO power consumption is related to the cache miss rate and store instruction percentage. A load does not contribute because if the data cache hits, the data asked for by the load comes from the data cache but not from the main memory. Thus, it will not communicate with main memory and consume more IO power.

From the analysis above, we can draw two key conclusions: (1) Regardless of which instructions are run, or whether there is a cache hit or miss, the core of the processor consumes similar power. (2) The IO ports consume a lot of power, but only when communication happens between processor and memory. This will only occur following a cache miss, or during execution of a store instruction. However, these two observations cannot be guaranteed to apply to other processors and need to be checked. For example, when a cache misses happen, the power consumption is related to the cache miss time penalty, which will be discussed in Section 5.3.

3.5 Instruction Level Modeling

Sometimes, we are more interested in the energy consumed by a chip rather than the power. Thus, the following section defines an energy model based on the instruction

level power analysis. The model is described by Equation 3.4.

$$\begin{aligned} E &= \int_0^T P(t) \times dt \\ E &= \bar{P} \times T \\ &= (\bar{P}_{core} + \bar{P}_{io}) \times (T_{cache\_hit} + T_{cache\_miss\_penalty}), \end{aligned} \tag{3.4}$$

where $\bar{P}$ is the average power, T is the total run time. $\bar{P}_{core}$ is the average power of the core, which can be considered as a constant, based on the analysis above. $\bar{P}_{io}$ is the average power of the IO ports, which is related to the cache miss rate and store instructions. Therefore, the following equation can be stated:

$$\bar{P}_{io} = \bar{P}_{io}(cache\_miss\_rate, st\_rate), \tag{3.5}$$

T_{cache_hit} is the time taken when there is a cache hit for the instruction:

$$T_{cache\_hit} = \sum_{i=1}^{4} N_i \times T_i, \tag{3.6}$$

where based on the function and the runtime of each instruction, instructions are divided into four groups: MAC (data calculation, five clock cycles to finish), ALU (data calculation, one clock cycle to finish), load (load data from memory, two clock cycles to finish), and store (store data, six clock cycles to finish). N_i is the number of instructions in each group (MAC, ALU, load and store) and T_i is the timing for each group.

$T_{cache\_miss\_penalty}$ is the time penalty for a cache miss, equation (3.7).

$$\begin{aligned} T_{cache\_miss\_penalty} &= R_{cache\_miss} \times N_{total} \times T_{penalty}, \\ &= R_{cache\_miss} \times \sum_{i=1}^{4} N_i \times T_{penalty}, \end{aligned} \tag{3.7}$$

where R_{cache_miss} is the cache miss rate for the whole program, N_{total} is the number of the instructions of the program, N_i is the number of instructions in each group (MAC, ALU, load and store), and $T_{penalty}$ is the timing penalty.

Based on the analysis above, from Equation 3.4 to Equation 3.7, we derive the following equation:

$$\begin{aligned} E =& (\bar{P}_{core} + \bar{P}_{io}(cache\_miss\_rate, ld\_st\_rate)) \times \\ & (\sum_i N_i \times T_i + R_{cache\_miss} \times \sum_{i=1}^{4} N_i \times T_{penalty}). \end{aligned} \tag{3.8}$$

3.6 Estimation And Analysis

3.6.1 Design of the tests

In order to analyze how the cache and store instruction affect the energy usage, we synthesized two different versions of OpenRISC: (1) 512B instruction cache and 512B data cache. (2) 4kB instruction cache and 4kB data cache. To provide a more realistic program environment, we decide to analyse full length programs. The main bodies of the tests are loops. The components of each test are divided into three cases based on the percentage of store instructions: low percentage (about 5%), mid percentage (about 10%), and high percentage (about 25%). On the other hand, for 512B cache, we can set the cache miss rate of the tests by changing the loop size of each test and create three groups of tests: small (nearly zero), mid (about 30%), and big (about 50%). Thus, there are three different tests (the percentage of store is low, mid, and high) in each group (cache miss rate is low, mid, and high).

Moreover, we designed another two simple programs to enrich our test coverage: a "3*3 matrix times another 3*3 matrix", and a "(for(a=1;a<500;a++)a=a+1;)". The first program is considered as a fourth group since it has MAC instructions that the other tests do not have. The second program is put into the first group (low cache miss rate group), because the cache miss rate is low (0.2% and cache misses only happen when the loop is first run). Consequently, eleven different tests (four from the first group, three from the second group, three from the third group, and one from the last group) were designed.

The method used to create these tests is similar to the base power cost tests in Section 3.4. We only need to modify the contents of the loop in the input.vmem file directly. The following is an example of the program which has a low percentage of store instructions, including input.c, input.asm and input.vmem. The size of the loop in input.vmem is controlled by the input.c file, since more instructions will be compiled if there are more instructions in the loop of the input.c file. Moreover, the size of the loop will determine the cache miss rate of the 512B cache.

```
int main()// the contents of the program is not important, because we need to
    modify it.
// However, the length of the loop is important (the machine code lines of the
    loop),
// because it affects the cache miss rate of the 512B processor.
int a=0;
int b=0;
......
int g=0;
for (a=0;a<50;a++)
{
```

```
a=1+a;
b=b+a;
c=c+b;
......
f=f+e;
g=g+f;
}
asm volatile("l.nop 0x3\n\t");
return a;
```

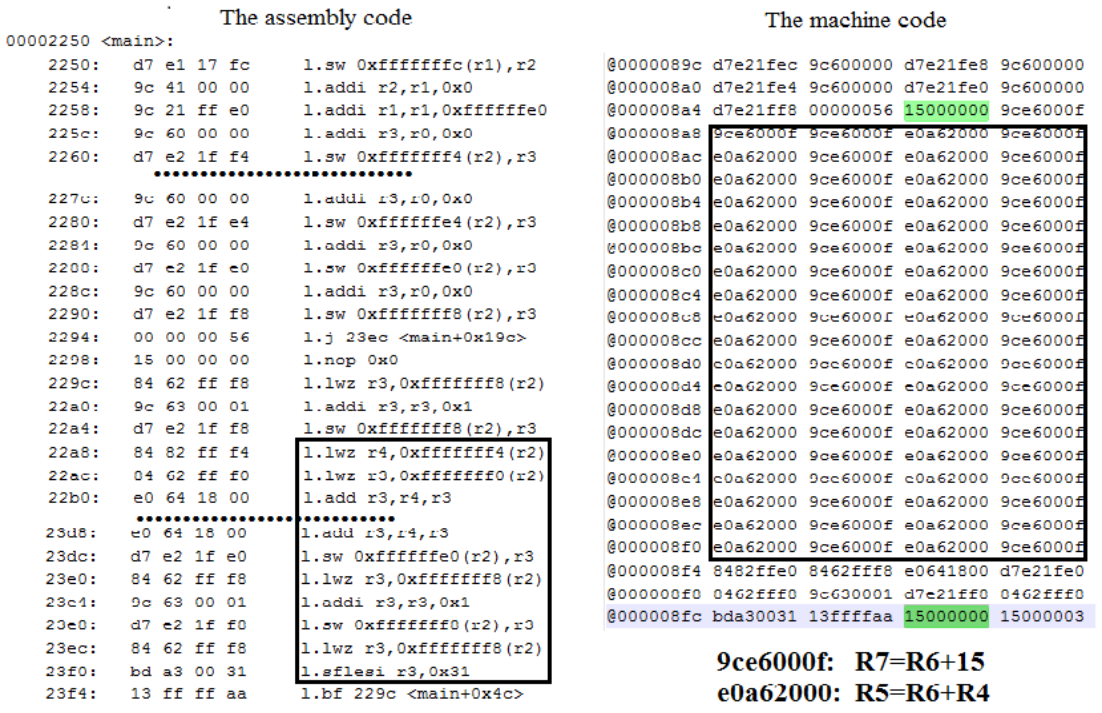


Figure 3.8: An example of the test program.

Figure 3.8 shows the compiled assembly file and modified .vmem file. The modified codes are highlighted. The code of the eleven tests is presented in Appendix A.1.3.

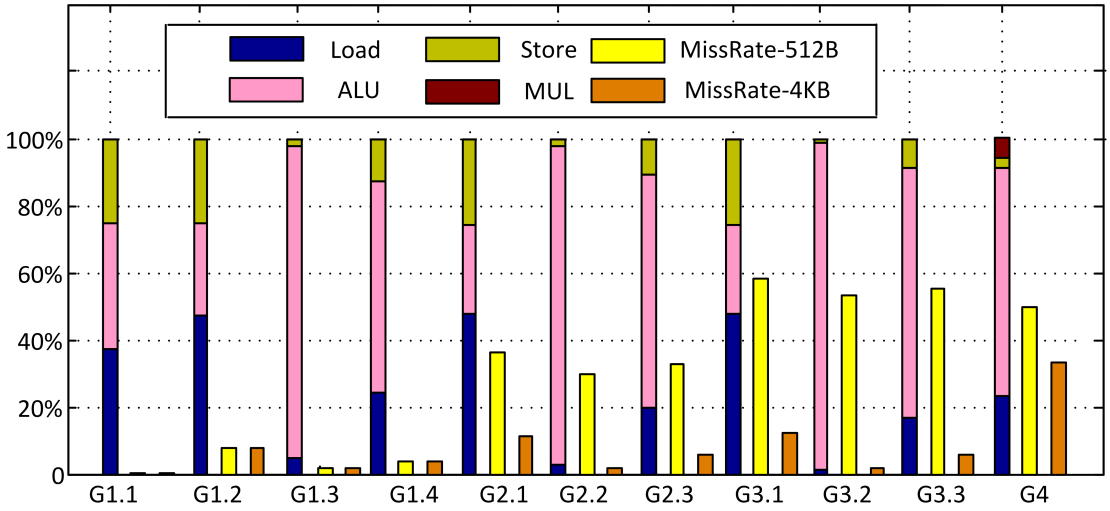


Figure 3.9: The components of each test.

Figure 3.9 shows the components of each test program, in terms of instructions. There are three columns in each test. The first column shows the components of each test including load instructions, ALU instructions, store and multiply instructions. The second and third columns shows the cache miss rate for the 512B cache and 4kB cache respectively.

As in the discussion above, the tests are divided into four groups. We do not include the *branch* and *NOP* components because branches only appear at the end of the program.

In the first group, there are four tests and the test loop size is short enough to be stored in the instruction cache (both 512B cache and 4kB cache) completely. In the second group, there are three tests and the loop size is a little bigger than 512B but smaller than 4KB, which means cache misses occur for the 512B cache but does not for 4kB cache. In the third group, there are three tests and the loop size is bigger than in the second group, but smaller than 1kB. Thus the cache miss rate of 512B cache is higher than before but cache hits still occur. For the 4kB cache, the cache miss rate is low since cache misses only happen when the loop runs for the first time. Group four consists of a single program to validate the instruction level energy model, and is based on matrix multiplication.

Test G1.1 in group one is a simple looping C program. Tests G1.2, G2.1 and G3.1, in groups 1, 2 and 3 respectively, have a higher percentage of store instructions compared with other tests in the same group. For tests G1.3, G2.2, and G3.2, the main components of the loop are logic instructions. Tests G1.4, G2.3 and G3.3 have a balance of store, and logic instructions. On top of this, all of instructions are distributed evenly and the codes of each test are presented in Appendix A.1.3. Although we have proved that the effect of the Hamming distance between operands is small, less than 4.46% in Section 3.4.2, we still need to set this value to a reasonable number. Considering the behaviour of

a program, low bit switching would be a better than high switching. For example, Montserrat etc. studied the Hamming distance for a VLIW processor (TI TMS320C6x) and showed that most of the Hamming distance between instructions can be assumed to be less than 10 [83]. On the other hand, Penolazzi *et al*. found that the average number of switching register bits is seven [11]. Moreover, we also tested the register switching bits for five different benchmarks and came to a similar conclusion, as described in Section 3.7, below.

On the other hand, in Section 3.4 we also proved that the operand switching rate only significantly affected the IO power. This register switching rate will be tested in Section 3.7. There are many other factors that can affect the IO power, such as the cache miss rate. Thus, for convenience, the effect of the operands can be neglected.

3.6.2 Test Result

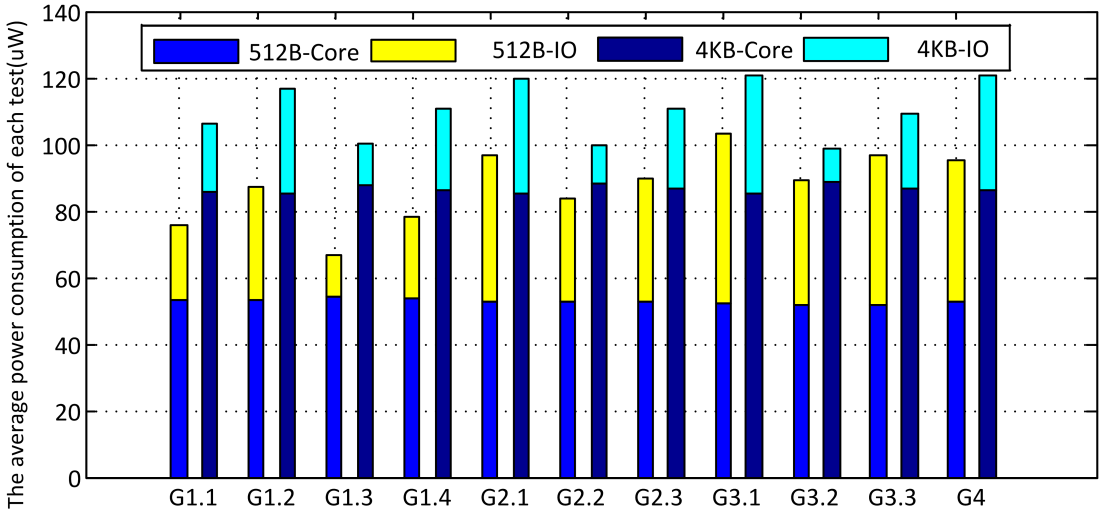


Figure 3.10: The power consumption of each test.

Figure 3.10 shows the power usage of each test program. The first bar and second bar of each group is the power consumption for the 512B cache and 4kB cache, respectively. The top part of each bar is the IO power consumption and the bottom part is the core power consumption. These simulation results do not include any randomness, so each result is measured once, and there is no margin of error. From Figures 3.9 and 3.10, the following points can be observed.

As hypothesized above, the core power of different programs is almost constant, and the maximum difference between test programs is 3.1% and 4.2% for the 512B and 4kB caches respectively. The average power is 52.7 μW and 86.7 μW, respectively. We take 52.7 μW and 86.7μW as the estimated power for the core with a 512B and 4kB cache, respectively.

The IO power is related to the proportion of store instructions, as expected. Group 1 shows that the higher the percentage of store instructions is in the program, the more IO power will be consumed. The reason for this is that the cache is a one-way direct mapped cache, which means any time the processor transfers data to memory, it will communicate with both cache and memory via IO ports.

The IO power is also related to the cache miss rate. For the 512B-cache processor in groups two and three, we observe that the lower the cache miss rate, the less IO power is consumed (this can be seen by comparing tests G2.1, G2.2, G2.3 with G3.1, G3.2, G3.3, respectively). Furthermore, for the 4kB processor the cache can store the whole program, which means the processor only experiences cache misses in the first loop iteration, whereas the 512B-cache processor cannot store the whole program. Therefore, it is quite clear that any time a cache miss happens, the 512B-cache processor will communicate with memory via IO ports, and hence will consume more IO power than the 4kB one.

Based on Figure 3.9 and 3.10, we can derive equation 3.9 by linear regression to describe the IO power, with the cache miss rate and store instruction rate.

$$\begin{aligned} Power_{io} =& 13.633 + 48.5273 \times p_{ST} + 48.3817 \times p_{miss} - \\ & 3.0835 \times p_{ST} \times p_{miss}, \end{aligned} \tag{3.9}$$

where the p_{ST} is the percentage of *store* instructions in the whole program and p_{miss} is the average cache miss rate. We also use this equation to estimate the power of the 4kB instruction cache processor.

Equation 3.10 is used to calculate the difference between estimation and measurement as the ratio of the difference and the measured value.

$$\frac{\Delta E}{E} = \frac{E_{estimation} - E}{E}, \tag{3.10}$$

where ΔE and E are the difference between the estimated and the measured values of energy and the measured energy, respectively.

Figure 3.11 shows the difference between the measured results and the model. It shows that the worst estimate comes from test G3.1 which is in error by 8.2% for the 512B cache. For most of the other case, the difference is less than 5%. The minimum difference is only -0.5% and -0.7% for 512B and 4kB, respectively. There are several reason the errors. The first one is we use a constant value for estimating the core power consumption of 512B cache case (52.7 μW) and 4kB cache case (86.7 μW). The second one comes from the IO power estimation. The reason is when cache misses happen, instructions have to be fetched from memory. However the IO switching rate varies according to the Hamming distance of the machine codes of two consecutive missing instructions.

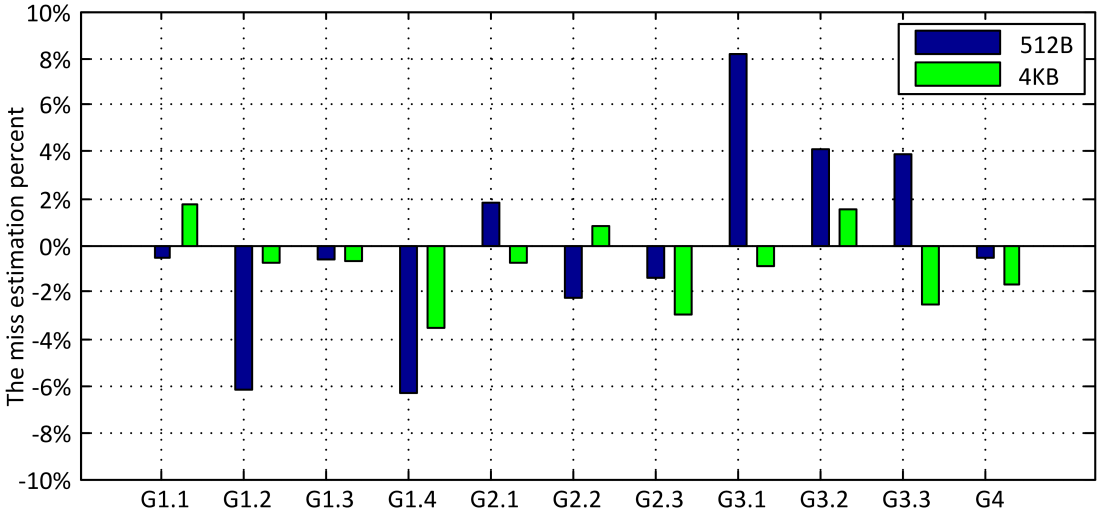


Figure 3.11: Estimation result of basic test.

Therefore, for some tests, the Hamming distance is small, and the IO switch rate is low and consumes less energy. But those tests whose Hamming distance is big consume more power.

3.7 Comparison

Five benchmarks are used to test the performance of the model: Fibonacci, FIR filter, Quicksort, Tak and Tower of Hanoi, and the input.c source codes are presented in Appendix A.1.4. The input values of each test are shown in Table 3.2. We do not use very complex tests or complex input data, because both the Modelsim simulation and Prmietime power analysis are time-consuming. Although these tests are simple, the aim of the OpenRISC experiments is to briefly test our idea: designing an energy model based on the average power consumption and the runtime. On the other hand, more input data just increases the runtime of the test but the instruction types do not change, such as in FIR and Quicksort. Moreover, we have already shown that the data may not affect the power significantly in Section 3.4. Thus, we do not consider the disadvantages of these benchmarks to be very important.

Table 3.2: The input value of each benchmark

Test name	Description of the input
Fibonacci	Generate 15 Fibonacci number
FIR	20 inputs number with 5 coefficients
Quicksort	25 random numbers from 0 to 99
Tak	tak(10,5,3)
Hanoi	five discs

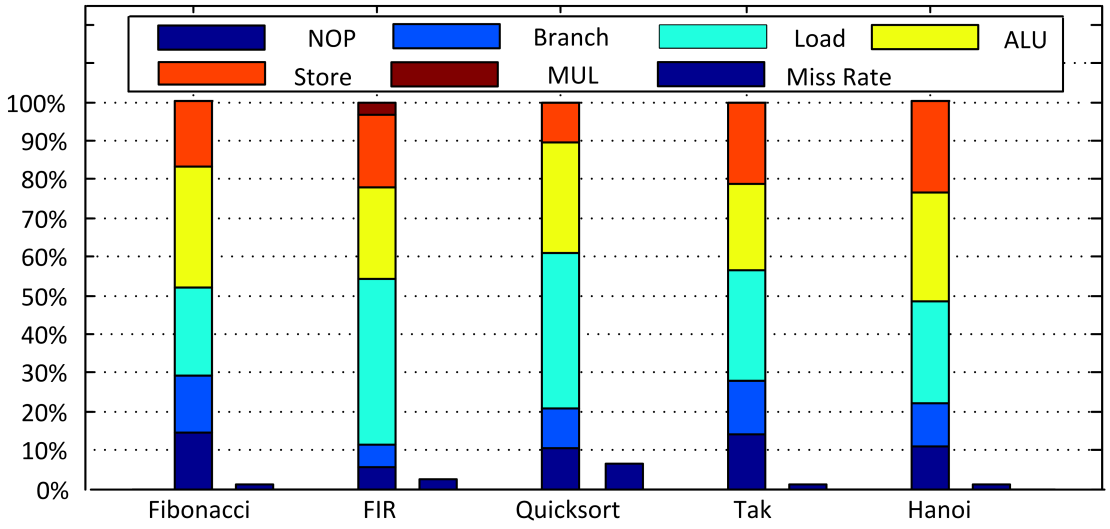
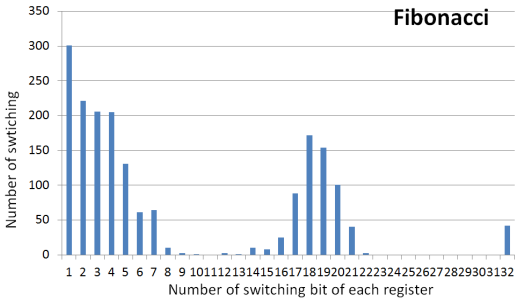
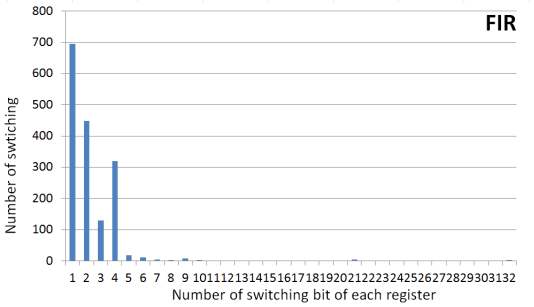


Figure 3.12: The components of each benchmark test program.

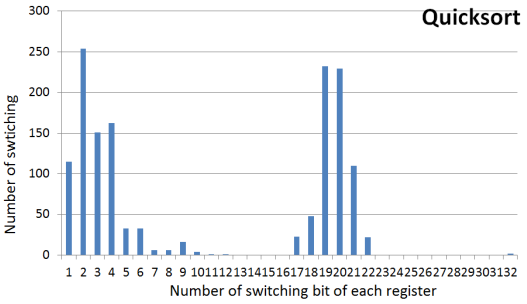
The detail of each benchmark is shown in Figure 3.12. The first bar shows the percentage of NOP, $branch$, $Load$, ALU, $store$, and MUL respectively. The second bar shows the cache miss rate of each test. In this test, we focus on the 4kB cache processor.



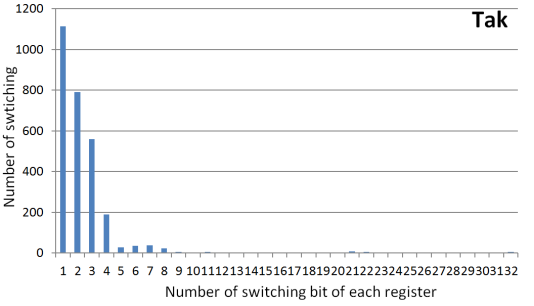
(a) The distribution of the register switching bits in Fibonacci.



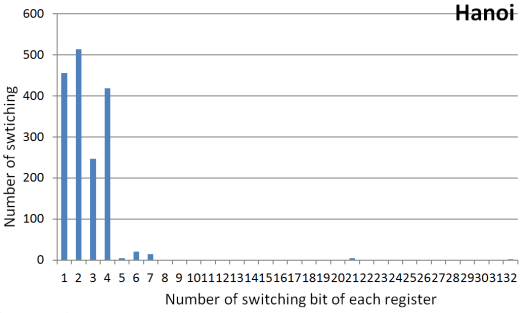
(b) The distribution of the register switching bits in Fir.



(c) The distribution of the register switching bits in Quicksort.



(d) The distribution of the register switching bits in Tak.



(e) The distribution of the register switching bits in Hanoi.

Figure 3.13: The distribution of the register switching bits.

Table 3.3: The average changing bit of each test

	Fibonacci	Tak	FIR	Hanoi	Quicksort
Average changing bit	2.26	2.39	8.78	2.55	10.69

Figure 3.13 and Table 3.3 show the distribution of the register switching bits and the average switching bits of each test, respectively. It is clear that the average number of switching bits in Fibonacci, Tak, and Hanoi is less than three, but the number in FIR and Quicksort is more than seven. The average switching rate of these five benchmark is 5.334. The first reason could be the operands of the tests do not need to have a big change, such as Hanoi. Thus, the register switching bit is low. Secondly, the tests

does not need to have a big jump because of the small instruction code size. Thus, the register which record the address does not need to change much to jump, which results in a small Hamming distance change. Hence, we assume the low switching rate case as the core power consumption is reasonable although the power is not related with it much (less than 4.46%). The relative test details are presented in Appendix A.1.4.

Table 3.4: Estimation results for standard benchmarks

Test	io_m	io_e	io_dif%	P_{core}_m	P_{total}_dif%	T_diff%	Energy diff%
Fibonacci	27.1	22.22	-18.01	86.1	-3.78	-6.80	-10.58
FIR	26.5	23.66	-10.72	86.1	-1.99	-4.01	-6.00
Quicksort	26.1	21.88	-16.17	86.9	-3.91	-7.37	-11.28
Tak	26.2	24.42	-6.79	85.5	-0.52	-6.54	-7.06
Hanoi	25	25.56	2.24	85	2.05	-4.75	-2.69

Table 3.4 shows the results of applying our model to benchmarks. Columns "io_m" and "io_e" show the simulated and estimated IO power in μW, respectively. "io_dif" shows the percentage difference between "io_m" and "io_e". "P_{core}_m" and "P_{total}_dif%" show the measured power consumption of the core and the total power difference between estimation and measurement (including IO and core power). "T_diff%" shows the timing error, and finally, "Energy diff%" shows the total energy difference between estimation and measurement. The test result is from simulation (no random parameters). Thus, we measure each test once, and there is no margin of error. Here, we use 86.7μW as the estimated power consumption of the core, because it is the average of the power consumption in Section 3.6.2.

From Table 3.4, it is clear that the IO power has a maximum error of -18.01% (Fibonacci). Most of the "IO" errors are negative. The reason is in our IO power model, we only take the percentage of *store* instruction and average cache miss rate into consideration. However, we do not consider the the Hamming distance of the machine code of the two missing instructions. In our training test, the Hamming distance is smaller than real tests because a lot of the instructions are similar, such as the example presented in Figure 3.8. However, the power consumption of the core is much bigger than IO: 3.31 times bigger on average. Thus, this prediction is good enough to present an accurate power model of the full processor. For example, the maximum power estimation error of the five benchmarks is -3.91% (Quicksort). On the other hand, the energy estimation is also accurate: the maximum error is -11.28% (also Quicksort).

The timing errors column ("T_diff%") shows the difference between the timing modelled by Equation 3.7 and measurement. The reason for the mismatch is that our timing model is too simple. For example, it ignores all of the data cache misses and pipeline stalls, such as control and data hazards. Moreover, all cache misses assume the same constant timing penalty. On the other hand, for a modern processor, a lot of new technologies are used to avoid cache misses and to try to minimise the timing costs caused by branch instructions. Thus, it becomes harder to create an accurate timing model, and so we will

not model timing further but measure the time in order to generate power and energy models. Therefore, we do not further investigate timing errors.

On top of this, the following calculation can be used to approximate the difference between estimation and the measurement:

$$\begin{aligned}\frac{\Delta E}{E} &= \frac{E_{estimation} - E}{E} \\ &= \frac{(P_{total} + \Delta P_{total}) \times (T + \Delta T) - P_{total} \times T}{P_{total} \times T} \\ &= \frac{P_{total} \times \Delta T + T \times \Delta P_{total} + \Delta P_{total} \times \Delta T}{P_{total} \times T} \\ &= \frac{\Delta P_{total}}{P_{total}} + \frac{\Delta T}{T} + \frac{\Delta P_{total} \times \Delta T}{P_{total} \times T} \\ &\approx \frac{\Delta P_{total}}{P_{total}} + \frac{\Delta T}{T}\end{aligned} \tag{3.11}$$

From Table 3.4, the error in the power estimation is less than the timing error. More specifically, the timing and the average power estimation errors are 5.894% and 2.45% respectively. Thus, the largest part of the energy misprediction comes from the timing model. Improving the timing model or finding a better method to predict the time would make the energy model more accurate.

Table 3.5: Comparison with Previous Work: Energy Estimate Percentage Error

	Our method	[5]	[11].model 1	[11].model 2	[12]
Fibonacci	-10.58%	—	15.58%	9.36%	—
Fir	-6.00%	-4.05%	—	—	11.52%
Quicksort	-11.28%	—	11.41%	3%	—

Our method is compared with previously reported results in Table 3.5. The method reported in [5] tests an ARM7TDMI processor and gives a better estimate because it considers the overhead energy of each instruction pair as an independent factor. In [11], Sandro *et al.* test a SPARC Leon3 processor and create two models. The [11].model 1 does not consider the data dependency but the [11].model 2 takes it into account. For the first model (Fibonacci: 15.58%, Quicksort: 11.41%), our model has a better performance but it is worse than the second model (Fibonacci: 9.36%, Quicksort: 3%). We have a better result than that reported in [12]. Oscar *et al.* tested a PowerPC 603e microprocessor, [12], but they use static analysis method, which can analyse the code and estimate the results fast.

We do not consider the effects of adjacent instructions, and thus save a lot of time in measurement. Table 3.6 shows how many measurements are needed when considering the effect of adjacent instructions. There are 9, 11 and 16 different instructions used in Fibonacci, FIR filter and Quicksort respectively. Moreover, there are an additional 10, 13 and 25 instruction pairs in Fibonacci, FIR filter and Quicksort that need to

Table 3.6: Comparison with previous work: measurement times of the models which consider the overhead energy.

	model considering overhead (The number of different instruction types + the number of different instruction pairs)
Fibonacci	9+10
Fir	11+13
Quicksort	16+25
General	$4325(93^2/2)$

be characterized in the more complex model. However, in our model, the measurement times are proportional to the numbers of instruction types, therefore we need four tests in total. There are 93 different opcodes in the ISA and therefore 4325 different instruction pairs. Another example, which considers the effect of adjacent instructions, shows that there are 49 instructions in the ISA and this needs 1176 tests for a DSP 56K chip [30]. In our model we consider the energy of the cache miss in terms of a timing penalty. The reason is that when a cache miss appears, the program needs more time to finish, and will therefore consume more energy. This approach gives an effective method for considering how cache misses affects the energy consumption, which is not considered in [5], [11] or [12].

3.8 Conclusions

We choose OR1200 as the target processor and present an instruction-level energy model for a single core, in-order RISC processor architecture, in which the effect of cache misses is considered. Firstly, the power in the processor does not change much for different operations and operand switch rates, and is thus considered constant. Several tests based on other processors have found the power of the core is fairly constant for different instructions, for example, the StrongARM [69]. Thus, the method may be applied to other RISC processor architectures. Secondly, the IO port power is related to the percentage of store instructions and the cache miss rate. Using linear regression, an accurate IO port power equation is derived. Instead of analysing the energy of each instruction individually, we use average power and run time to estimate the total energy. Finally, a timing equation considering the cache miss rate is also presented. We demonstrate that our model is almost as accurate as those that consider the effect of adjacent instructions, but that the model can be characterized with significantly less effort.

3.9 Limitation of the work

This work has several limitations:

Firstly, the architecture of the processor is simplistic. A lot of advanced technologies which can improve the performance of a processor are not used, such as branch prediction and instruction pre-fetch. The branch predictor can predict the decision of the branch instruction: taken or not-taken and the target address. Thus, the processor will not stall and keep working before the final result of the branch comes [84, 85], thus this makes the pipeline have a steady flow. The aim of the instruction pre-fetch is similar: improve the performance of the processor, but the idea is to decrease the latency and wait time between the cache and the memory. The method is to pre-fetch the instructions from the lower level memory before they are needed. Therefore, when these instructions are really needed by processor, they are already in the cache or buffer [86–88].

Moreover, the cache of OpenRISC is a direct map, write-though cache, without a write-buffer. It is probably the simplest cache design. However, it affects the performance of the processor. The reason is that a write-though cache is relatively slow compared with a write-back cache when writing data to the memory. Moreover, a direct map cache has a higher cache miss rate than a set associative cache. The write-though cache visits the memory often, so it will consume more power than write-back [89].

Secondly, advanced power saving technologies, such as clock gating, power gating and dynamic voltage scaling, are not used . A clock tree in a CPU consumes a lot of power (estimated as 15%-45% [90, 91] and 34% [92]), because it is the one of the most active networks. Clock gating can save power by adding more logic circuits into the clock tree to stop the flip-flops in them switching when not necessary. Power gating uses similar ideas and the method is to cut off the supply voltage when the block of circuit is not used, [93, 94].

On the other hand, the dynamic power is proportional to the square of the supply voltage. If the power supply for the circuits that are not in the critical path is lower but still meets the requirements of speed, the power will be much lower [95, 96].

Thirdly, the results come from simulation, not a fabricated chip. The result will be affected by the limitations of the simulation.

In order to extend our method, we decided to analyse the behaviour of a real chip, the ARM11.

Chapter 4

ARM11

As discussed in Section 2.6 and 3.9, generating a power/energy model based on the RTL simulation has several disadvantages. Therefore, we extend our previous method to the ARM1176JZF-S and measure the power and energy on a real processor.

4.1 Target Processor

The ARM11 is designed for high performance and low power and is the first implementation of the ARMv6 instruction set architecture. Moreover, it supports a lot of technologies that can improve the performance of the processor, such as dynamic branch prediction [97]. Figure 4.1 shows the pipeline stages of the ARM11; there are eight

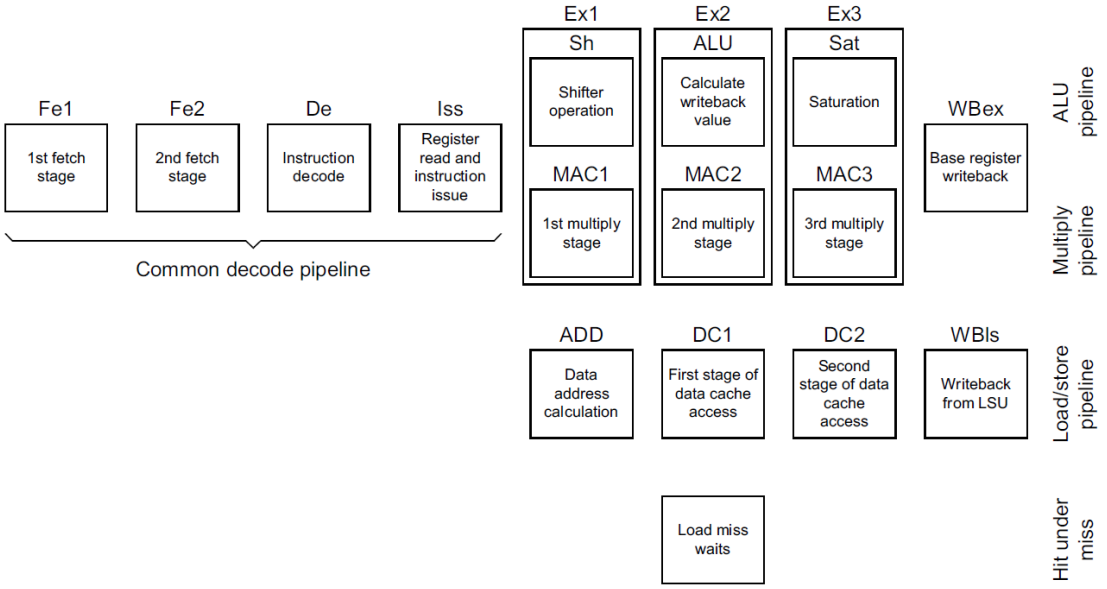


Figure 4.1: Power supply schematic diagram [97].

stages in the ARM11. For the last four stages, ARM11 has three different pipelines for

three different types of instructions. Furthermore, arithmetic and logic instructions use the stages Ex1, Ex2, Ex3 and WBex. Stage MAC1, MAC2, MAC3 and WBex are used by Multiply instructions. Both load and store instructions use the ADD, DC1, DC2 and WBIs stages.

Compared with OpenRISC, the advantages of ARM11 are:

1. Managing instruction branches

   Branching can affect the performance of a processor significantly because the result of the target address is not available until several clock cycles after the branch instruction is fetched. The processor may do some unnecessary work if the processor cannot notice the branch in time. In ARM 11, there are two techniques to solve this problem: a dynamic branch predictor and a branch target address cache (BTAC). The dynamic branch predictor is used to check whether the branch has been fetched before and whether it has a higher chance to be taken or not. The BTAC is used to predict the destination address of the branch [20].

2. Improved memory access

   In order to solve the problem of the speed mismatch between the CPU and memory, a lot of technologies have been developed, such as caches. However, for a simple processor, the cache may only allow instructions that do not need to visit the data cache to be executed during a cache miss. The ARM11 uses a hit-under-miss operation of the memory system and non-blocking cache design. Because of this design, it allows the processor to execute instructions that access the data cache during a cache miss. If three consecutive data misses are encountered, the pipeline will stall [20].

3. Pipeline parallelism

   The ARM11 is an out of order (OoO) scalar processor, which means instructions can finish earlier than a previous instruction if they do not have dependency on the result of previous instructions. For example, because of missing data, the load/store pipeline may stall. However, the consecutive ALU instructions are not related to the missing load/store instructions. Thus, they can be dispatched into the ALU pipeline without waiting. This technology makes the processor usage more efficient.

4.2 Experimental Methodology

A Mini6410 development board with the Samsung S3c6410A embedded processor which uses the ARM1176JZF-S as the CPU was used [98]. It supports dynamic voltage and

Table 4.1: Samsung S3c6410A features.

Technology	65nm
Vdd	1.1V
Frequency	533MHz
Prefetch Unit	uses both static and dynamic branch prediction
Branch target address cache	128-entry
L1Icache	write-though cache with 16kB, 4-way, 2 words per cycle for all requesting sources,
L1Dcache	write-though cache with 16kB, 4-way, 2 words per cycle for all requesting sources

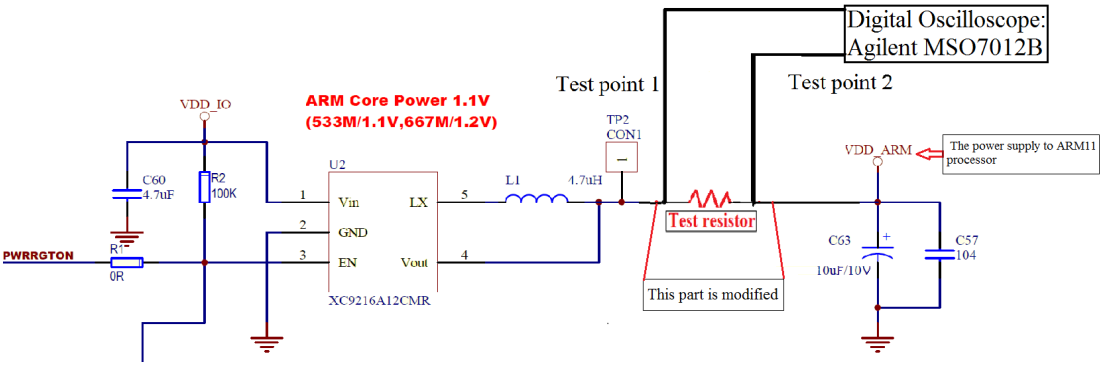


Figure 4.2: The original power supply schematic diagram [100].

frequency scaling (DVFS) and also has interfaces for low power memory. Table 4.1 shows the key parameters of the CPU [99].

Figure 4.2 shows the original power supply of the ARM11 processor and how we modified it. To make the necessary power measurements, a resistor is included between the power supply and the CPU. However, if the resistance is too low, an oscilloscope cannot measure it very accurately. If it is too big, the voltage drop between two sides of the resistor will be too large and the power supply to the CPU will not be enough. Thus, after tests, a 0.51Ω series resistor was chosen. A digitizing oscilloscope, the Agilent MSO7012B, with a sample rate of 2GSa/s was used to measure the instantaneous power as tests were carried out. We used two probes to measure each side of the resistor, $V_1(t)$ and $V_2(t)$. The instantaneous power, average power and the energy are calculated by the following three equations:

$$\begin{aligned} P(t) &= I(t)V(t) \\ &= \frac{V_1(t) - V_2(t)}{R} \times V_2(t) \\ &= \frac{V_1(t) - V_2(t)}{0.51} \times V_2(t), \end{aligned} \tag{4.1}$$

$$
\begin{aligned}
P_{average} &= \int_0^T P(t)dt/T \\
&= \int_0^T \frac{V_1(t) - V_2(t)}{0.51} \times V_2(t)dt/T,
\end{aligned}
\tag{4.2}
$$

$$
E = P_{average} \times T, \tag{4.3}
$$

where $V_1(t)$ and $V_2(t)$ are the instantaneous voltages at test points 1 and 2 in Figure 4.2 respectively. T is the runtime of the program. Linux is used as the operating system. The runtime of the experiments and benchmark applications can be measured directly.

4.3 Basic Power Consumption of Different Instructions

One of the most significant components of a power model is the base power consumption of each instruction. Therefore, we wrote different tests to measure it. The main body of each test is a loop with a number of instances of the same opcode in each loop. In order to avoid cache misses, we chose 8kB (2000 instructions) as the loop size. All of these tests can be fully cached, because both the L1 data and instruction caches are 16kB.

The following is an example of pseudo code which is used to measure the base power cost of AND.

```
while(1);// use a infinite loop to measure the power
{
asm(" AND r3, r2, r1 ");
asm(" AND r4, r1, r2 ");
asm(" AND r3, r2, r1 ");
asm(" AND r4, r1, r2 ");
 ..............
asm(" AND r3, r2, r1 ");
asm(" AND r4, r1, r2 ");
// there are 2000 instructions in the loop
}
```

Table 4.2 shows the operand of each test. ALU stand for any arithmetic logic instructions. In order to distinguish between addressing modes, we have put “i” or “r” at the end of the test name, for immediate or register respectively. In the loop, the opcode is not changed but the operands are changed slightly (the Hamming distance is less than five). Section 4.4 will demonstrate that the affect of the Hamming distance is small, less than 4.65% on average. Thus, the key parameter to effect the power is the opcode. An example of the full test code is presented in Appendix A.2.1.

Table 4.2: The operand of each basic test.

Test	Code	Test	Code
$ALU(i)$	r1: 0x11 r2: 0x5 ALU r3, r2, #0x3f ALU r4, r1 , #0xf3	$ALU(r)$	r1: 0x11 r2: 0x05 ALU r3, r2, r1 ALU r4, r1 , r2

Load	[r5]=0x11
	[r5,#4]=0x5
	ldr r3, [r5]
	ldr r4, [r5, #4]
		Store	r1:0x11
			r2:0x3f
			str r1, [r5]
			str r1, [r5, #4]

Table 4.3: The sample standard deviation (STDEVA) and margin of error (MOE) of the ARM11 basic power test.

	$MOV(i)$	$MOV(r)$	MUL	$ADD(i)$	$ADD(r)$	$AND(i)$	$AND(r)$	$SUB(i)$	$SUB(r)$	
AVEDEV	0.001717	0.00144	0.001918	0.002013	0.002482	0.002367	0.001122	0.011883	0.018067	
MOE(W)	0.001683	0.001412	0.00188	0.001973	0.002432	0.002319	0.001099	0.011646	0.017705	
	$EOR(i)$	$EOR(r)$	$OR(i)$	$OR(r)$	$ASR(i)$	$ASR(r)$	$LSL(i)$	$LSL(r)$	*Load*	*Store*
AVEDEV	0.004786	0.012916	0.001795	0.003221	0.002033	0.013266	0.001243	0.01204	0.003165	0.002235
MOE(W)	0.00469	0.012658	0.001759	0.003157	0.001992	0.013	0.001218	0.011799	0.003101	0.002191

Table 4.3 shows the sample standard deviation (STDEVA) and margin of error (MOE) of the ARM11 basic power test. In order to minimize the effect of the cache, we tested each instruction with four different cache usage patterns (number of instructions in a loop): 1.6kB, 4kB, 8kB and 16kB. We measured the power twice in each case. If there was more than 5% difference, we measured again. The MOE is calculated at a 95% confidence level. For example, the MOE of $MOV(i)$ is $0.001683W$, which means that we can be 95% confident that the power consumption of $MOV(i)$ is the average power of the measurement plus or minus $0.001683W$.

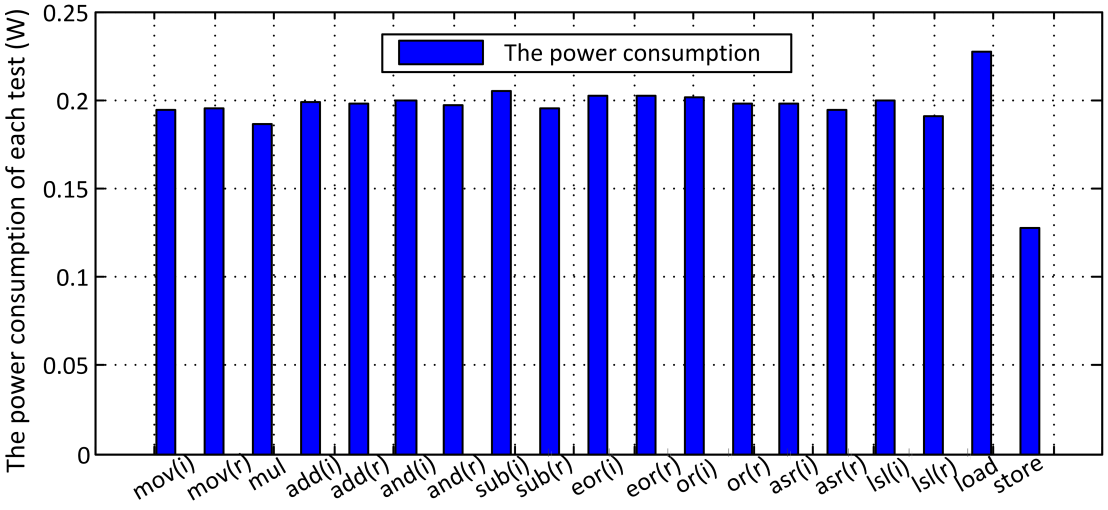


Figure 4.3: The basic power consumption of ARM11.

Figure 4.3 shows the power consumption of arithmetic and logic functions in different addressing modes, multiply, load, and store. The following conclusions can be drawn:

- For different arithmetic and logic instructions, the processor power consumption is similar. The opcode does not affect the power much because all of the arithmetic logic instructions use the same pipeline stage. (See also [67].) For example, the instruction $MOV(i)$ consumes the least power, which is $0.195W$. However, $SUB(i)$ consumes the most power, which is $0.2052W$. Thus, the maximum difference of the arithmetic and logic instructions is 5.45%. The standard deviation (σ=0.00304) divided by the average power ($\overline{P_{ALU}}$=$0.199W$) is 1.53%, so the basic power of different arithmetic logic instructions is very similar.

- The addressing mode does not affect the power very much. For example, the minimum difference is -0.03% (between $EOR(i)$ and $EOR(r)$) and the maximum difference is 5.02% (between $SUB(i)$ and $SUB(r)$).

- Load consumes the most power. There are not any instruction or data cache misses in the load test because all the target operand addresses are the same. Therefore, the load test runs faster and consumes more power than arithmetic/logic functions.

- Store consumes the least power because the instructions per clock cycle (IPC) of store is 0.04. Furthermore, the fact that it takes 25 cycles to finish one store instruction means pipeline stalls happen often. The reason is the cache is a write-though cache, thus when stores execute, some data will be written to the main memory. However, writing data back to the main memory takes a relatively long time and the cache write buffer has to ensure the coherence of the data cache and the main memory. Consequently, although the processor tries to keep writing data to memory, the pipeline may stall and has to wait until the buffer is empty before writing new data. Moreover, the cache write buffer is only 1-2 words in the ARM11 and is easy to fill. Hence, the processor spends most of the time waiting for the cache write buffer and so the power of a store instruction is the lowest.

Based on the analysis above, in order to simplify the model, we assume all arithmetic and logic instructions consume the same base power in all addressing modes since they use similar hardware. In order to test the effect of the different opcodes, the value of r1 and r2 in the code example is set to two and four to reduce the effect of data. On the other hand, the effect of data, such as the Hamming distance of two adjacent instructions, is not important and it will be discussed in Section 4.4.

On the other hand, cache misses can affect the power and speed of a processor. In order to study how cache misses affect the power consumption, we increased the loop body size in different tests and measured how the power changes with the cache miss rate. Figure 4.4 shows those results, where both the L1 data and instruction caches are 16kB and the following conclusions can be drawn:

- When the loop body is larger than 16kB, the instruction cache is not sufficiently large to contain it which causes an increase in the cache miss rate, and hence

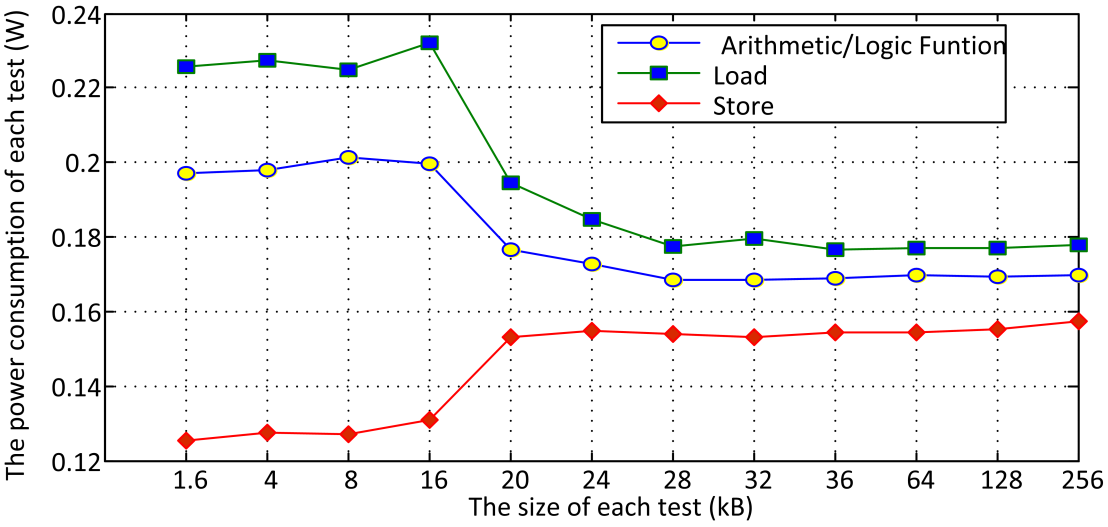


Figure 4.4: Power consumption versus cache miss rate.

arithmetic/logic and load instructions consume less power. This is due to the processor having nothing to do while the instruction fetch unit is reading from the instruction memory. Therefore, the power consumption is lower, but the energy per instruction is higher. As the cache miss rate increases, the processor spends more and more time waiting and the pipeline stalls happen more often.

- The power consumption keeps decreasing until 32kB. The reason is that when the size is bigger than 32kB, most of the instructions of the previous loop have been evicted from the cache when the new loop starts. Thus, most of new instructions have to be fetched from main memory. The lowest speed of the tests is determined by the main memory speed. Thus, even though the loop size increases to more than 32kB, the power does not change. The power for an arithmetic/logic instruction is about $0.170W$ and for load $0.177W$.

- The behaviour of store is quite different. For a cache miss, the power consumption becomes greater than for a cache hit. The reason is that for a cache hit, because of the write buffer, the pipeline stalls often and the IPC is low (0.04). However, for a cache miss, the IPC does not change very much (0.028) and the processor has to fetch new instructions from main memory. Hence, the communication rate with main memory is higher and more data goes though the IO ports. Consequently, a cache miss consumes more power than a cache hit.

4.4 The Power Consumption of Different Hamming Distances

Previous research suggests that the Hamming distance between the operands of two consecutive instructions may affect the power consumption [11,34]. Our third test considers

this on an ARM11 when the L1 cache always hits.

Table 4.4: The opcode and operand of Hamming distance test.

Test	Hamming distance			
	4	8	12	16

ADD(r)	r1: 0x3
	r2: 0xc
	ADD r3, r2, r1
	ADD r4, r1, r2
		r1:0xf
		r2:0xf0
		ADD r3, r2, r1
		ADD r4, r1, r2
			r1:0x3f
			r2:0x3fc0
			ADD r3, r2, r1
			ADD r4, r1, r2;
				r1:0xff
				r2:0xff00
				ADD r3, r2, r1
				ADD r4 , r1 , r2

ADD(i)	r1:0x3
	r2:0xc
	ADD r3, r1, #0x3
	ADD r4, r2, #0xc
		r1:0xf
		r2:0xf0
		ADD r3, r1, #0xf
		ADD r4, r2, #0xf0
			r1:0x3f
			r2:0x3fc0
			ADD r3, r1, #0x3f
			ADD r4, r2, #0xfc0
				r1:oxff
				R2 oxff00
				ADD r3, r1, #0xff
				ADD r4, r2, #0xff00

Load	[r5]=0x3
	[r5,#4]=0xc
	ldr r3, [r5]
	ldr r4, [r5, #4]
		[r5]=0xf
		[r5,#4]=0xf0
		ldr r3, [r5]
		ldr r4, [r5, #4]
			[r5]=0x3f
			[r5,#4]=0x3fc0
			ldr r3, [r5]
			ldr r4, [r5, #4]
				[r5]=0xff
				[r5,#4]=0xff00
				ldr r3, [r5]
				ldr r4, [r5, #4]

Store	r1:0x3 r2:0xc str r3, [r5] str r4, [r5, #4]	r1:0xf0 r2:0xf str r3, [r5] str r4, [r5, #4]	r1:0x3f r2:0x3fc0 str r3, [r5] str r4, [r5, #4]	r1:0xff r2:0xff00 str r3, [r5] str r4, [r5, #4]

Table 4.4 shows the opcode and operand of the Hamming distance test. The Hamming distance between the operands of two consecutive instruction increases from 4 to 16. The main body of this test is still a loop and an example of the full code is present in Appendix A.2.2.

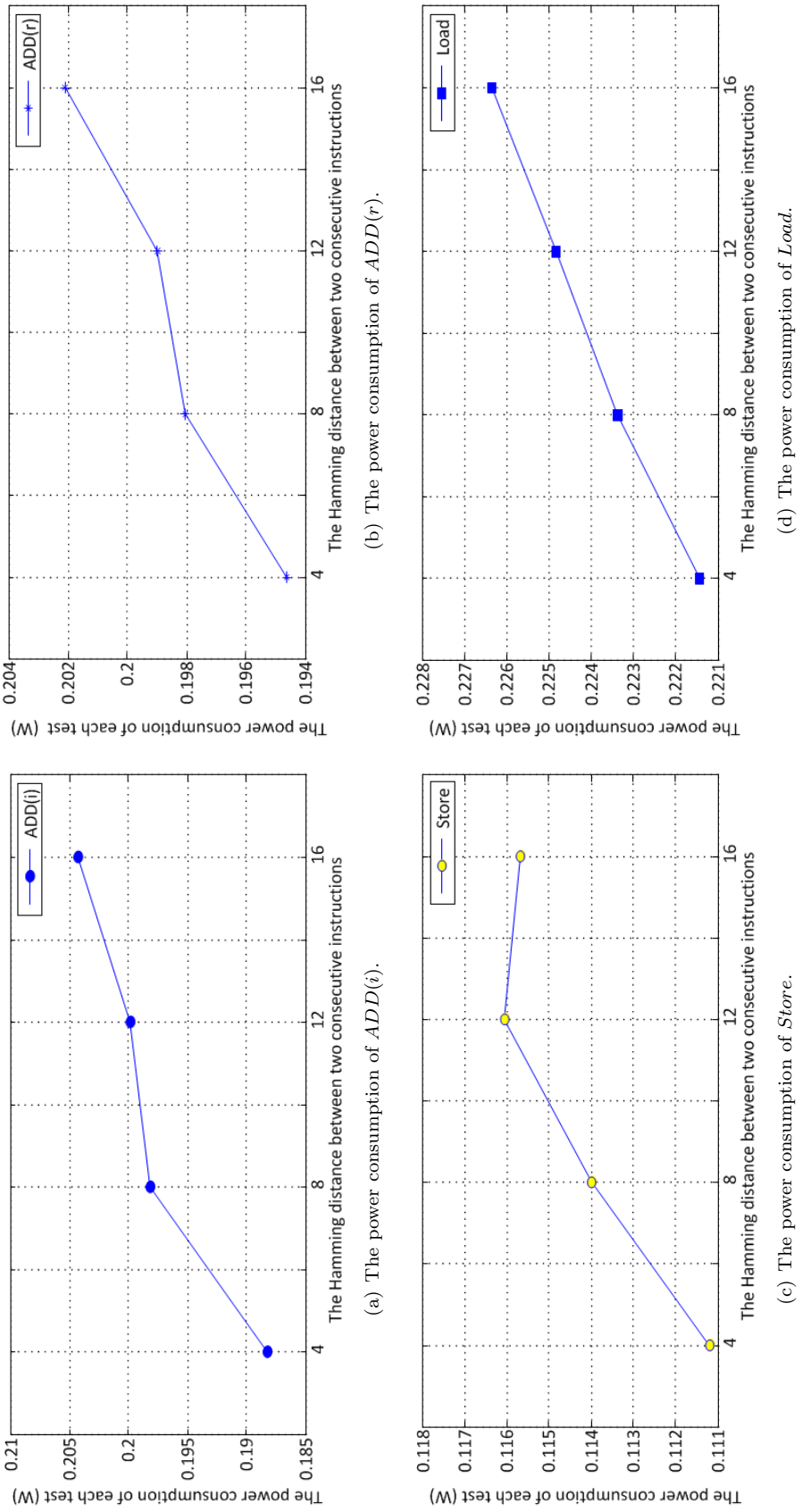
Table 4.5: The sample standard deviation (STDEVA) and margin of error (MOE) of the ARM11 Hamming distance power test.

Bit switches	4		8		12		16	
	STDEVA	MOE(W)	STDEVA	MOE(W)	STDEVA	MOE(W)	STDEVA	MOE(W)
ADD(i)	0.001679	0.002327	0.001122	0.001556	0.001802	0.002039	0001999	0.002771
ADD(r)	4.86E-05	6.74E-05	0.001586	0.001794	0.000235	0.000326	0.002787	0.003863
Load	0.000133	0.000184	0.000124	0.000172	0.003464	0.004801	0.000598	0.000828
Store	0.001819	0.002058	0.002057	0.002328	9.84E-05	0.000136	0.001077	0.001218

Table 4.5 shows the sample standard deviation (STDEVA) and margin of error (MOE) of the ARM11 Hamming distance power test. We measured each test twice. However, if the difference between them was bigger than 5%, we measured a third time. The MOE is calculated at a 95% confidence level. For example, the MOE of $ADD(i)$ is $0.002327W$, which means that we can be 95% confident that the power consumption of $ADD(i)$ is the average power of the measurement plus or minus $0.002327W$.

Figure 4.5 shows the test results, including $ADD(i)$, $ADD(r)$, *Store*, *Load*. In these figures, the X-axis represents the operand Hamming distance of two consecutive instructions and we chose 4, 8 12 and 16 for the target experiments.

From Figure 4.5, we can see that the power consumption is increasing with the Hamming distance of the operand. However, Hamming distance does not affect the power consumption significantly. The maximum difference of each test comes from when the Hamming distance is 4 (the least power) and 16 (the maximum power). For example,



(a) The power consumption of $ADD(i)$.

(b) The power consumption of $ADD(r)$.

(c) The power consumption of *Store*.

(d) The power consumption of *Load*.

Figure 4.5: The power consumption of different Hamming distance.

the maximum difference of $ADD(i)$ is only 8.5%; the maximum differences of $ADD(r)$, *Load* and *Store* are 3.85%, 2.22% and 4.02% respectively. Hence, the maximum and average difference of all tests is only 8.5% (from $ADD(i)$) and 4.65%, respectively.

On top of this, previous research shows the average switching number in a program is about seven [11]. Therefore, in order to have a concise model, we do not put any Hamming distance variables into our power model.

4.5 The Overhead Power Cost

Previous research suggests that the overhead of two consecutive instructions may affect the power consumption [7, 8]. However, there are 51 different instructions in the ARM assembly language set [101] and it would need 1275 ($\frac{51\times50}{2}$) measurements to cover every potential pair. This takes a lot more effort than if we do not consider the overhead power. For example, the NOP model only needs to measure as many examples as the number of instruction types [30]. Thus, in considering every instruction pair, the number of measurements would be about 25 ($\frac{NO.of instructions}{2}$) times more than the NOP model.

However, for a modern processor, we believe that the effect of the overhead power cost is very small for arithmetic and logic instructions, since different instructions share a lot of resources, such as the L1 cache, or the branch predictor. Thus, we assume that the overhead cost will not be significant and our fourth test will pick four instructions to analyse the overhead power cost on an ARM11 when the L1 cache always hits.

To demonstrate the overhead power of arithmetic and logic instructions is not important, we picked four common ALU instructions: $ADD(r)$, $AND(r)$, $SUB(r)$, and $OR(r)$ as examples. The reason for picking these four instructions is ADD(rank 5), AND(rank 6), SUB (rank 7) is that they are the most common logic and arithmetic instructions for the 80x86 [102]. MIPS has similar rankings, based on the mix of instructions for five SPECint2000 programs [102]. The following is an example of the code, which is used to measure the overhead cost of $SUB(r)$ and $ADD(r)$.

```
while(1);
{
asm (" SUB  r3 , r2,  r1");   //1
asm (" AND r4 , r1 ,  r2");  //2
asm (" SUB  r3 , r2,  r1");   //3
asm (" AND r4 , r1 ,  r2");  //4
.....................
}
```

Table 4.6: The opcode and operand of the overhead power test.

Test	Code	Test	Code	Test	Code
$ADD(r)\_AND(r)$	*ADD* r3, r2, r1 *AND* r4, r1, r2	$AND(r)\_SUB(r)$	*AND* r3, r2, r1 *SUB* r4, r1, r2	$SUB(r)\_ORR(r)$	*SUB* r3, r2, r1 *ORR* r4, r1, r2
$ADD(r)\_SUB(r)$	*ADD* r3, r2, r1 *SUB* r4, r1, r2	$ADD(r)\_ORR(r)$	*ADD* r3, r2, r1 *ORR* r4, r1, r2		
$AND(r)\_ORR(r)$	*AND* r3, r2, r1 *ORR* r4, r1, r2				

Table 4.6 shows the opcode and operand of the overhead tests. In order to minimize the effect of the operands, we set both $r1$ and $r2$ to be 0. The main body of this test is still a loop and an example of the full code is presented in Appendix A.2.3.

Table 4.7: The sample standard deviation (STDEVA) and margin of error (MOE) of each instruction.

	$ADD(r)$	$AND(r)$	$OR(r)$	$SUB(r)$
STDEVA	0.001881	0.001145	0.002388	0.000617
MOE(W)	0.002128	0.001295	0.002702	0.000698

Table 4.8: The sample standard deviation (STDEVA) and margin of error (MOE) of instruction pairs.

	$AND(r)$		$OR(r)$		$SUB(r)$	
	STDEVA	ME(W)	STDEVA	ME(W)	STDEVA	ME(W)
$ADD(r)$	0.001595	0.002210	0.001663	0.002305	0.000662	0.000917
$AND(r)$			0.002208	0.00306	0.001509	0.002092
$ORR(r)$					0.002355	0.003264

As in Section 4.5, we measured each test twice unless the difference was bigger than 5%. Table 4.7 and Table 4.8 shows the sample standard deviation (STDEVA) and the margin of error (MOE) of each instruction and and of each instruction pair, respectively. The MOE of each table is calculated at the 95% confidence level.

Table 4.9: The power consumption of each instruction(W).

$ADD(r)$	$AND(r)$	$SUB(r)$	$OR(r)$
0.18815	0.18879	0.19716	0.18774

Table 4.10: The average power consumption of each pair(W).

	$AND(r)$	$SUB(r)$	$OR(r)$
$ADD(r)$	0.18740	0.19136	0.18687
$AND(r)$		0.19276	0.18827
$SUB(r)$			0.19223

Table 4.9 and Table 4.10 show the power consumption of each instruction and the average power consumption of each pair, respectively. Because the base power consumption of each instruction is similar, the average power of each pair is also similar.

Table 4.11: The measured power consumption of each pair(W).

	$AND(r)$	$SUB(r)$	$OR(r)$
$ADD(r)$	0.19304	0.20053	0.18868
$AND(r)$		0.19907	0.18956
$SUB(r)$			0.20233

Table 4.12: The overhead power and the difference ratio(W) .

	$AND(r)$	$SUB(r)$	$OR(r)$
$ADD(r)$	0.0056 (2.92%)	0.00917 (4.57%)	0.00180 (0.96%)
$AND(r)$		0.00631 (3.17%)	0.00129 (0.68%)
$SUB(r)$			0.01010 (4.99%)

Table 4.11 shows the measured power consumption of each pair, and Table 4.12 shows the overhead power and the different ratio between the overhead and the average power consumption. The first column is the first instruction of each pair and the first row is the second instruction of the pair. Based on the definition of the overhead power cost proposed by Tiwari [8], the sequence of the instructions inside of the instruction pairs will not affect the overhead power/energy. The overhead power cost of $Instruction_1$ and $Instruction_2$ is the same as $Instruction_2$ and $Instruction_1$ [8]. The reason is that it is hard to distinguish whether $Instruction_1$ is after $Instruction_2$ or $Instruction_2$ is after $Instruction_1$ when you run a sequence of instructions "$Instruction_1$, $Instruction_2$, $Instruction_1$, $Instruction_2$, $Instruction_1$, $Instruction_2$..." to measure the overhead power. Eventually, both of these two cases exist in this sequence and are considered to be equal for convenience. Thus Table 4.11 and Table 4.12 are triangular matrixes.

It is clear that the overhead power cost is positive but very small, which is expected. For example, the minimum ratio is 0.68%, which is from the instruction pair AND and ORR. The maximum ratio is 4.99%, which is from $OR(r)$ and $SUB(r)$.

Based on these test results, we do not need to consider the overhead cost of different pairs from the same instruction class since this effect is small but it would need a lot of effort to measure every possibility.

4.6 Instruction level power analysis and modeling

In order to study how the different instructions and IPC affect the power in combination, such as the overhead power between different classes, a more realistic program environment was created. The instructions are grouped into three classes: ALU logic, load and store. The reason is that the function and power consumption of classes ALU, load, and store are very different. Furthermore, ALU instructions focus on calculation, load focuses on reading data from memory, and store focuses on saving data to the memory. Thus, these three classes of instructions use different hardware of the processor and consume different power, which is shown in Figure 4.3. Chapter 7 shows how to cluster instructions into different classes in more details. The components of the program are coded manually to allow understanding of the distribution of the program in detail and to change it easily. The following is an example of pseudo code and the different components are evenly distributed in the program.

```
while(i<0xFFFF);
{
asm(" AND r3, r2, r1 ");
asm(" ADD r4, r2, r1 ");
asm (" STR r1, [r5]");
asm (" LDR r1, [r5]");
 ..............
asm(" AND r3, r2, r1 ");
asm(" ADD r4, r2, r1 ");
asm (" STR r1, [r5]");
asm (" LDR r1, [r5]");
// 25% are STR, 25% are LDR and 50% are Logic
}
```

Table 4.13 shows the test opcodes and operands in detail. R1 and R2 are set to 0x11 and 0x5, respectively. For each test, N is set to 1000, 2000, 3000, 4000, which mean the cache usage is from 25% to 100% and the final test power is the average of each. In each test, we try to use as many different instructions as possible, which can minimise the effect of one single instruction. An example of the full code is presented in Appendix A.2.4.

We choose 25% as a step size and the percentage of the arithmetic logic instructions decreases from test 1 to test 9. For example, all of the instructions in test 1 are arithmetic/logic but only 25% of instructions come from arithmetic/logic in test 7, test 8 and test 9. In contrast, the load and store instruction percentages increase. Finally, Table 4.13 shows all of the possibilities. The advantage is that we can take the effect of the instructions between different classes, such as overhead power, into consideration without involving extra tests. Furthermore, we do not need measure all of the possible

Table 4.13: The opcode and operand of the tests for modeling.

Test	Code	Test	Code	Test	Code

test1	ADD r3, r2, r1
	ORR r4, r1, r2
	SUB r6, r1, #0xf
	EOR r7, r2, #0xf3
	ADD r3, r2, r1
	AND r4, r1, r2
	MOV r6, r1
	ORR r4, r1, r2
	...
	repeat N times
		Test2	LDR r3, [r5, #4]
			ORR r4, r1, r2
			SUB r6, r1, #0xf
			EOR r7, r2, #0xf3
			LDR r3, [r5,#8]
			ADD r4, r1, #0xf
			MOV r6, r1
			AND r3, r1, r2
			...
			repeat N times
				Test3	STR r3, [r5, #4]
					ORR r4, r1, r2
					SUB r6, r1, #0xf
					EOR r7, r2, #0xf3
					STR r2, [r5,#8]
					ADD r4, r1, #0xf
					MOV r6, r1
					AND r3, r1, r2
					...
					repeat N times

Test4	LDR r3, [r5, #20]
	ORR r4, r1, r2
	LDR r6, [r5, #0x4]
	EOR r7, r2, #0xf3
	LDR r3, [r5,#8]
	ADD r4, r1, #0xf
	LDR r6, [r5, #16]
	AND r3, r1, r2
	...
	repeat N times
		Test5	LDR r3, [r5, #20]
			ORR r4, r1, r2
			STR r1, [r5, #0x4]
			EOR r7, r2, #0xf3
			LDR r3, [r5,#8]
			ADD r4, r1, #0xf
			STR r1, [r5, #16]
			AND r3, r1 , r2
			...
			repeat N times
				Test6	STR r1, [r5, #20]
					ORR r4, r1, r2
					STR r2, [r5, #0x4]
					EOR r7, r2, #0xf3
					STR r1, [r5,#8]
					ADD r4, r1, #0xf
					STR r2, [r5, #0x16]
					AND r3, r1, r2
					...
					repeat N times

Test7	LDR r3, [r5, #20]
	LDR r4, [r5, #8]
	LDR r6, [r5, #0x4]
	EOR r7, r2, #0xf3
	LDR r3, [r5,#8]
	LDR r3, [r5, #12]
	LDR r6, [r5, #16]
	AND r3, r1, r2
	...
	repeat N times
		Test8	LDR r3, [r5, #20]
			STR r3, [r5, #16]
			LDR r6, [r5, #0x4]
			EOR r7, r2, #0xf3
			LDR r3, [r5,#8]
			STR r1, [r5, #12]
			LDR r6, [r5, #16]
			AND r3, r1, r2
			...
			repeat N times
				Test9	LDR r3, [r5, #20]
					STR r1, [r5, #24]
					STR r3, [r5, #16]
					EOR r7, r2, #0xf3
					LDR r3, [r5,#8]
					STR r3, [r5, #12]
					STR r2, [r5, #16]
					AND r3, r1, r2
					...
					repeat N times

different instruction pairs. We ignore the 0% logic case because it is unlikely that a program does not have any logic instructions.

Table 4.14: The sample standard deviation (STDEVA) and margin of error (MOE) of the modelling tests.

	Test1	Test2	Test3	Test4	Test5	Test6	Test7	Test8	Test9
STDEVA	0.001719	0.000688	0.001827	0.002374	0.002354	0.00082	0.002255	0.001978	0.003249
MOE(W)	0.001924	0.000778	0.002068	0.002687	0.002664	0.000928	0.002552	0.002238	0.003677

Table 4.14 shows the sample standard deviation (STDEVA) and margin of error (MOE) of the modelling tests. We measure each test three times unless the difference between any measurements is bigger than 5%. The MOE is calculated at a 95% confidence level.

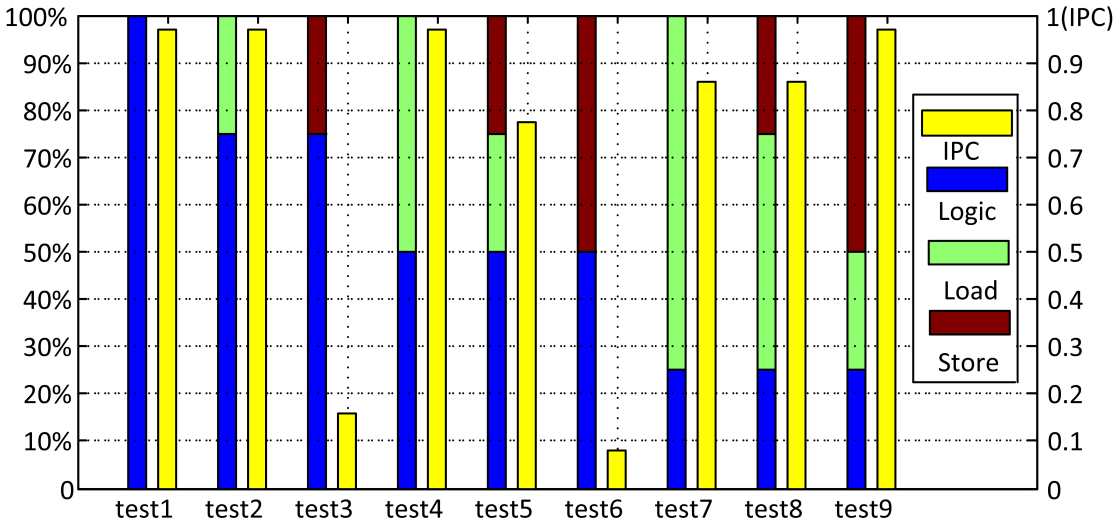


Figure 4.6: Instruction distribution and IPC of each basic test.

Figure 4.6 shows the components and IPC of each test in the first and second columns, respectively. The first column shows the same data as that present in Table 4.13. The second column shows the IPC for each test. There are a lot of different reasons why the pipeline stalls, such as data dependencies and cache misses. But in all cases, the IPC becomes low and the processor has nothing to do but wait. Thus, IPC can be used as a parameter to estimate how smoothly a program runs and to reflect the effect of the cache miss rate and pipeline stall rate.

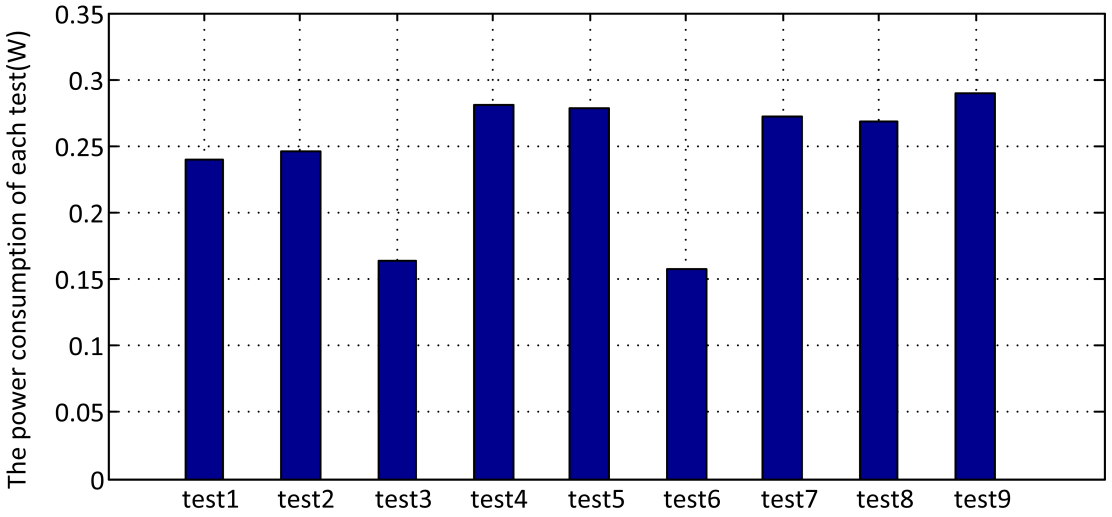


Figure 4.7: The power consumption of each basic test.

Figure 4.7 shows the power consumption of each basic test. Test 3 and test 6 consume the least power because the IPC is the lowest in these tests. Therefore, the pipeline stalls happen more often and the processor has to wait longer than in other tests. Test 9 consumes the most power because the IPC is very high (more than 0.968 for the tests). It means the processor is extremely busy calculating and has few pipeline stalls. For

the tests with similar IPCs, if the test has more logic instructions, it will consume less power, as in test 1 and test 2. Therefore, we use linear regression to generate an average power model to describe how these factors determine the power:

$$\begin{aligned} Power_{average} =& 0.1882 - 0.0601 \times p_{logic} + 0.0081 \times p_{store} \\ &+ 0.1251 \times IPC, \end{aligned} \tag{4.4}$$

where the $Power_{average}$ is the average power consumption of the program, and p_{logic}, p_{store} and IPC are the logic instruction percentage, store percentage and IPC of the program respectively. We assume that all of the instructions come from these three cases. Thus, $p_{logic} + p_{store} + p_{load} = 100\%$, and p_{load} can be presented by p_{store} and p_{logic} after creating the power model by linear regression, such as $p_{load} = 100\% - p_{logic} - p_{store}$. Therefore, only p_{logic} and p_{store} are presented in the model.

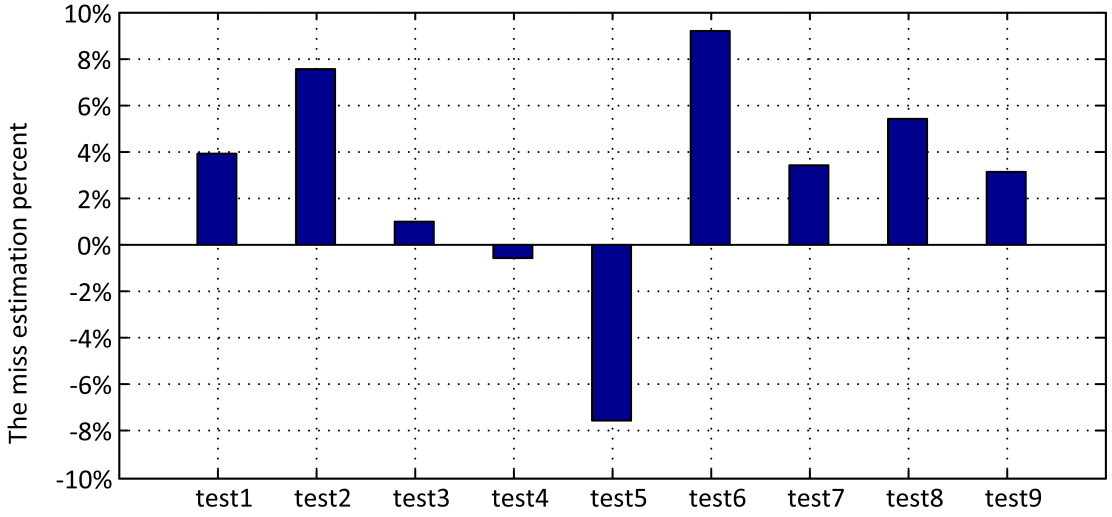


Figure 4.8: The estimation results for each basic test.

Figure 4.8 shows the difference error percentage between the model and the measured results. It is clear that all of the tests are estimated accurately, with errors less than 10%. The reason for the under-estimation of test 5 is that the IPC of test 5 is lower compared with other tests, such as test 1 and test 2. There are several factors which may induce errors. For example, one factor may be measurement since we measure ten times for each test and there are slight differences between each measurement. Thus, we use the average value as the final power and this may induce an error. On the other hand, the effect of the operate system (OS) may be another reason. However, this result also shows that our estimation is accurate. If all of tests are over-estimated or under-estimated, it means something is over considered or less considered, and a constant factor should be added into the model to increase the accuracy.

4.7 Validation

Six benchmarks: Bitcount, Fibonacci, Tak, FIR filter, Quicksort and Tower of Hanoi were used to test the performance of the model. The input values of each test are shown in Table 4.15, and the input.c source code is presented in Appendix A.2.5.

Table 4.15: The input value of each benchmark.

Test name	Description of the input
Bitcount	Default small test from Mibench
Fibonacci	Generate 25 Fibonacci number
Tak	Tak(3000,2,3)
Fir	4000 inputs number with 5 coefficients
Quicksort	4000 data from Mibench\ automotive\qsort\input_large.dat
Hanoi	9 discs

The components of each test are shown in Figure 4.9. The distribution was generated by the instruction simulator tool gem5 [78]. We do not need a cycle-accurate simulation tool and the ARM performance counter also can supply this information. Therefore, the distribution is fast and easy to measure.

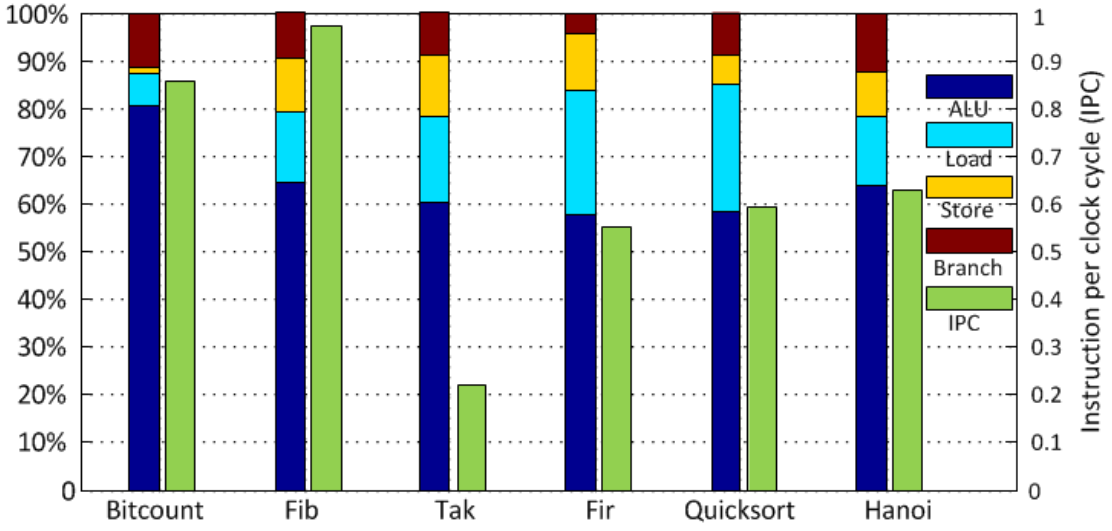


Figure 4.9: The components of each benchmark test program.

Figure 4.9 shows that the lowest IPC is for Tak (0.22), while for Fibonacci it is more than 0.95. The components of the different tests are also very different. For example, the logic percentage of Fir is less than 60% but it is more than 80% in Bitcount.

Table 4.16 shows the sample standard deviation (STDEVA) and margin of error (MOE) of each benchmark. We measure each test twice unless the difference of the two measurements is bigger than 5%. The MOE is calculated at a 95% confidence level.

Table 4.16: The sample standard deviation (STDEVA) and margin of error (MOE) of each benchmark.

	Bitcount	Fib	Tak	Fir	Quicksort	Hanoi
STDEVA	0.00174	0.000854	0.001323	0.001294	0.001639	0.002288
MOEW	0.002411	0.001184	0.001834	0.001793	0.002271	0.003171

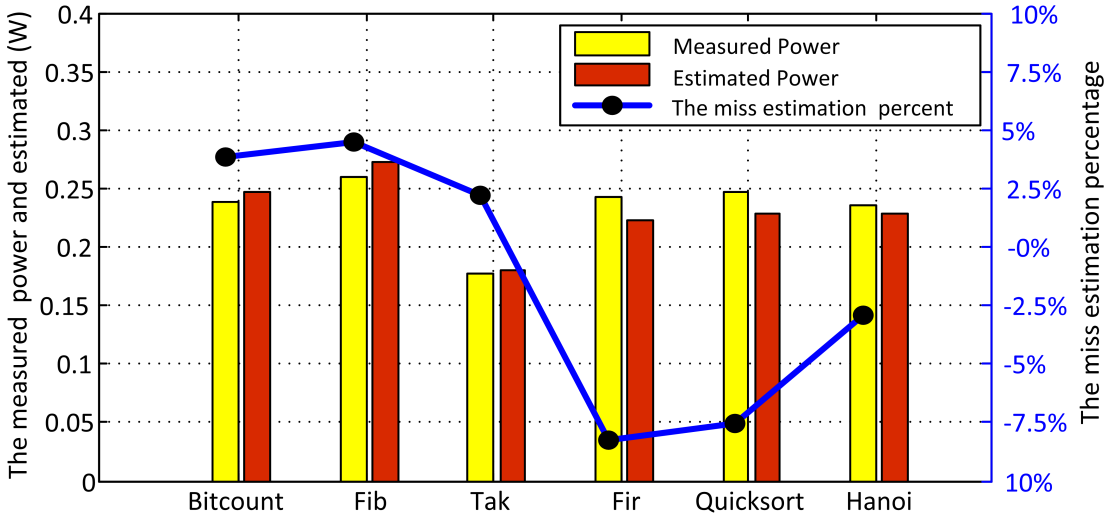


Figure 4.10: The power consumption of measurement and estimation.

Figure 4.10 shows the measured power (column 1), estimated power from our model (column 2) and the difference percentage (curve). Tak consumes the least power, which is 0.177W, because it has the lowest IPC compared with the others. Although Bitcount, Fir, Quicksort and Hanoi have different IPC and opcode percentages, they consume similar power. Moreover, the power estimation is very accurate and the maximum estimation error is less than 9% with an average absolute error of 4.88%. The reason for errors may be because even when different programs have the same IPC, they may still have different hardware usage. For example, one program may have more cache misses but the other may have more branch miss predictions or data dependencies. Thus, different hardware usage means different power and energy.

4.8 Energy model

Sometimes, people are concerned more with the energy usage of a program than the power. However, an energy model is hard to create because there are a lot of factors that affect either the power or the runtime of a program, and all of these factors can affect the energy. For example, the types of instruction or the Hamming distance of two adjacent instructions can affect the power, and pipeline stalls can affect both the runtime and the power.

Moreover, a pipeline stall has many causes, such as cache misses, write buffer limitations, etc. Thus, it is hard to consider every possibility. On the other hand, even when the energy model is created, in order to use it, the target program has to be analysed, which is usually achieved by an ISA simulater, such as gem5. This is because it is hard to get the input data for an energy model without a simulator, such as the cache miss rate and the number of each type of instruction.

Compared with an energy model, an average power model can be generated much more easily and the parameters of the power model are also easier to obtain. Instead of establishing the energy model directly, it is easier to formulate the energy of a program in two steps: 1) create the power model, and 2) measure the runtime. The runtime of the program can be measured easily using the program counter, for example, and simulated by instruction set simulators, such as gem5 [78]. Therefore, the power model can also be extended to model the energy.

In this case study, the runtime of the program is measured from the OS, thus the error in estimating the energy consumption of a program is the the same as the power estimation error. The relationship between power and energy is given by Equation 4.3.

4.8.1 Comparison with Previous Work

Compared with the other models, one benefit of ours is its simplicity because the overhead power of two consecutive instructions is not considered separately. Our method and tests have already analysed the effect of the overhead cost between the instructions. In other words, we do not need to do extra tests or work for measuring the overhead cost. However, if a model considers the overhead energy as an independent factor, the measurement times will be proportional to the square of the number of instructions in the instruction set architecture (ISA). For example, ARM assembly language consists of 51 different instructions [101], so it would need at least 1300 measurements to cover every potential pair.

Another benefit of our model is that it is easy to create. In order to generate the power model, we use just nine training tests to achieve minimum and maximum errors of -0.56% and 9.15% based on six benchmarks, respectively. However, the energy model for the MeP processor requires sophisticated training tests and considers the standard deviations of every parameter value [103]. The minimum and maximum error of that model are 2% and 16%. However, Bazzaz et al. created a model for the AT91SAM7X256 processor which uses the ARM7TDMI as the core [15]. They used 60 specialized tests to estimate the coefficients of each energy sensitive factor. On top of this, there are 35 parameters for the model including: the ARM7 instruction set, register bank bit flip, instruction word Hamming distance etc. [15].

We also consider the effect of cache misses and pipeline stalls to some extent. Furthermore, cache misses and data dependency can make the pipeline stall, hence will affect the power and energy consumption of a program. We take IPC into account in terms of these factors and this approach gives an effective method for analyzing them. However, these factors are not considered much in [5, 11]. For example, Nikolaos *et al.* did not provide a detailed method to estimate the effect of cache misses [5]. Sandro *et al.* did not present a clear full energy model and only presented several parts of models, such as 1-instruction-based and 2-instruction-based models [11].

Table 4.17: Comparison with previous work.

	Our method	[5]	[11].model 1	[11].model 2	[12]	[104]	[15]
Fibonacci	4.47%	—	15.58%	9.36%	—	—	—
Fir	8.29%	-4.05%	—	—	11.52%	—	—
Quicksort	-7.51%	—	11.41%	3%	—	8.98%	—
Bitcount	3.8%	—	—	—	—	—	-4.22%

Table 4.17 compares previous work and our method. The compared results come from the previous research and based on their own processors and testing systems (we did not redo the tests on our ARM11 system). The Fir test has a better estimation for an ARM7TDMI processor than ours in [5] because it considers the overhead energy separately. This will make the model more complex and harder to build but the model would have a better performance. A better estimation of Quicksort comes from the second model in [11] because it considers the data dependency. But for the other tests, our model gives a better prediction. The reason for errors is that even when different programs have the same IPC, they may still have different hardware usage. For example, one program may have more cache misses but the other may have more branch mispredictions or be data dependent. Thus, different hardware usage means different power and energy.

4.8.2 Comparison with the Basic Energy Model

In order to compare the performance of our model with other methods shown in Table 4.17, we ran the tests using the ARM11 board system. All of the models shown in Table 4.17 are related to the basic instruction level energy model discussed in Section 2.1 and based on it. Thus, instead of comparing with each different model, we compare the performance of our model and the basic instruction level energy model.

The basic instruction level model, discussed in Section 2.1.3, can be represented as the following equation:

$$\begin{aligned} E &= E_{Base} + E_{overhead} + \sum_k E_k \\ E_{Base} &= \sum_i (B_i \times N_i) \\ E_{overhead} &= \sum_{i,j} (O_{i,j} \times N_{i,j}) , \end{aligned} \tag{4.5}$$

where the most fundamental factor is the sum of the base energy cost of each instruction E_{Base}, B_i is the base cost of instruction i and N_i is the number of instructions, i, executed in the program. $E_{overhead}$ is the overhead cost, and E_k is any additional energy due to cache misses or resource constraints [7, 8, 29]. However, none of the methods presented in Table 4.17 shows how to measure E_k, thus we assume that the cache miss penalty dominates E_k and the following section will demonstrate how important it is. On the other hand, we have clustered the instructions into three classes: logic, load, and store, thus the base energy cost, E_{Base} will be:

$$\begin{aligned} E_{Base} &= \sum_{i \in (logic, load, store)} (B_i \times N_i) \\ &= B_{logic} \times N_{logic} + B_{load} \times N_{load} + B_{store} \times N_{store}, \end{aligned} \tag{4.6}$$

where B_{logic}, B_{load}, and B_{store} are the average base energy costs of each class: logic instructions, load instructions, and store instructions, respectively. N_{logic}, N_{load}, and N_{store} are the number of instructions in each class, respectively.

Firstly, we try to measure the energy cost by E_{Base}. Assuming that the runtime of a test is T, the corresponding number of clock cycles N_{cycles} will be

$$N_{cycles} = T \times F, \tag{4.7}$$

where F is the clock frequency of the ARM11. Thus, from Equation 4.7, the number of instructions, N, in the program will be:

$$\begin{aligned} N &= N_{cycles} \times IPC \\ &= T \times F \times IPC \end{aligned} \tag{4.8}$$

On the other hand, the relationship between the total number of instructions, N, and the distribution of each class of instructions is:

$$\begin{aligned} N_{logic} &= N \times p_{logic} \\ N_{load} &= N \times p_{load} \\ N_{store} &= N \times p_{store}, \end{aligned} \tag{4.9}$$

where p_{logic}, p_{load}, and p_{store} are the percentage of the logic instructions, load instructions, and store instructions, respectively. N_{logic}, N_{load}, N_{store} are the number of instructions from logic, load, and store, respectively. Combining Equation 4.6, Equation 4.8, and Equation 4.9, the following equation can be derived:

$$E_{Base} = (B_{logic} \times p_{logic} + B_{load} \times p_{load} + B_{store} \times p_{store}) \times T \times F \times IPC \tag{4.10}$$

Because energy is equal to the average power multiplied by time, the base energy cost of each instruction can be presented as:

$$\begin{gathered} B_i = P_i \times T_i, \\ (i \in (logic, load, or store)) \end{gathered} \tag{4.11}$$

where P_i is the average base power cost of instruction i and T_i is the time period to finish the instruction i. Because the base power/energy cost of each instruction is measured in the circumstance of a cache hit, for all of these three classes: logic, load, and store, T_i always equals the time period of the clock cycle T_{clk} ($T_{clk} \times F = 1$). Combining Equation 4.10, and Equation 4.11 together, the base energy cost of the program can be described as:

$$\begin{aligned} E_{Base} &= (B_{logic} \times p_{logic} + B_{load} \times p_{load} + B_{store} \times p_{store}) \times T \times F \times IPC \\ &= (P_{logic} \times p_{logic} + P_{load} \times p_{load} + P_{store} \times p_{store}) \times T_{clk} \times T \times F \times IPC \\ &= (P_{logic} \times p_{logic} + P_{load} \times p_{load} + P_{store} \times p_{store}) \times T \times IPC, \end{aligned} \tag{4.12}$$

On the other hand, the overhead energy can be derived similarly based on Equation 4.5 and Equation 4.10:

$$E_{overhead} = (O_{logic\_load} \times p_{load} \times 2 + O_{logic\_store} \times p_{store} \times 2) \times T \times F \times IPC \tag{4.13}$$

where O_{logic_load} and O_{logic_store} are the overhead energy cost between *Logic* and *Load*, and *Logic* and *Store*, respectively. p_{load} and p_{store} are the percentage of *Load* and *Store* instructions, respectively. As the main components of a program are *Logic* instructions and we assume all instructions are distributed evenly, there will be only two overhead energy costs for one *Load* or *Store*. For example, for the sequence of code: *Logic*1, *Store*, *Logic*2, there are two instruction switches: the first is from *Logic*1 to *Store*, and the second is from *Store* to *Logic*2.

Based on Equation 2.1, the relation between the overhead energy $O_{instruction1\_instruction2}$ of the instruction pair *instruction*1 and *instruction*2, and the average power consumption of the instruction pairs, $P_{instruction1\_instruction2}$ described in Section 2.1.2, can be

presented as

$$\begin{aligned} O_{instruction1\_instruction2} &= T_{clk} \times (\frac{P_{instruction1\_instruction2} \times 2 \times N}{2 \times N} - \\ &\quad \frac{P_{instruction1} \times N - P_{instruction2} \times N}{2 \times N}) \\ &= (P_{instruction1\_instruction2} - \frac{P_{instruction1} + P_{instruction2}}{2}) \times T_{clk}, \end{aligned} \tag{4.14}$$

where $P_{instruction1}$ and $P_{instruction2}$ are the base power costs of $instruction1$ and $instruction2$, respectively. N is the number of $instruction1$ and $instruction2$ in the test.

Based on Equation 4.13 and Equation 4.14, the following equation can be derived:

$$\begin{aligned} E_{overhead} &= ((P_{logic\_load} - \frac{P_{logic} + P_{load}}{2}) \times p_{load} + (P_{logic\_store} - \frac{P_{logic} + P_{store}}{2}) \times p_{store}) \\ &\quad \times 2 \times T_{clk} \times T \times F \times IPC \\ &= ((P_{logic\_load} - \frac{P_{logic} + P_{load}}{2}) \times p_{load} + (P_{logic\_store} - \frac{P_{logic} + P_{store}}{2}) \times p_{store}) \\ &\quad \times 2 \times T \times IPC \end{aligned} \tag{4.15}$$

Because it is unclear how to model the energy consumed by a pipeline stall, $\sum_k E_k$, in [7, 8, 29], we assume that the most fundamental part of this cost is the cache miss penalty, and the following equation can be derived.

$$\begin{aligned} E_{miss} &= P_{miss} \times T_{miss}, \\ &= P_{miss} \times N_{cycles\_miss} \times T_{clk}, \\ &= P_{miss} \times (\frac{N}{IPC} - N_{cycles\_hit}) \times T_{clk}, \\ &= P_{miss} \times (\frac{N}{IPC} - N) \times T_{clk}, \\ &= P_{miss} \times N \times (\frac{1}{IPC} - 1)) \times T_{clk}, \end{aligned} \tag{4.16}$$

where N is the total number of instructions, P_{miss} and T_{miss} are the cache miss power consumption and timing penalty, respectively. IPC is instruction per clock cycle. Since ARM11 is a scalar processor, the IPC is one for a cache hit. Therefore, the number of clock cycles spent on cache hits, N_{cycles_hit}, is the same as the number of instructions N.

On the other hand, Figure 4.4 shows the cache miss power consumption, P_{miss}, depends on the instruction types. Thus, it can be presented as:

$$\begin{aligned} P_{miss} &= \sum_i Pm_i \times p_i, \\ &= Pm_{logic} \times p_{logic} + Pm_{load} \times p_{load} + Pm_{store} \times p_{store}, \end{aligned} \tag{4.17}$$

where Pm_{logic}, Pm_{load}, and Pm_{store} are the cache miss power consumption of instruction logic, load and store, respectively.

Combining the Equation: $T_{clk} \times F = 1$, Equation 4.8, Equation 4.16, and Equation 4.17, the cache miss energy consumption is calculated by the following equation:

$$\begin{aligned} E_{miss} = &(Pm_{logic} \times p_{logic} + Pm_{load} \times p_{load} + Pm_{store} \times p_{store}) \\ &\times (\frac{1}{IPC} - 1)) \times T \times IPC, \end{aligned} \tag{4.18}$$

thus, based on Equation 4.5, Equation 4.12, Equation 4.15, and Equation 4.18, the total energy and the average power consumption can be presented as:

$$\begin{aligned} E =& E_{Base} + E_{overhead} + E_{miss} \\ =& ((P_{logic} \times p_{logic} + P_{load} \times p_{load} + P_{store} \times p_{store}) + \\ & (P_{logic\_load} - \frac{P_{logic} + P_{load}}{2}) \times p_{load} \times 2 + \\ & (P_{logic\_store} - \frac{P_{logic} + P_{store}}{2}) \times p_{store} \times 2 + \\ & (Pm_{logic} \times p_{logic} + Pm_{load} \times p_{load} + Pm_{store} \times p_{store}) \times (\frac{1}{IPC} - 1)) \times \\ & T \times IPC, \end{aligned}$$

$$\begin{aligned} Power_{average} = & \frac{E}{T} \\ =& ((P_{logic} \times p_{logic} + P_{load} \times p_{load} + P_{store} \times p_{store}) + \\ & (P_{logic\_load} - \frac{P_{logic} + P_{load}}{2}) \times p_{load} \times 2 + \\ & (P_{logic\_store} - \frac{P_{logic} + P_{store}}{2}) \times p_{store} \times 2 + \\ & (Pm_{logic} \times p_{logic} + Pm_{load} \times p_{load} + Pm_{store} \times p_{store}) \times (\frac{1}{IPC} - 1)) \times IPC, \end{aligned} \tag{4.19}$$

where $Power_{average}$ is the average power consumption of the basic energy model.

Based on Equation 4.19, Figure 4.11 shows the comparison between the average power consumption of the base model, $Power_{average}$, the estimate from our method, and the measured power. The base power consumption data of each class (P_{logic}, P_{load}, and P_{store}) comes from the test results presented in Figure 4.3. The power consumption data of the instruction pair P_{logic_load} and P_{logic_store} comes from test4 and test6 in yellow Section 4.6. The cache miss power data of each class (Pm_{logic}, Pm_{load}, and Pm_{store}) comes from Figure 4.4. Both the data for the IPC and the distribution of different instruction types come from Figure 4.9.

The first bar of each test is the $Power_{average}$, which consists of three parts as described in Equation 4.19: the average base power cost, $\overline{P_{Base}}$, the average overhead power cost $\overline{P_{overhead}}$, and the average cache miss power consumption $\overline{P_{miss}}$.

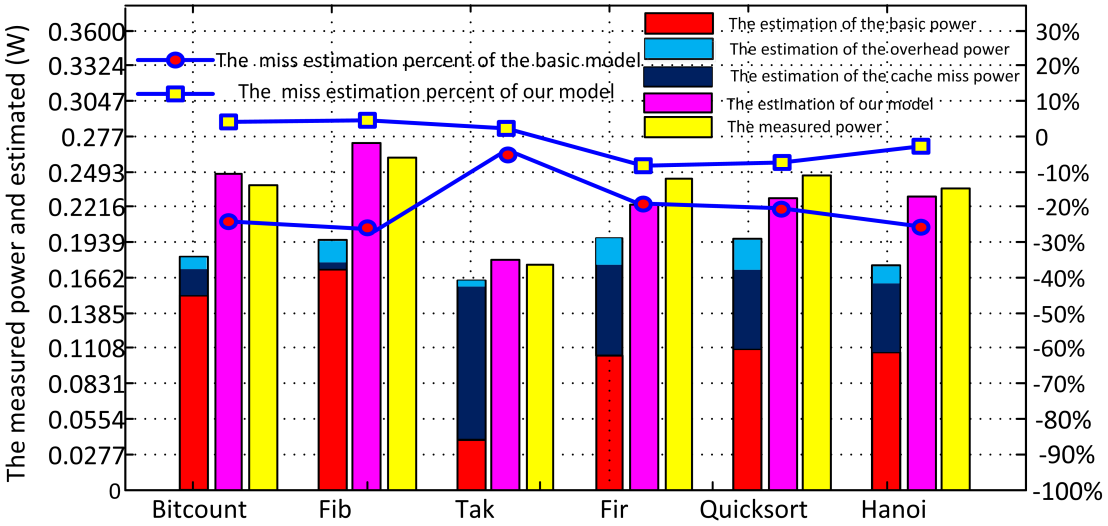


Figure 4.11: The estimation of our model, the basic energy model, and the power consumption of the measurement.

Based on the test results in Figure 4.11, it is clear that our method is more accurate. For example, the most accurate estimate by the base energy model is Tak, which is in error by -6.29%. All of the other test errors are between 0% to -30%. The average absolute error is 20.44%. The reason is that the ARM11 supports branch prediction and an OoO pipeline, thus the IPC may be low but the processor is still busy. In other words, a lower IPC does not have to mean a pipeline stall and the processor could do some useless work and consume more power. Thus, most estimates from the base energy model are always less than the reality.

It is also very clear that the cache miss power consumption is very important. For example, for test Tak, the average cache miss power consumption, $\overline{P}_{miss}$, is 3.07 times as much as the average base power cost $\overline{P}_{Base}$. Thus, if the basic instruction-level energy model does not consider this, the model is inaccurate.

4.8.3 Discussion: Low Energy Software

Power has a close relationship with energy, thus the power model can be extended to study the energy usage of a program. In this subsection, we will consider how the power model might be applied to writing low energy software.

Energy per instruction (EPI) describes the energy efficiency of a microprocessor [105]. We use EPI to estimate the energy efficiency as follows:

$$\begin{aligned} EPI &= \frac{Energy}{N} = \frac{P \times T}{N} = \frac{P}{N/T} \\ &= \frac{P}{N/(Clycles \times (1/F))} \\ &= \frac{P}{IPC \times F}, \end{aligned} \tag{4.20}$$

where N, P and F are the total number of operations in the program, the average power and the frequency of the processor, which is $533MHz$ in this case, respectively. Combining Equation 4.4 and Equation 4.20 leads to the following equation:

$$\begin{aligned} EPI =& \frac{P}{IPC \times F} \\ =& \frac{0.1882 - 0.0601 \times p_{logic} + 0.0081 \times p_{store} + 0.1251 \times IPC}{IPC \times F} \\ =& \frac{C_1}{IPC \times F} + \frac{0.1251}{F}, \end{aligned} \tag{4.21}$$

where C_1 is $0.1882 - 0.0601 \times p_{logic} + 0.0081 \times p_{store}$. It is clear that the EPI is inversely proportional to IPC. Therefore, if programs have similar instruction distributions, the bigger the IPC, the less energy is consumed by each instruction.

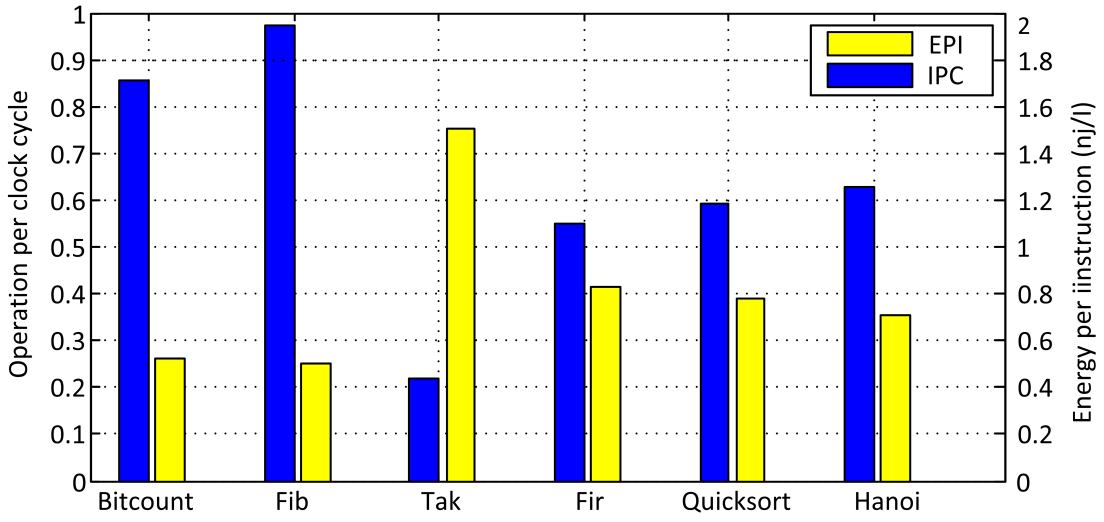


Figure 4.12: The energy per operation VS instruction per clock cycle.

Table 4.18: The benchmarks ranked by EPI and IPC.

	Bitcount	Fib	Tak	Fir	Quicksort	Hannoi
EPI	5	6	1	2	3	4
IPC	2	1	6	5	4	3

Figure 4.12 shows the Energy Per Instruction and IPC of each test, and Table 4.18 ranks the workloads by EPI and IPC in detail. They demonstrate the conclusion proposed by Equation 4.21. For example, the speed of Fib is the fastest but the EPI is the smallest. In contrast, Tak is the slowest but has the biggest EPI. Furthermore, if pipeline stalls can be reduced, by reducing, for example, the cache miss rate, the energy usage will be better. Consequently, it is important to make the pre-fetch unit and branch predictor run more efficiently to reduce pipeline stalls.

However, the pre-fetch is not always helpful, especially if it fetches wrong instructions and there are two reasons for this: (1) if new data is fetched into the cache, some useful data has to be kicked out of the cache because of the limitation of the cache size; (2) it wastes memory bandwidth in fetching useless data and consumes more energy. Hence, the software engineer should consider these aspects when designing applications.

4.9 Conclusions

In this chapter, we present a new instruction-level power model to estimate the average power usage of a program on a single core processor: ARM1176JZF-S. In this model, the power is affected by two factors: the components and the instructions per cycle (IPC) of the program. Instead of studying the different instructions individually, we cluster instructions into three groups: ALU, load and store. The power model is affected by the percent of each groups in the program. It is not necessary to track and find out what the instructions are. Thus, this model does not consider the effect of two adjacent instructions and it is concise and easy to use. On top of this, the maximum error is less than 9% across six benchmarks and the average absolute error of all tests is 4.88%.

Moreover, pipeline stalls are considered by using IPC instead of cache miss as the metric and this makes the model both concise and accurate. Pipeline stalls have not always been well considered in previous models; some models consider too many conditions under which the pipeline might stall. Such models are hard to generate and use. Models that do not consider it sufficiently lose accuracy.

The power model has been extended to a method to estimate the energy consumed by the processor. Comparing this with other models and methods, the advantage is ease of use without losing accuracy. The reason is that the power model is easier to create than a energy model and the runtime of a program is one of the easiest variables to measure. Therefore, we avoid analysing some complicated factors which can affect the energy, such as pipeline stalls.

Furthermore, we consider conditions under which increasing the IPC leads to decreased EPI, both theoretically and empirically.

Chapter 5

ARM Cortex-A8

In order to improve performance, superscalar processors have been developed. Compared with scalar processors such as ARM11, one of the advantages of a superscalar processor is that it implements a form of parallelism called instruction level parallelism. Furthermore, the ideal speed of a normal scalar processor is one instruction per clock cycle, but the speed of superscalar is at least two, depending on the number of pipelines.

The ARM Cortex-A8 is a superscalar processor, widely used in embedded systems such as the iPhone4 [106]. However, the instruction level power/energy consumption of superscalar processors is not well studied. We find the previous basic model (presented in 2.2) cannot be extended to superscalar processors.

In this chapter, we use the ARM Cortex-A8 as the target processor and analyze the instruction-level power/energy consumption of a superscalar processor in detail.

1. A detailed instruction level power analysis is given for the superscalar processor ARM Cortex-A8. Furthermore, the aspects we have studied include: how the power consumption of a processor is affected by L1/L2 instruction and data cache misses; by different instruction types, including arithmetic and logical instructions, load and store; by dual-issue restrictions; by the Hamming distance between the operands of two consecutive instructions; and by the overhead power cost of two adjacent instructions.

2. We show that the previous instruction level energy models do not work for a superscalar processor. Thus, we extend the method for the ARM11 to the ARM Cortex-A8. Furthermore, the model is created based on two factors: the distribution of each class of instructions and IPC.

3. The power model has been extended to estimate the energy consumed by the processor.

5.1 Target Processor

The ARM Cortex-A8 processor is a low power, high performance, cached application processor which provides full virtual memory capabilities [75]. It is targeted at a wide variety of embedded systems such as mobile phones, automotive navigation/entertainment systems and gaming consoles.

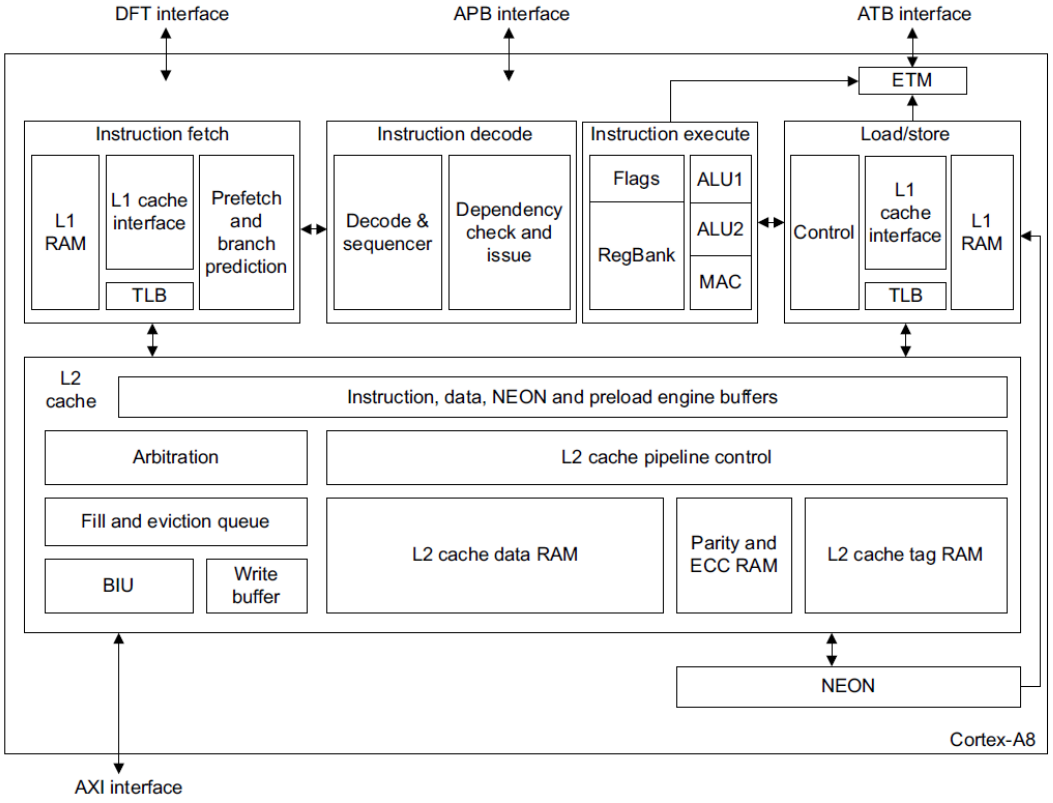


Figure 5.1: Cortex-A8 block diagram [75].

Figure 5.1 shows the architecture of Cortex-A8. This processor contains a three-stage instruction fetch pipeline, five-stage instruction decode pipeline and a five-stage instruction execute and load/store pipeline. Furthermore, it also contains two ALU pipelines, and one load/store pipeline.

Compared with the ARM11, the advantages of the ARM Cortex-A8 are:

1. Superscalar Pipeline

   A superscalar pipeline is probably the most significant of these new features of the ARM cortex-A8 as it can fetch two instructions in one clock cycle. Therefore, the processor runs two instructions in parallel and the IPC is two in the best case. The dual ALU pipelines are symmetric and both can handle arithmetic and logic instructions [75].

2. Support instruction pre-fetch

The Pre-fetch technique is used to speed up the execution of a program by fetching instructions from lower level memory before they are actually needed. The destination of these pre-fetched instructions can be either cache or a pre-fetch buffer and for the Cortex A8, they are stored in a buffer [75].

3. Dynamic branch prediction

   The Cortex A8 uses a more advanced dynamic branch prediction methodology than the ARM11 and the prediction accuracy rate is 95% across industry benchmarks [107]. The branch prediction includes two parts: global history buffers(GHB) and branch target buffers(BTB). Furthermore, the duty of the BTB is to predict whether or not the return instruction of the current fetch address is a branch instruction, if so, it gives the branch target address. If a hit appears in the BTB, the the GHB is accessed. The GHB is used to predict whether or not the conditional branch should be taken [75].

5.2 Experimental Methodology

A Beaglebone REV A6 development board was chosen since it uses the ARM Cortex-A8 as the CPU [72], as part of a TI AM3359 processor. Moreover, it supports dynamic voltage and frequency scaling (DVFS) and also has interfaces for low power memory. The following are the features of this processor which are related to our experiments [75]:

- Full implementation of the ARM architecture v7-A instruction set;
- A pipeline for executing ARM integer instructions;
- Dynamic branch prediction with branch target address cache, global history buffer, and 8-entry return stack;
- Memory Management Unit(MMU) and separate instruction and data Translation Look-aside Buffer(TLBs) of 32 entries each;
- Level 1 instruction and data caches of 16kB or 32kB configurable size, in our case we use 32kB for both instruction and data caches;
- Level 2 caches of 0kB, 128kB through 1MB configurable in size; in our case we use 256kB for the level 2 cache.

Figure 5.2 shows the original power supply of the ARM Cortex-A8 processor and how we modified it. To make the necessary power measurements, we inserted a 1 Ω series resistor with 1% tolerance between the power supply and the CPU, and used a digitizing oscilloscope, the Agilent MSO7012B, with a sample rate of 2GHz to measure the instantaneous power as tests were carried out. We used two probes to measure each side

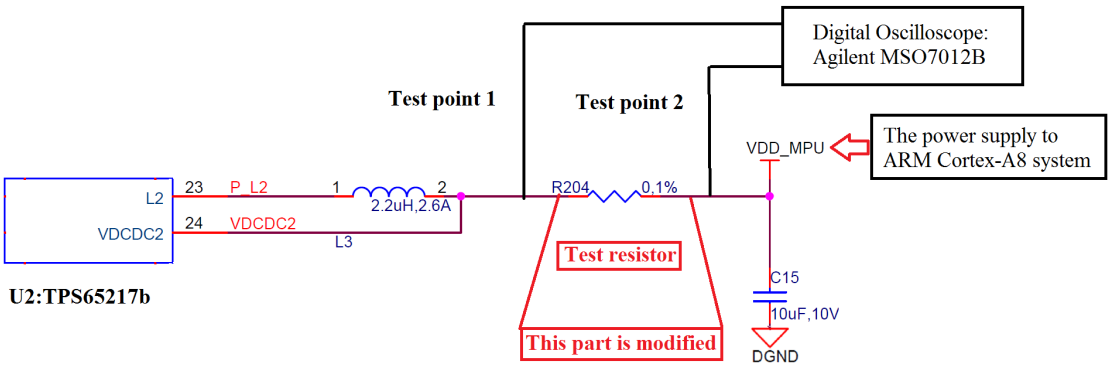


Figure 5.2: Power supply schematic diagram [108].

of the resistor. The instant power model, the average power model and the total energy model is the same as Equations 4.1, 4.2, and 4.3, which are defined in Section 4.2.

5.3 The Power Consumption of Different Instructions When Dual-Issue

As the discussion in Chapters 2 and 4 shows, one of the most significant components of a model is the base power/energy cost of instructions. A superscalar processor, such as the Cortex-A8, can fetch two instructions in one clock cycle and has two ALU pipelines. Hence, when the arithmetic and logic instructions do not have data dependency, two instructions can be run in parallel and the IPC will be two. This section focuses on this situation and analyses how instruction types and cache misses affect the power consumption of the processor.

5.3.1 The IPC for Each Dual-Issue Test

The test is to study the individual instruction power consumption when instructions are run in parallel. The main body of each test is a loop where all of the opcodes in the loop are the same. In order to study how cache misses affect the power consumption, especially L1 instruction cache misses, we increase the loop body size in different tests. Here, both the L1 data cache and instruction cache are 32kB and the L2 cache is 256kB.

The following is an example of pseudo code which is used to measure the base power cost of AND. The operand and opcodes are almost the same as for the ARM11 basic power test in Section 4.3. The only different is loop size is changed for achieving different cache miss rate. An example of the full test code is presented in Appendix A.2.1.

```
while(i<0xFFFF);
{//0xFFFF times the number of the instructions in each loop equals to the
    total number of instructions
```

```
// the total number divided by the number of clock cycles equals to the IPC
asm(" AND r3, r2, r1 ");
asm(" AND r4, r1, r2 ");
asm(" AND r3, r2, r1 ");
asm(" AND r4, r1, r2 ");
 ..............
asm(" AND r3, r2, r1 ");
asm(" AND r4, r1, r2 ");
// the number of loop size varies for different tests
}
```

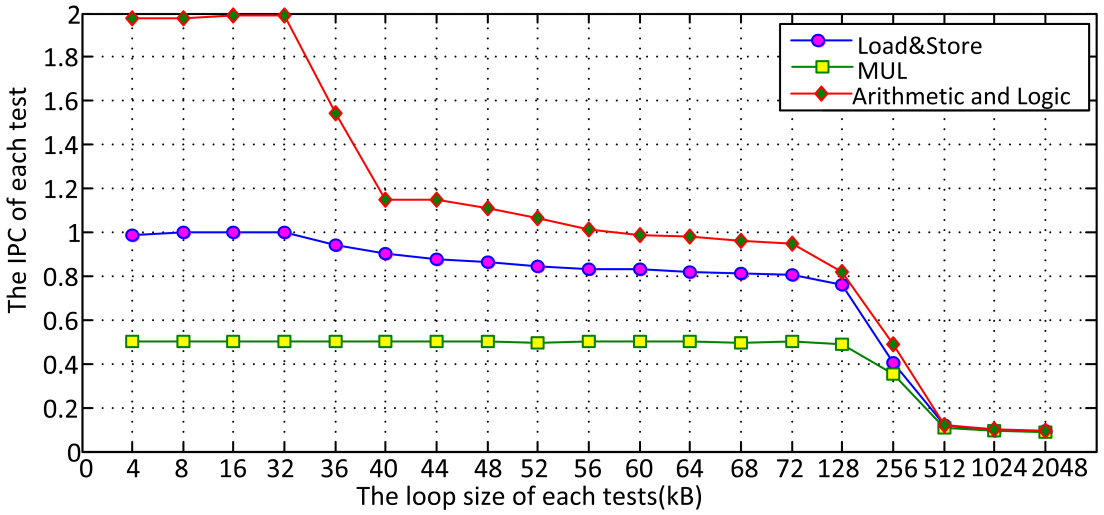


Figure 5.3: The IPC (Instruction per clock cycle) of the each dual-issue test.

Figure 5.3 shows the instructions per clock cycle (IPC) for each of the dual-issue tests. The IPCs of all arithmetic and logic instructions are the same since they use the same pipeline. The load and store use the same pipeline, hence they have the same IPC as well. From the figure, it is obvious that when the loop size is bigger than the L1 instruction cache size, the IPC of ALU logic instructions drops substantially but the IPCs of load and store instructions only drops a little from 0.98 to about 0.82. For MUL instructions, an L1 instruction cache miss does not change the IPC. The reason is that the Cortex-A8 has two ALU logic pipelines, one load/store pipeline and one *MUL* pipeline. Thus, the arithmetic and logic instructions are consumed faster than the other instructions. The load/store pipeline and MUL pipeline can get enough instructions to run and do not have to wait for the instructions to be fetched from the L2 cache. In contrast, the two ALU logic pipelines do not have enough instructions to execute and have to wait. Thus, the IPC of the ALU logic pipelines decreases.

From Figure 5.3, we can also find that for the L2 misses, the IPC of all instructions falls and finally becomes the same, about 0.096. The reason is the time for fetching instructions from main memory is much longer compared with fetching instructions from the cache. For example, the access time of cache and main memory is 0.5-15ns,

and 30-200ns, respectively [21]. An IPC equal to 0.096 means it takes 10.41 cycles, on average, to fetch one instruction from external memory or main memory. The penalty is lower compared with the data presented by Hennessy et al. [21] and the reason is the pre-fetch, so that when the next cache line instruction is actually used, it is already half-way loaded.

5.3.2 The Power Consumption of Arithmetic and Logic Instructions

Table 5.1: The sample standard deviation (STDEVA) and margin of error (MOE) of the test of arithmetic and logic instructions when cache hits.

	$MOV(i)$	$MOV(r)$	MUL	$ADD(i)$	$ADD(r)$	$AND(i)$	$AND(r)$	$SUB(i)$	$SUB(r)$	
STDEVA	0.002654	0.001477	0.000921	0.002462	0.00209	0.002042	0.001907	0.002183	0.001664	
MOE(W)	0.002601	0.001448	0.000903	0.002412	0.002049	0.002001	0.001869	0.00214	0.00163	
	$EOR(i)$	$EOR(r)$	$OR(i)$	$OR(r)$	$ASR(i)$	$ASR(r)$	$LSL(i)$	$LSL(r)$	$LOAD$	$STORE$
STDEVA	0.001963	0.001317	0.003004	0.00212	0.002175	0.002318	0.002089	0.002176	0.000985	0.002387
MOE(W)	0.001924	0.00129	0.002944	0.002077	0.002131	0.002271	0.002048	0.002132	0.000965	0.002339

Table 5.1 shows the sample standard deviation (STDEVA) and margin of error (MOE) of the ARM Cortex-A8 arithmetic and logic instructions for cache hits. This test aims to investigate the behaviour of each instruction with different cache miss rates. The power consumption should be similar for all cache hits. Thus, in Table 5.1 each test is measured four times with the different cache usages: 4kB, 8kB, 16kB and 32kB. If any test power was more than 5% different from the average, we measured it again. The MOE is calculated in a 95% confidence level. For example, the MOE of $MOV(i)$ is 0.002601, which means that we can be 95% confident that the power consumption of $MOV(i)$ is the average power of the measurement plus or minus 0.002601 W.

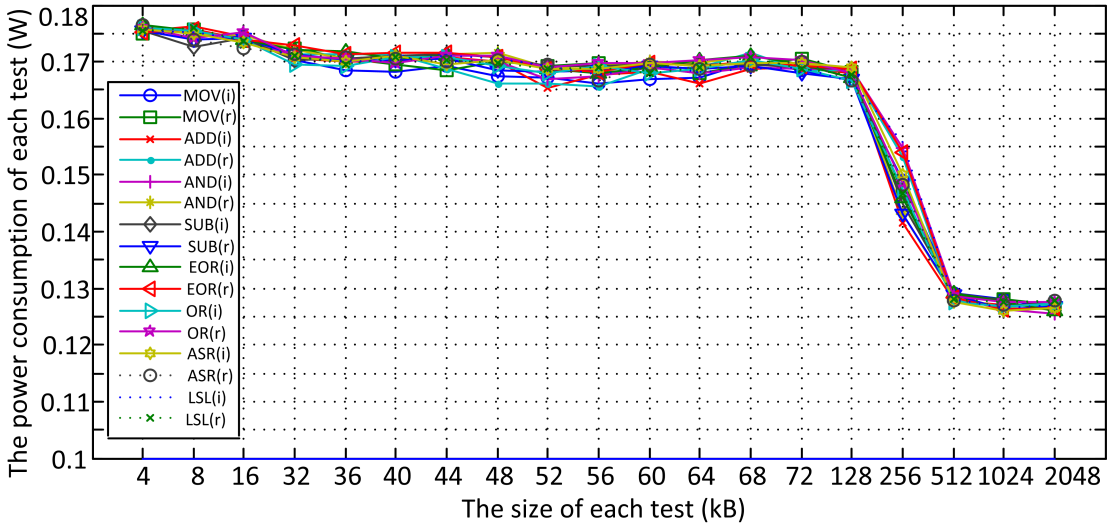


Figure 5.4: The basic power consumption of arithmetic and logic instructions.

Figure 5.4 shows the power consumption of arithmetic and logic instructions in different addressing modes. The following conclusions can be drawn:

- For different instructions, the average power consumption of the processor is similar and the opcode does not affect the power much. For a cache hit, where the loop body size is less than the L1 instruction cache size, the instruction $MOV(i)$ consumes the most power at $0.176W$ and the instruction $OR(i)$ consumes the least power at $0.169W$ (giving the maximum difference of 3.97%). This is also found in some other work [67] but their target processor is not superscalar. The reason is that all these functions pass though the pipeline and the hardware used by these different logic operations is quite similar.

- For different instructions, the addressing mode does not affect the power much. For example, the minimum difference between $MOV(i)$ and $MOV(r)$ is 0.032%, which comes from the 2kB test and the maximum difference is 2.07%, which comes from the 56kB test.

- The L1 cache misses do not significantly affect the power. For example, the average power consumption of the 4kB and 36kB tests are $0.176W$ and $0.170W$ respectively (the difference is 3.07%). It is obvious that the power consumption for the 36kB tests and the 72kB tests is similar and the power curves are nearly straight lines. For example, the power consumption of $ADD(i)$ is $0.171W$ and $0.169W$ in the 36kB and 72kB tests, respectively. The difference is less than 1%. Furthermore, the average power consumption of these two tests is $0.170W$ and $0.169W$, respectively.

  Although the power consumption is quite similar in the 36kB and 72kB tests, the level one cache miss rate is quite different in these two cases. In the 36kB tests, cache misses start to appear. In the 72kB, the L1 cache misses are much more common than cache hits but no level two cache misses occur. However, it is hard to find the specific and accurate cache miss rate, because the cache replacement policy is random replacement. For the 36kB test, the cache can hold 88.8% (32kB/36kB) of all of the instructions. However, for the 72kB test, the cache can only hold 42.1% (32kB/72kB). On top of this, as 72kB is more than twice the size of the level one instruction cache, most of the useful instructions have been replaced after one loop of the test. Thus, the cache hit rate for 72kB should be even less than 42.1%. Consequently, we can draw the conclusion that the level one cache miss rate does not affect the power much.

  The reason why an L1 cache miss does not affect the power much is that the IPC falls, but transferring data from L2 to L1 consumes more power. Consequently, the pipeline consumes less power but the caches consume more. Overall, the power is not significantly affected by L1 cache misses.

- Level two cache misses affect the power significantly. When the loop size is bigger than 256kB, the L2 cache cannot hold all of the instructions and some instructions have to be fetched from main memory. The average power drops from $0.169W$ to $0.126W$. The reason is that although the pre-fetch strategy can solve part of the

problem of the speed mismatch between processor and main memory, the processor still has to wait for the fetched instructions and has nothing to do during the fetching, hence the IPC falls and hence the power consumption drops substantially.

Both L1 and L2 cache misses affect the **energy**, especially L2 cache misses. Although L1 cache misses do not affect the power, the IPC is only half as much as for a cache hit (compare the IPC of 32K tests and 72K tests shown in Figure 5.3). Therefore, the energy will double. For L2 cache misses, the IPC of the program will be much less but consume proportionally more energy.

Based on the previous analysis, in order to produce a simple, concise model, we will assume all arithmetic and logic instructions consume the same power for dual issue in all addressing modes, and will not distinguish between them in the rest of the chapter.

5.3.3 The Power Consumption of Load&Store and MUL

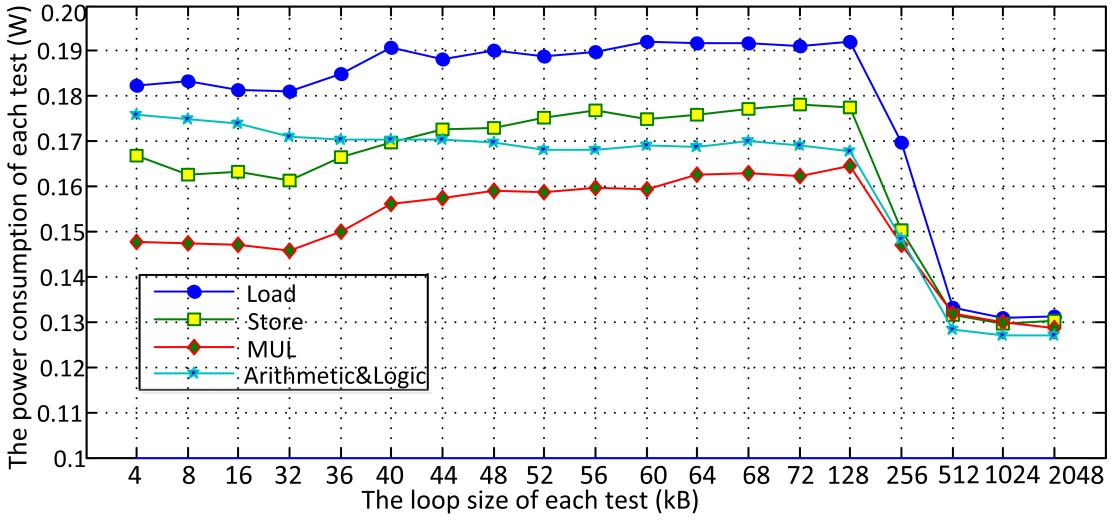


Figure 5.5: The power consumption of Load&Store and MUL.

The second experiment is to test the power consumption of load/store and multiply instructions including both instruction cache misses and hits. Figure 5.5 shows the experimental results. In order to compare with arithmetic and logic instructions, the average power consumption of arithmetic and logic instructions is also displayed in the figure. The following conclusions can be drawn:

- When the L1 instruction cache hits and all of the instructions are fetched from the L1 instruction cache, the load consumes the most power, which is about $0.182W$ on average. The arithmetic and logic instructions consume the second most. The store consumes about $0.164W$ on average and the difference between store and load is less than 8%. However, the MUL instruction consumes the least power, which is

about $0.147W$ on average, and the difference is quite big compared with the other three instruction types. For example, the difference between MUL and load, store and arithmetic/logic instructions are 23.8%, 11.56%, and 19.04%, respectively.

The reason is that the load and store use a different pipeline to the arithmetic and logic instructions. For the load test, the operand addresses are the same and so there are no data cache misses and the processor runs at a high speed. Hence, load consumes more power than arithmetic logic functions. For the store test, the target address are always the same, thus the store buffer will never be full. Thus, the data in the buffer is only updated locally, and the store does not consume more energy since the data is never sent to external memory. Consequently, store uses less average power than load although they use the same pipeline. The IPC of a multiply is lower than the other types of instructions and hence MUL consumes the least power.

- When the loop body is bigger than 32kB but less than 256kB, which means there are L1 instruction cache misses but L2 cache hits, the power of arithmetic and logic instructions falls a little but all of the other three rise. From Figure 5.3, the IPC for Load/Store falls a little, but stays constant for Multiply. Nevertheless, the processor still gets enough instructions to (nearly) fill the pipeline and does not have to wait for the L2 cache.

  However, for the ALU instructions, the pre-fetch design cannot solve the cache miss time penalty problem completely. The ALU instructions are consumed too fast by the processor and the ALU pipeline is more likely to be hungry than the Load/Store pipeline. Thus, the processor still has to wait for the fetched ALU instructions from the L2 cache and the IPC decreases a lot. On the other hand, fetching instructions from the L2 cache consumes more power. Consequently, when there is an L1 instruction cache miss, Load, Store and Multiply consume more power.

- For an L2 cache miss, the power for all of the instructions drops significantly. After 512kB, the power of all of instructions is about $0.128W$ which is the same as in the ALU test. The reason is the same: the instructions have to be fetched from main memory and the processor has to wait. Likewise, the energy increases for both L1 and L2 cache misses.

5.4 The Power Consumption of Dual-issue Restrictions

However, there are several dual-issue restrictions that mean the processor cannot run in parallel. For example, instructions cannot be issued if their data is not available. Furthermore, if one instruction's operands come from the previous instruction's results, it has to wait until the result of its previous instruction are ready. In this section, the

tests focus on the power consumption when the instructions meet the constraints and have to be run one by one.

As in the previous tests, the main body is a loop and the opcode is still the same, but the operand of a new instruction depends on the previous one. However, there are only 15 general registers in Cortex-A8 and it is forbidden to use all of them. Therefore, in our test, the instruction will repeat every eight instructions. The following is an example of the code.

```
while(i<0xFFFF);
{//0xFFFF times the number of the instructions in each loop equals to the
    total number of instructions
// the total number divided by the number of clock cycles equals to the IPC
//the first 8 instructions
asm(" AND r3, r2, r1 ");
asm("  AND r4, r3, r2 ") ;
 ..............
asm("  AND r10, r9, r8 ") ;
// the second 8 instructions
asm("  AND r3, r2, r1  ");
asm("  AND r4, r3, r2 ") ;
 ..............
asm("  AND r10, r9, r8 ") ;
.....................
}
```

Table 5.2: The operand of dual-issue restrictions test.

Test	Code	Test	Code
$ALU(i)$	r0: 0x3 r1:0x5 r2: 0x5 ALU r1, r0, #0x3 ALU r2, r1, #0x5 ALU r3, r2, #0x3 ALU r4, r3, #0x5 ALU r5, r4, #0x3 ALU r6, r5, #0x5 ALU r7, r6, #0x3 ALU r8, r7, #0x5 repeat N times	$ALU(r)$	r0: 0x3f r1:0x11 ALU r2, r1, r0 ALU r3, r2, r1 ALU r4, r3, r2 ALU r5, r4, r3 ALU r6, r5, r4 ALU r7, r6, r5 ALU r8, r7, r6 ALU r9, r8, r7 repeat N times

Table 5.2 shows the operand and opcode of the test. The eight instructions are repeated N times for achieving different cache miss rates. An example with the full structure is presented in Appendix A.3.1.

5.4.1 The IPC of the Dual-issue Restriction Tests

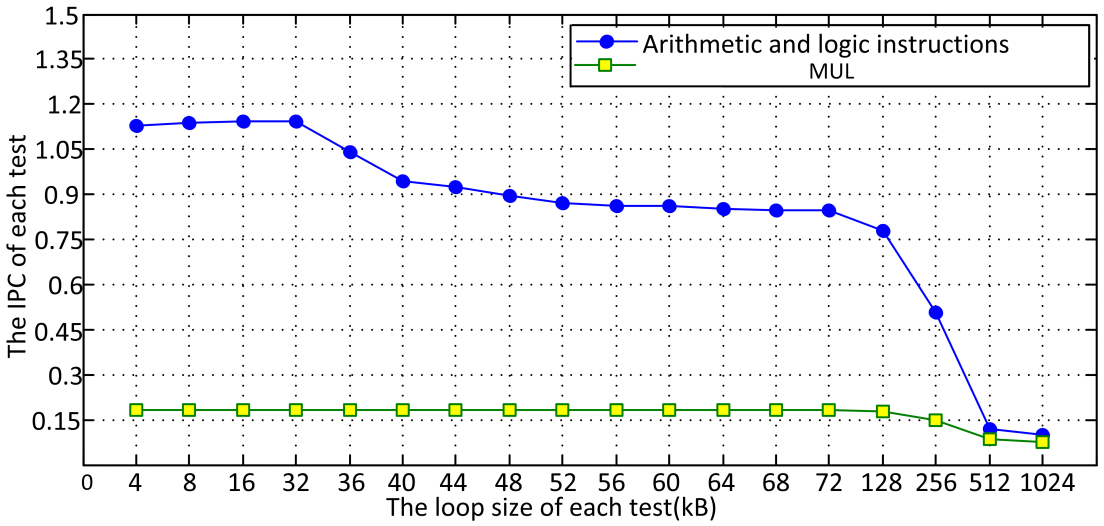


Figure 5.6: The IPC (Instruction per clock cycle) for dual-issue restriction tests.

Figure 5.6 shows the IPC of the dual-issue restriction test and it is quite similar to the IPC of the basic tests in Section 5.3.1. For the arithmetic and logic instructions, when the L1 cache misses and the L2 cache hits, the IPC start to drop from about 1.130 to 0.845. After L2 cache misses appear, the IPC drops again to 0.097, which is equal to that for the basic tests in Section 5.3.

L1 cache misses do not affect the IPC of the MUL instruction at all because of its slow speed. However, L2 cache misses affect the IPC significantly, because it takes more time to fetch instructions from main memory.

5.4.2 The Power Consumption of Dual-issue Restrictions

Table 5.3: The sample standard deviation (STDEVA) and margin of error (MOE) of dual-issue restrictions when cache hits.

	$ADD(i)$	$ADD(r)$	$AND(i)$	$AND(r)$	$EOR(i)$	$EOR(r)$
STDEVA	0.000873	0.000294	0.000155	0.001075	0.001014	0.001085
MOE (W)	0.000856	0.000288	0.000151	0.001053	0.000994	0.001064
	$OR(i)$	$OR(r)$	$SUB(i)$	$SUB(r)$	MUL	
STDEVA	0.001238	0.000757	0.000813	0.000812	0.00131	
MOE (W)	0.001213	0.000742	0.000796	0.000796	0.001284	

Similar to Table 5.1 in Section 5.3.2, Table 5.3 shows the sample standard deviation (STDEVA) and margin of error (MOE) of the ARM Cortex-A8 dual-issue restriction tests when cache hits. In Table 5.3 each test is measured four times with the different cache usage:4KB, 8KB, 16KB and 32KB unless any test power is more than 5% different from the average. The MOE is calculated in a 95% confidence level.

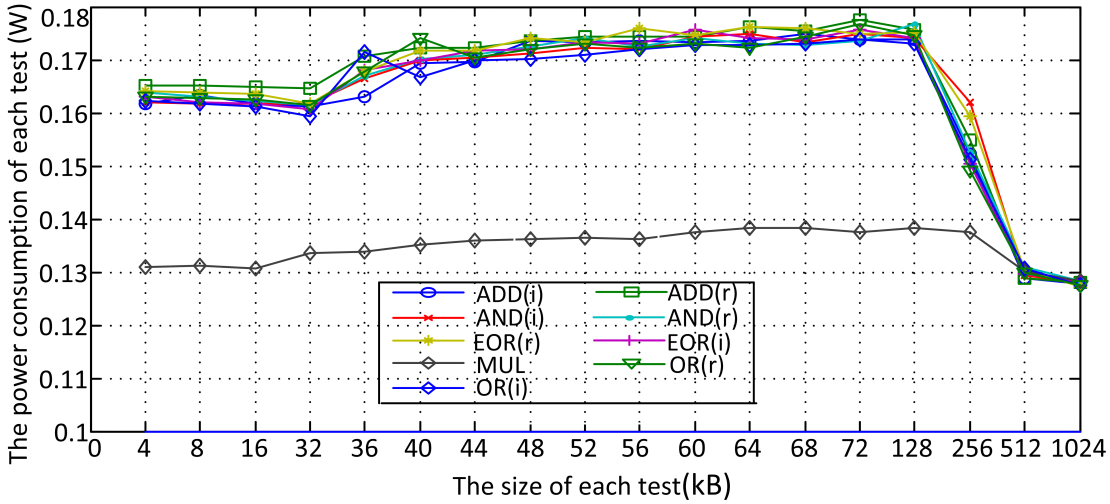


Figure 5.7: The power consumption of dual-issue restrictions.

Figure 5.7 shows the power consumption of dual-issue restrictions. In order to analyze the instruction cache miss effects, especially the L1 instruction cache effects, the size of the loop body changes from 4kB to 1MB and the following conclusions can be drawn:

- It is clear that when the L1 instruction cache hits, all of the arithmetic and logic instructions consume similar power, which is about $0.163W$. The reason is that all of the instructions use similar hardware. However, the MUL still consumes the least power and even an L1 instruction cache miss will not affect the power very significantly because the speed of the MUL is the slowest.
- When the L1 instruction cache misses but the L2 cache hits, the power is greater than before, but not by very much. For example, the average power consumption of cache hits and cache misses is $0.1636W$, and $0.1727W$ respectively. This is similar to Figure 5.5. The reason is the processor runs slower than the dual-issue case because of the dual-issue constraints. For example, the IPC is about 0.9, compared with about 1.1 in the dual-issue test (Figure 5.3). The power increases a little overall, because the processor does not lose too much speed compared with the cache hits,L1 only 18.18%, but accessing the L2 cache consumes more power. Thus, the power increases from $0.1635W$ to $0.1726W$ on average.

5.5 The Power Consumption with Different Hamming Distances

Previous research suggests that the Hamming distance between the operands of two consecutive instructions may affect the power consumption [11, 34]. Our third test considers this on a Cortex-A8 when the L1 cache always hits. The test codes are the same as ARM11 in Section 4.4 and an example is presented in Appendix A.2.2.

Table 5.4: The sample standard deviation (STDEVA) and margin of error (MOE) of the Cortex-A8 Hamming distance power test.

	4		8		12		16	
	STDEVA	MOE(W)	STDEVA	MOE(W)	STDEVA	MOE(W)	STDEVA	MOE(W)
Store	0.000635	0.000718	0.000797	0.000902	0.000329	0.000372	0.000948	0.001072
Load	0.000166	0.000188	0.001327	0.001501	0.000948	0.001072	0.000894	0.001012
$ADD(i)$	0.000562	0.000636	0.000265	0.000299	0.000123	0.000139	0.000861	0.000974
$ADD(r)$	0.000649	0.000734	0.001538	0.001741	0.00124	0.001403	0.001223	0.001384

Table 5.4 shows the standard deviation (STDEVA) and margin of error (MOE) of the Cortex-A8 Hamming distance power test. We measured each test three times. However, if the difference of any two tests was bigger than 5%, we measured another time. The MOE is calculated in a 95% confidence level.

Figure 5.8 shows the test results including $ADD(i)$, $ADD(r)$, *Store*, *Load*. In these figures, the X-axis represents the operand Hamming distance of two consecutive instructions and we chose 4, 8 12 and 16 for the target experiments.

From Figure 5.8, we can see that the power consumption increases with the Hamming distance of the operand. However, the Hamming distance does not affect the power consumption significantly. For example, the maximum difference of *Store* is only 1.96%; the maximum differences of $ADD(i)$, $ADD(r)$ and *Load* are 0.80%, 0.24% and 1.74% respectively. Hence, the maximum and average difference of all tests is only 1.96% (from *Store*) and 1.185%, respectively. Because the effect of the Hamming distance is small, in order to have a concise model, we have not put any Hamming distance variables into our power model.

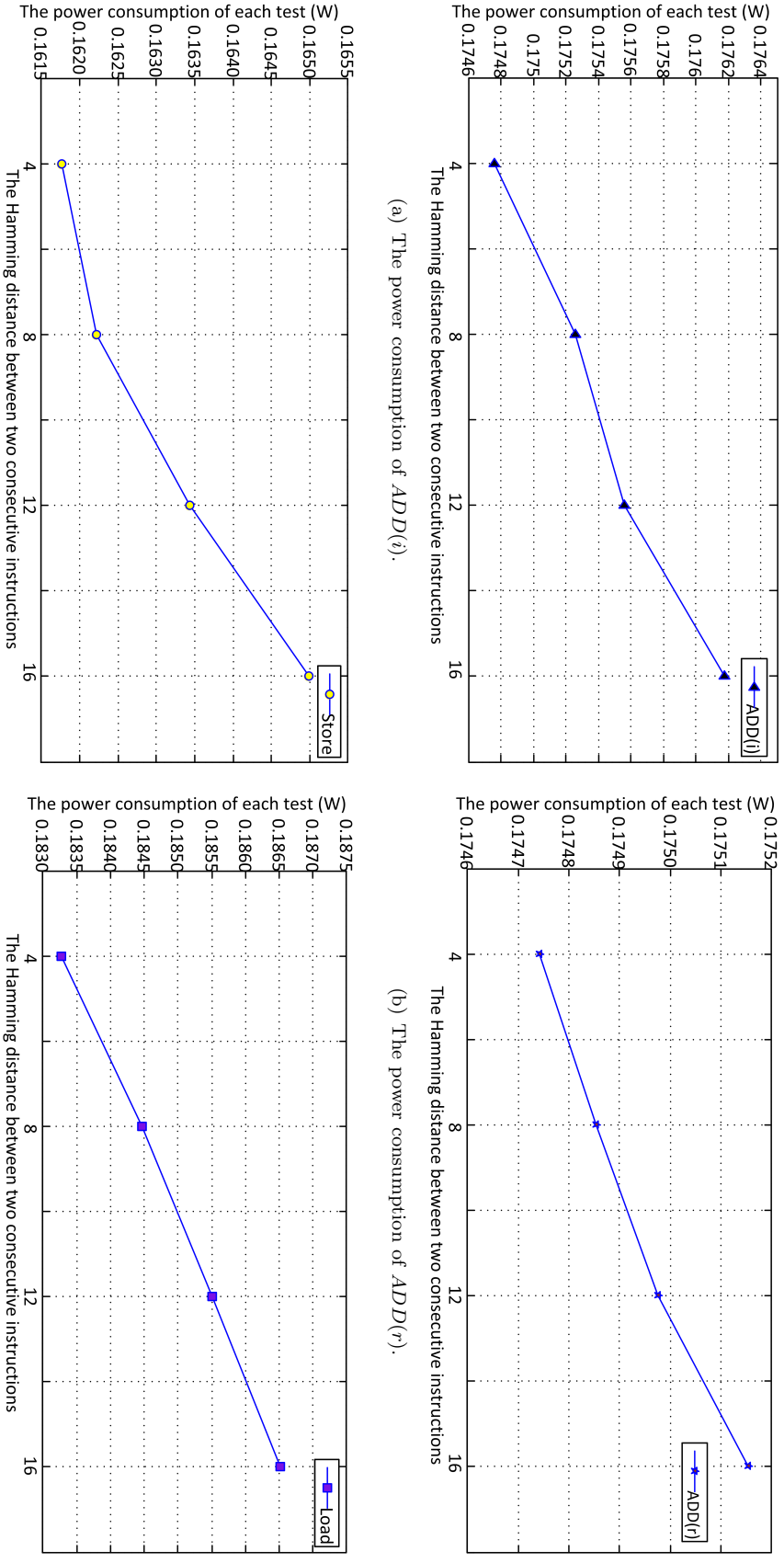
5.6 The Overhead Power Cost

Previous research suggests that the overhead of two consecutive instructions may affect the power consumption [7, 8]. We have proved that the effect on arithmetic and logic instructions can be ignored on an ARM11 processor in Section 4.5. The fourth test in this Chapter analyses the overhead power cost of arithmetic and logic instructions on a Cortex-A8 when the L1 cache always hits. The test codes are the same as ARM11 in Section 4.5 and an example is presented in Appendix A.2.3.

Table 5.5: The sample standard deviation (STDEVA) and margin of error (MOE) of each instruction.

	$ADD(r)$	$AND(r)$	$SUB(r)$	$OR(r)$
STDEVA	0.001083	0.001016	0.000834	0.000861
MOEW	0.001225	0.00115	0.000944	0.000975

Table 5.5 and Table 5.6 shows the sample standard deviation (STDEVA) and margin of error (MOE) of each instruction and instructions pairs respectively. We measured each



(a) The power consumption of $ADD(i)$.

(b) The power consumption of $ADD(r)$.

(c) The power consumption of *Store*.

(d) The power consumption of *Load*.

Figure 5.8: The power consumption with different Hamming distances.

Table 5.6: The sample standard deviation (STDEVA) and margin of error (MOE) of instruction pairs.

	$AND(r)$		$SUB(r)$		$OR(r)$	
	STDEVA	MOEW	STDEVA	MOEW	STDEVA	MOEW
$ADD(r)$	0.000852	0.000964	0.000944	0.001068	0.001594	0.001804
$AND(r)$			0.001298	0.001469	0.000727	0.000823
$SUB(r)$					0.001227	0.001389

test three times unless the difference of any two tests was bigger than 5%. The MOE of each table is calculated in a 95% confidence level.

Table 5.7: The power consumption of each instruction (W).

$ADD(r)$	$AND(r)$	$SUB(r)$	$OR(r)$
0.17505	0.17546	0.17448	0.17511

Table 5.8: The average power consumption of each pair (W).

	$AND(r)$	SUB	$OR(r)$
ADD	0.17525	0.17476	0.17508
AND		0.17497	0.17529
SUB			0.17480

Table 5.7 and Table 5.8 show the power consumption of each instruction and the average power consumption of each pair respectively. Because the base power consumption of each instruction is similar, the average power of each pair is close.

Table 5.9: The measured power consumption of each pair (W).

	AND	SUB	$OR(r)$
ADD	0.17605	0.18013	0.17878
AND		0.17967	0.18112
SUB			0.17704

Table 5.10: The overhead power and the difference ratio (W).

	AND	SUB	$OR(r)$
ADD	0.000798 (0.05%)	0.005371 (0.30%)	0.003702 (0.21%)
AND		0.004702 (0.26%)	0.005831 (0.32%)
SUB			0.002243 (0.13%)

Table 5.9 shows the measured power consumption of each pair, and Table 5.10 shows the overhead power and the different ratio between the overhead and the average power consumption. The columns are the first instruction of each pair and the rows are the second instruction of the pair.

It is clear that for arithmetic and logic instructions, the overhead power cost is positive but very small, which is expected. For example, the minimum ratio is 0.05%, which is

from the instruction pair ADD and AND. The maximum ratio is 0.32%, which is from AND and $OR(r)$.

Compared with the ARM11, the overhead effect of the arithmetic and logic instructions on the Cortex A-8 is smaller. For example, the average of the overhead ratio of ARM11 is 2.88%, but the ratio of Cortex-A8 is 0.211%. The reason is that the Cortex-A8 is more complicated than the ARM11 and a lot of blocks are shared by the different instructions, such as L1/L2 cache, MMU. However, the Cortex-A8 has more complex shared resources, such as the L2 cache, which consumes more power, thus the cost of the circuit state changes is less important on a Cortex-A8.

Based on these test results, we do not need to consider the overhead cost of different pairs from arithmetic and logic instructions since this effect is small but needs a lot of effort to measure every possibility.

5.7 The Power Consumption of Data Cache

The previous tests focus on the power consumption with dual-issue and its restrictions, and how instruction cache misses affect power. In this section, we consider how data cache hits and misses affect the power.

5.7.1 Design of the experiment

The data cache is configured as follows. The L1 memory system provides the core with [75]:

- write through policy,
- fixed line length of 64 bytes,
- cache size of 32kB,
- two 32-entry fully associative ARMv7-MMU,
- 4-way set associative cache structure,
- random replacement policy.

The L2 memory system provides the core with [75]:

- write-allocate policies,
- cache size of 256kB,

- configurable 64-bit or 128-bit wide AXI system but interface with support for multiple outstanding requests,
- 8-way set associative cache structure,
- random replacement policy.

The cache of the Cortex-A8 in the TI AM3359 is a high performance write-though cache with a buffer at the end of a pipeline. Although the write-back policy is power efficient and faster, the write-though cache gets around a consistency problem and is easier to implement. With the help of a buffer, as long as any external memory accesses are only reads from main memory addresses which are not in the write-buffer, then the write-buffer is able to act independently in the background to update the main memory. Depending on the implementation, a read from a pending write in the write buffer could access the data there without stalling the system. Therefore, this write-though cache is still efficient.

The Cortex-A8 fetch pipeline is a pre-fetch and speculative fetching design. In order to analyze how data cache misses affect the power, we have to write a program and make it access the target memory randomly. If the size of the target memory is less than the cache size, it will get a cache hit, otherwise, cache misses will exist because the cache cannot hold all of the data. The program tests are like the following pseudo code:

```
void main()
{
int target[size];//create a target test memory.
while(i<0xFFFF);//the test should run many times and a big number is used
//0xFFFF times the number of the instructions in each loop equals to the total
    number of instructions
// the total number divided by the number of clock cycles equals to the IPC
{
asm(" MOV r5, %[va]"::[va] "r"(target));
// move the first address of the target array into register R5.
asm (" LDR r9, [r5, #random_1]");
asm (" LDR r8, [r5, #random_2]");
.....
asm (" LDR r8, [r5, #random_n]");
//random is from 0 to the size of the target memory
}
}
```

In this code the offset value can be from 0 to the size of target test memory space. However, the offset value of LDR and STR has a limitation and the compiler will produce a error if the offset value is bigger than 4095 [109]. Therefore, the test is divided

into two parts. Firstly, a target memory space is claimed and divided into several sub-block pieces. Then, an array (baseAddress[i]) is used to store the start address of each sub-block randomly. The sub-block size is smaller than or equal to 4095 depending on the different tests. Therefore, the *LDR* and *STR* instructions can visit every space of the sub-block. The data structure is shown in Figure 5.9.

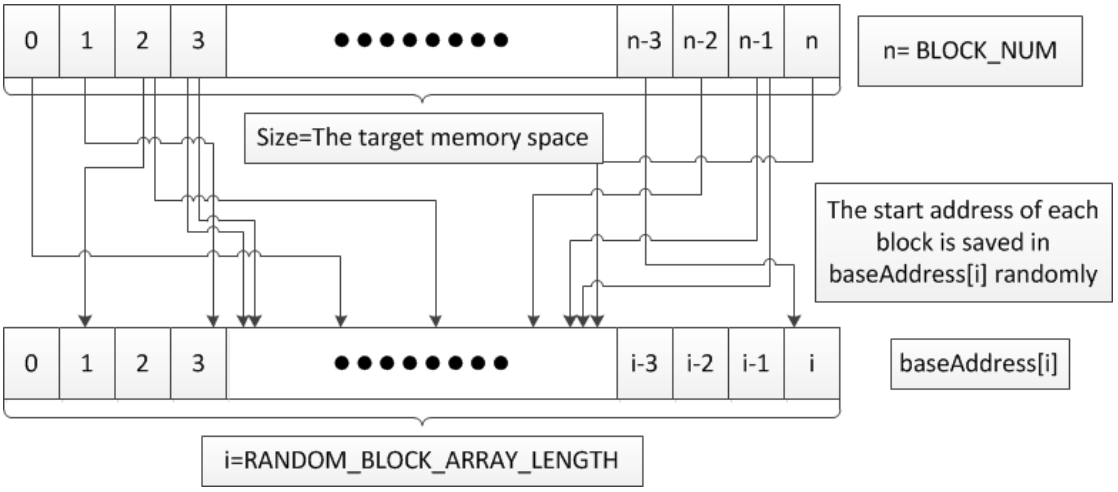


Figure 5.9: The design of the data cache test program.

Figure 5.10 shows the flow chart of the test program: The first part is the initialization. Firstly, we generate a target memory space which is an array and its size can be changed from 1kB to 4MB depending on the requirements of different tests. Then, the second step is the initialization of the array.

Then, we divide the memory space into sub-blocks and use array 'baseAddress[i]' to store the start address of each sub-block in a random sequence. The size of 'baseAddress[i]' is determined by the parameter 'RANDOM_BLOCK_ARRAY_LENGTH'. In order to distribute the value of each 'baseAddress[i]' (here, it is the first address of each sub-block) uniformly, we make sure that the difference of each two consecutive value is bigger than one eighth of the size of memory target space. Meanwhile, we also make the difference of each two consecutive offset value in the main loop bigger than one quarter of the size of sub-block. This will make sure that no two sub-blocks or offset values are close to each other.

After initialization, the main body is two loops. Although we want to run this test many times to test the behaviour when accessing cache or memory, there is a limit to the size of an array in the C compiler. Thus, it is impossible to only increase the value of 'RANDOM_BLOCK_ARRAY_LENGTH' to make the size of 'baseAddress[i]' big enough. Therefore, we put in another variable 'N' to make the test run many times and this is the outer loop.

The inner loop is the core of this test and at the beginning, a new base address will be sent to register 5 from 'baseAddress[i]'. The offset values are also randomly picked,

which makes sure that the speculative fetching does not work. The L2 cache misses exist when the target memory space is bigger than the L2 cache.

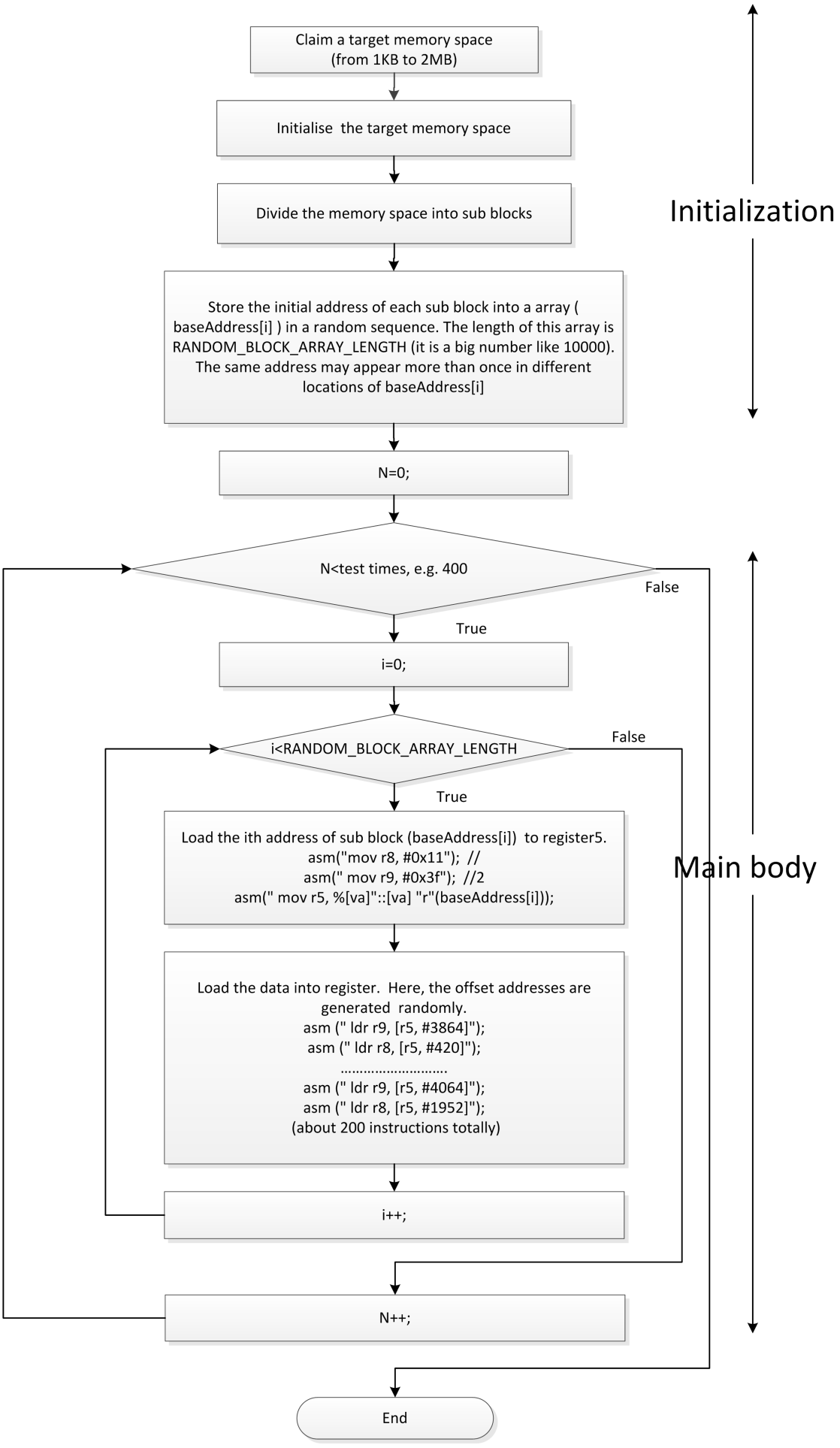


Figure 5.10: The flowchart of the data cache test.

The following is the modified pseudo code and the source code Section A.3:

```
void main()
{
int target[size];
//create a target test memory.
//because each int is 4 Bytes, the target memory space =4 Byte * size
while(i<RANDOM_BLOCK_ARRAY_LENGTH)
{
r = rand()%(BLOCK_NUM);
//pick the rth block in random
baseAddress[i]=target+r*Target_SIZE/BLOCK_NUM;
//Target_SIZE/BLOCK_NUM equals the size of each block
//generate the first address of the rth block address and save into
    baseAddress randomly
}

while(i<RANDOM_BLOCK_ARRAY_LENGTH);
{
asm(" MOV r5, %[va]"::[va] "r"(baseAddress[i]));
//baseaddress[i] holds the first address of every sub block and load into r5.
asm (" LDR r9, [r5, #random_1]");
asm (" LDR r8, [r5, #random_2]");
.....
asm (" LDR r8, [r5, #random_n]");
}
```

5.7.2 The Power Consumption And IPC of The Data Cache Experiment

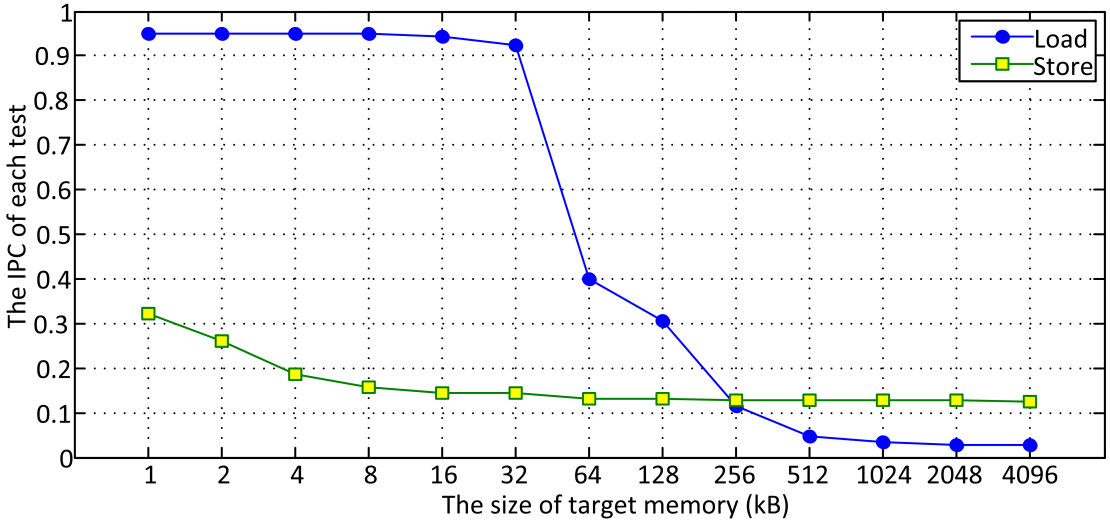


Figure 5.11: The IPC of the data cache tests.

Figure 5.11 shows the IPC of these tests. From the figure, we find that when the target memory is smaller than 32kB, the IPC of load is nearly equal to 1, which means the L1 cache can hold all of data and does not miss, as expected. When the target memory is bigger than 32kB, the IPC of Load drops significantly, because of the L1 cache misses. However, when the size of the target memory is greater than 256kB, the IPC of Load drops again. Finally, beyond 512kB, the IPC is about 0.0268 and no longer changes. The reason is the size of the target memory is too great and the data cache can only hold a small percentage of the total data. Thus, nearly every data has to be fetched from external memory. On top of this, the data pre-fetching is disabled in these tests because of the random accesses.

The IPC of Store is lower than Load when L1 cache hits, and drops as the size of the target memory increases until 8kB. When the size of the target memory is bigger than 8kB, the IPC falls to 0.12. Store is not sensitive to L2 cache misses. The reason is the cache is a write-though cache and the processor is busy writing data to main memory. Writing data to main memory has a speed limitation.

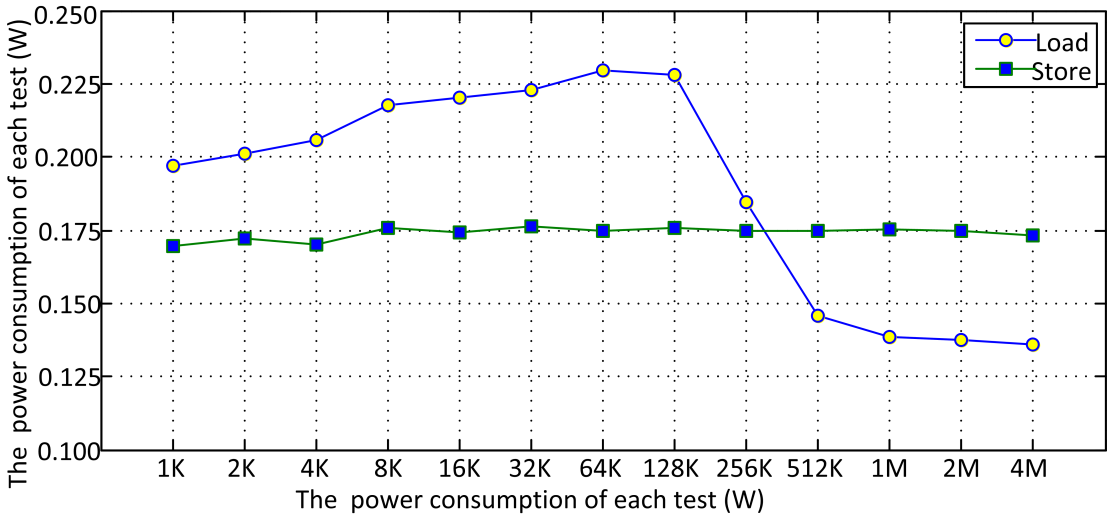


Figure 5.12: The power consumption of the data cache tests.

Figure 5.12 shows the power results of these tests including load and store. When the target memory is less than 32kB the load consumes more power as the size of the target memory becomes bigger. When L1 misses appear, the power does not change much and reaches the peak, which is from $0.223W$ about $0.230W$. The reason is the same as mentioned before: activating the L2 cache consumes more power and the processor does not lose too much speed. However, with L2 cache misses, the power drops substantially and beyond 512kB the power consumption remains approximately constant at about $0.135W$. The reason is the time penalty of L2 misses is too big and a system stall appears.

However, the store consumes a constant power which is independent of target memory space and is from $0.169W$ to $0.175W$. The reason is the cache is a write-though cache and always writes data to main memory when the write-buffer is full.

5.8 Instruction Level Power Analysis and Modeling

In order to study how different instructions affect the power together, a more realistic program environment is provided. We analyze the power by changing the distribution of each type of instruction in different programs and name them the combined tests. From this, we created an instruction-level power model.

5.8.1 The Power Analysis of the Combined Tests

From the previous tests, we know that the L2 cache affects the power more than the L1 cache. Typically, the L1 cache miss rate is less than 10% for a 32kB cache [21]. Similarly, the L2 cache hits more often than misses. We focus on the effect of the L1 cache and divide the tests into two cases: L1 cache always hits and L1 cache always misses. The test codes are almost the same as ARM11 in Section 4.6 and an example is presented in Appendix A.2.4. The only difference is the loop size had to be changed to cause the L1 cache to always miss, because the ARM11 has a 16KB instruction cache and the Cortex-A8 has a 32 KB cache.

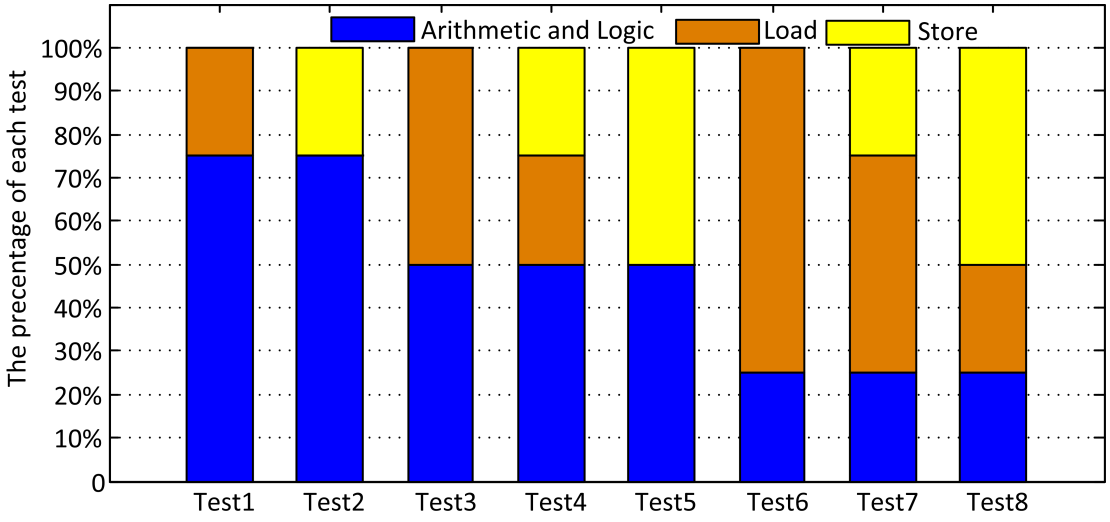


Figure 5.13: The components of the combined tests.

Figure 5.13 shows the components of each test. All of the programs are coded manually, allowing us to understand the distribution of the program details and change it easily. We chose 25% as a step size and the percentage of the arithmetic logic instruction decreases from test 1 to test 8. For example, 75% of instructions in test 1 are arithmetic and logic

instructions but only 25% of instructions come from arithmetic and logic in tests 6, 7 and 8. In contrast, the load and store percentages increase. Finally, Figure 5.13 shows all of the possibilities. For arithmetic and logic instructions, the 100% and 0% cases are ignored because it is unlikely that a program consists entirely of purely arithmetic and logic instructions or has no logic instructions.

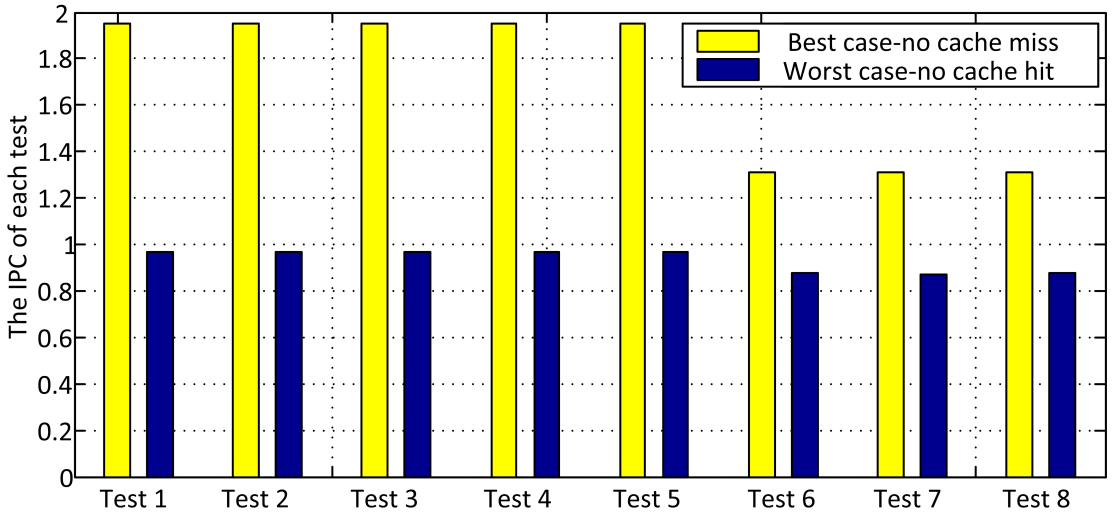


Figure 5.14: The IPC of the combined tests.

Figure 5.14 shows the IPC of each test in two cases: the best case and the worst case. For the best case, the loop size of each test is less than the L1 cache size and there are no cache misses. However for the worst case, the loop size is 64kB which is twice as large as the L1 cache size and nearly every instruction has to be fetched from the L2 cache. It is clear that the IPC of the best case is always bigger than the worst case. However, the IPC of each test in the best case is not the same. The IPC of tests 6, 7 and 8 is 1.31 but the IPC of the others is 1.97. The reason is explained by Table 5.11.

Table 5.11: The pipeline status of each test.

Test	Pipeline Stage							
1,2	Logic	Logic	Logic	Logic	Logic	Logic	Logic	Logic
	logic	L/S	Logic	L/S	Logic	L/S	Logic	L/S
3,4,5	Logic	Logic	Logic	Logic	Logic	logic	Logic	Logic
	L/S	L/S	L/S	L/S	L/S	L/S	L/S	L/S
6,7,8	Logic	-	-	Logic	-	-	Logic	-
	L/S	L/S	L/S	L/S	L/S	L/S	L/S	L/S

Table 5.11 presents the status of the pipeline stage in each test. It is clear that there are bubbles in the pipeline of tests 6, 7, and 8 but no bubbles in the other tests. The reason is that there is only one Load/Store pipeline in the processor and the 75% of the instructions in tests 6, 7 and 8 are Load/Store. However the instructions have to be implemented in the instruction order since the Cortex-A8 is an in-order superscalar

processor. Therefore, the bottle neck is the the speed of load/store. While the load/store pipeline is busy, the ALU logic pipeline has nothing to do and has to wait, which introduces bubbles in the ALU pipeline.

There are a lot of different reasons for a pipeline stall, such as cache misses and data dependency. However, no matter what the reason is, when the pipeline stalls, the processor has to wait and the instruction per clock cycle (IPC) becomes lower. Instead of considering each different factor individually, the IPC can be used as a variable to describe how smoothly a program runs and to reflect the effect of the pipeline stall rate and cache miss rate.

Table 5.12: The sample standard deviation (STDEVA) and margin of error (MOE) of the Cortex-A8 modeling test.

	tes1	test2	test3	test4	test5	test6	test7	test8
STDEVA	0.000626	0.001028	0.001123	0.005959	0.000544	0.00063	0.000943	0.001172
MOE(W)	0.000501	0.000822	0.000899	0.004769	0.000435	0.000504	0.000754	0.000938

Table 5.12 shows the sample standard deviation (STDEVA) and margin of error (MOE) of the Cortex-A8 Hamming distance power test. The test result will be used to generate the power model, thus we test it more times than before. Furthermore, we measure each test two times with different cache usage, which is set to 4kB, 8kB, 16kB. Thus, each test is measured six times in total. The MOE is calculated in a 95% confidence level.

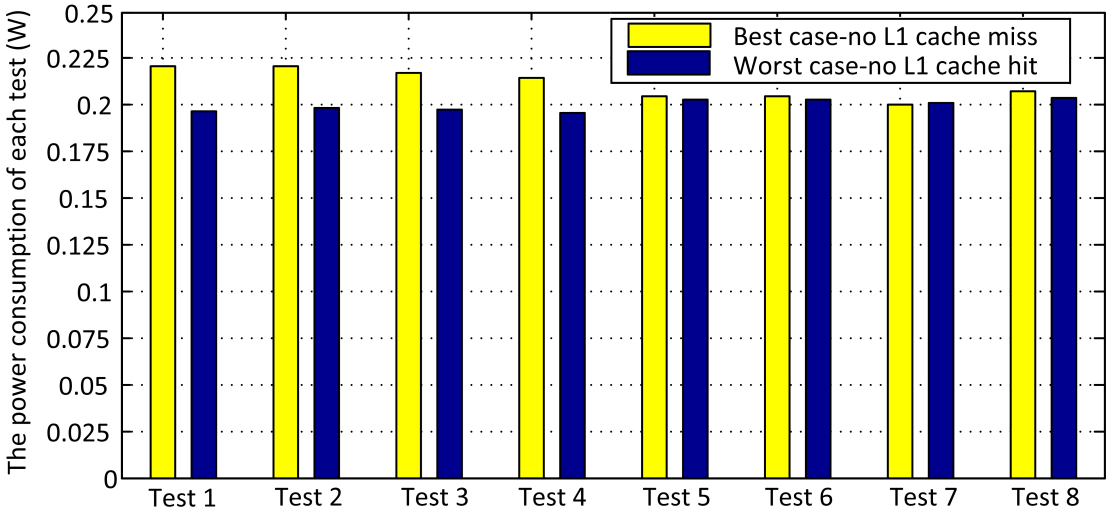


Figure 5.15: The power of the combined tests.

Figure 5.15 shows the power consumption of the combined tests. In the best case, test 1 consumes the most power, which is $0.2208W$, and test 7 consumes the least power, which is $0.2005W$ (a difference of 9.18%). In addition, the power consumption of the first four tests is greater than for the last four tests. The reason is the IPC of the first four tests is bigger than the last three tests. Although test 5 has a high IPC, which is 1.95, its power consumption is still lower than for the first four tests. The reason

is based on the test results in Section 5.3.3, store consumes the least power in all the different kinds of instructions (ignoring MUL) and 50% of the instructions of test 5 are store.

However for the worst case, test 8 consumes the most power, which is $0.204W$, and the least power consumption is $0.196W$ which comes from test 1 (a difference of 3.9%). This result is different to the best case. Moreover, the power consumption of the last four tests is bigger than for the first four tests. This result is the opposite to that in the best case. From the test results in Section 5.3.3, load and store consumes more power when L1 cache misses exist. The percentage of the load and store in the last four tests is higher than in the first four tests. Thus, the last four tests consume more power in worst case.

It is clear that the power consumption can be affected by both IPC and the components of a program, hence the power model has to consider both.

5.8.2 Instruction Level Model

From the results in Section 5.3.2 and Section 5.4.2, we see that the definition of the base power/energy cost of instructions explained in Section 2.1 does not work. The reason is that in both cases: two instructions may be run in parallel or one by one, but the power consumptions in these two cases are similar to each other. Furthermore, the power of the dual-issue tests in Section 5.3.2 and the dual-issue restricted tests in Section 5.4.2 are about $0.175W$ and $0.163W$, respectively, and the difference is only 6.85%. However, the IPC of the dual-issue tests is nearly equal to two but the IPC of dual-issue restricted tests is nearly equal to one. The different is almost 50%. Assuming that a program has 1000 instructions and all of these instructions are independent, the energy will be $0.175W \times 500 \times T_{clk}$. However, if these instructions meet the dual-issue restrictions, the energy will be $0.163W \times 1000 \times T_{clk}$. Thus, the energy consumed by these two programs will be very different and it is impossible to find the base energy for each instruction in a superscalar processor. Thus, a new model has to be created to predict the energy for a superscalar processor.

In order to derive a concise model, we divide the instructions into three classes: 1. arithmetic and logic instructions, 2. load, and 3. store. We assume the power is affected by both IPC and the components of a program. Therefore, power can be represented by the following equation:

$$Power_{average} = k_0 + k_1 \times p_{instruction\_distribution} + k_2 \times IPC \tag{5.1}$$

We have already analysed eight different tests with two cases. Therefore, we have eight different distributions, sixteen IPC values and the corresponding power consumption

(eight from the best case and eight from the worst case). Based on these results, we use linear regression to derive the model as follows:

$$\begin{aligned} Power_{average} =& 0.1842 + 0.0005 \times p_{ALU} + 0.0026 \times p_{Load} \\ &+ 0.0155 \times IPC, \end{aligned} \tag{5.2}$$

where $Power_{average}$, p_{ALU} and p_{Load} are the average power consumption, the ALU instruction percentage and the load percentage of a program, respectively. We assume that all of the instructions come from these three cases, thus $p_{ALU} + p_{Store} + p_{Load} = 100\%$. For the same reason discussed in Section 4.6, the p_{Store} is represented by p_{Logic} and p_{Load} after linear regression.

Energy is estimated from:

$$\begin{aligned} E =& \int_0^T P(t) \times dt \\ =& P_{average} \times T \\ =& T \times (0.1842 + 0.0005 \times p_{ALU} + 0.0026 \times p_{Load} \\ &+ 0.0155 \times IPC), \end{aligned} \tag{5.3}$$

where T is runtime of the program. The experimental platform uses Angstrom Linux as the operating system, thus it is easy to measure the runtime of the program.

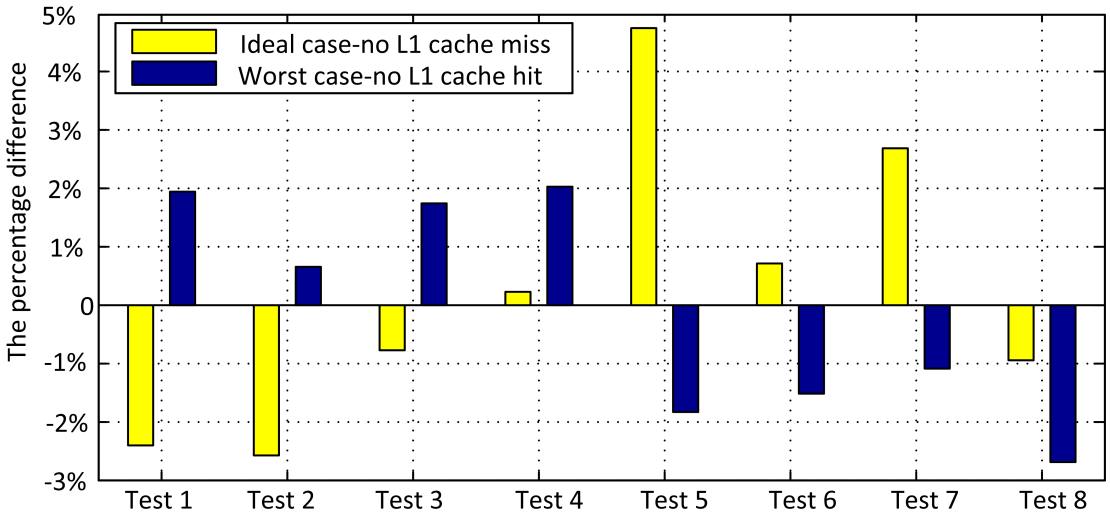


Figure 5.16: The estimation results of the combined tests.

Figure 5.16 shows the power consumption difference percentage between the model and the measured results. It is clear that all of the tests are estimated accurately, with the maximum error less than 5%. On top of this, for 93% of the estimations this model has less than 3% error. There are several factors which may induce these errors. For example, the measurement environment changes, such as the temperature of the processor. In order to avoid this, we measured ten instances of each test and used the

average value as the power consumption. The effect of the operating system (OS) and the small number of training tests may be the other reasons of the miss estimation. On top of this, because the architecture of the superscalar processor is much more complicated than of the single scalar processor, such as dual-instruction fetch, more ALU blocks, thus there may be some factors which can induce errors that are not considered well, such as level 2 cache misses. However, both overestimation and underestimation are acceptable and necessary because if all tests are over estimated or under estimated, it means something is over considered or less considered. Then, a constant factor would be added into the model to increase the accuracy.

5.9 Validation of the Power Model

Ten benchmarks: Stringsearch, Susan.corner, Susan.edges, Bitcount, Sha, Fibonacci, Tak, FIR filter, Quicksort and Tower of Hanoi were used to test the performance of the model. Table 5.13 shows the input value of each test.

Table 5.13: The input value of each benchmark.

Test name	Description of the input
Stringsearch	Default small test from Mibench
Susan.corner	Default small test from Mibench
Susan.edges	Default small test from Mibench
Bitcount	Default small test from Mibench
Sha	Default small test from Mibench
Fibonacci	Generate 25 Fibonacci number
Tak	Tak(3000,2,3)
Fir	4000 inputs number with 5 coefficients
Quicksort	4000 data from Mibench\ automotive\qsort\input_large.dat
Hanoi	9 discs

The components of each test are shown in Figure 5.17. The distribution was generated by the instruction simulator tool, gem5 [78]. However, the ARM performance counter also can supply this information. Therefore, the distribution is fast and easy to get. The stringsearch, susan.corner, susan.edges and Bitcount benchmarks were chosen from Mibench benchmark suites [110].

From Figure 5.17, the lowest IPC comes from Tak (0.66), while for Sha it is more than 1.40. The components of the different tests are also very different. For example, the logic percentage of Bitcount is more than 80% but for Fir is less than 60%.

Table 5.14 shows the sample standard deviation (STDEVA) and margin of error (MOE) of the Cortex-A8 benchmarks. We measured each test two times. However, if the difference is bigger than 5%, we measured another time. The MOE is calculated in a 95% confidence level.

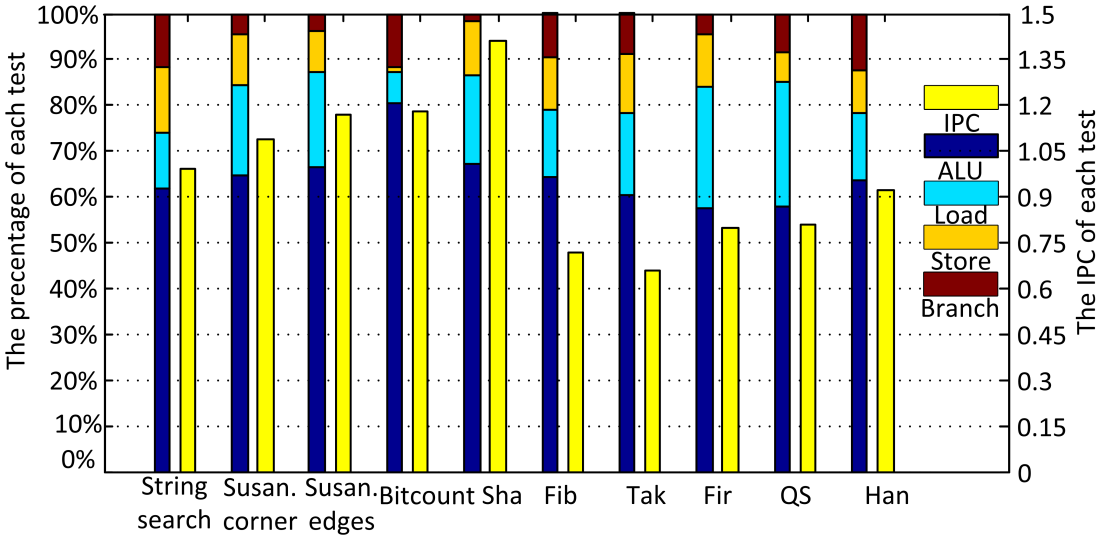


Figure 5.17: The components of each benchmark test program.

Table 5.14: The standard deviation (STDEVA) and margin of error (MOE) of the Cortex-A8 benchmarks.

	String.search	Susan.conrner	Susan.edges	Bitcount	Sha
STDEVA	0.000426	0.001913	0.000217	0.000201	0.000416
MOE(W)	0.000591	0.002652	0.000301	0.000278	0.000577
	Fib	Tak	Fir	QS	Han
STDEVA	0.00017	0.000199	0.000693	3.52E-05	0.000176
MOE (W)	0.000236	0.000276	0.000961	4.88E-05	0.000245

Figure 5.18 shows the measured power (column 1), estimated power from our model (column 2) and the difference percentage (curve). Quicksort and FiR consumes the least power, 0.1926W and 0.1932W respectively, and they have the lowest IPC compared with the others. Because of its high IPC, Sha consumes the most power, 0.215W and the maximum difference of the benchmark suite is 10.4%.

The maximum and minimum power estimation errors are -6.69% and 0% respectively. On top of this, the average absolute error is 3.33% and for 70% of the estimations this model provides less than 4% error.

5.10 Comparison and Discussion

As discussed above, our method is easily extended to an energy model. The runtime of a program can be achieved from the OS, thus the error in estimating the energy of a program is equal to the power estimation error. In this section, our model is compared with some previous power and energy models.

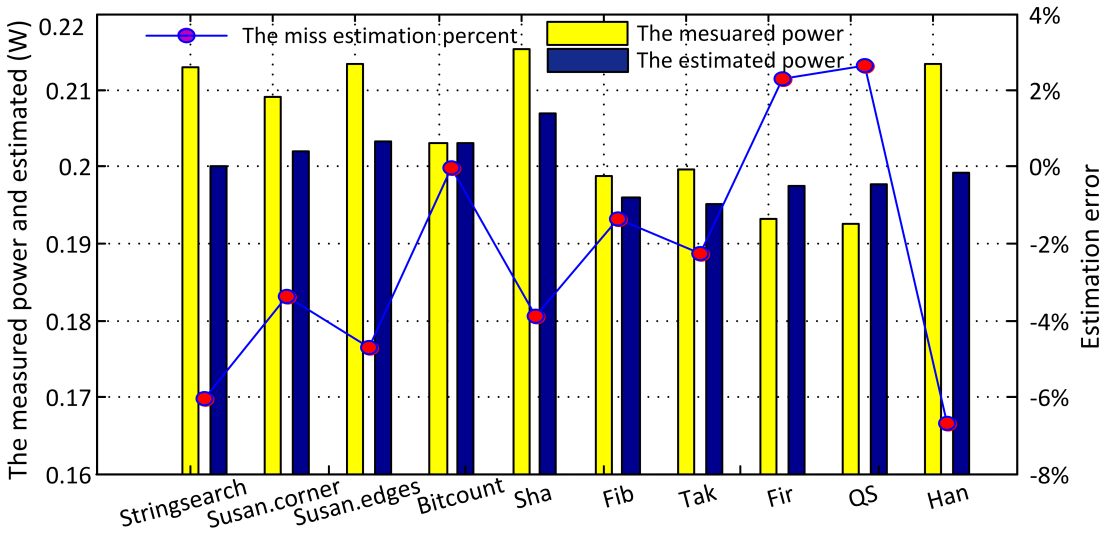


Figure 5.18: The error between predicted power and measured.

Table 5.15: Accuracy comparison with previous work.

	Our method	[5]	[11].model 1	[11].model 2	[12]	[104]	[15]
Fibonacci	-1.37%		15.58%	9.36%			
Fir	2.28%	-4.05			11.52%		
Quicksort	2.63%		11.41%	3%		8.98%	-5.97%
Bitcount	0%						-4.22%
Sha	-3.88%						5.77%
Stringsearch	-6.06%						3.03 %

Table 5.15 shows the comparison of accuracy between our method and previous work. The results come from the previous research and based on their own processors and testing systems (not repeated on our ARM Cortex-A8 system). Our method is more accurate than most of the others except for the Stringsearch test presented in [15]. There are several possible reasons for the errors, such as the effect of the OS and the different hardware usage. Even when different programs have the same input values of our model: IPC, and the distribution of instructions, they may still have different hardware usage. For example, there are many factors that can make the pipeline stall, such as cache miss, or data dependency. Therefore, the power/energy consumption can be different.

We try to compare our method to the methods presented in Table 5.15, however, none of these models is based on a superscalar processor and they are based on the basic instruction level model and single pipeline processors. We have shown that for a superscalar processor, the definition of the base energy does not work in Section 5.8.2. Thus, we cannot compare those models in Table 5.15.

On the other hand, Jeffry *et al.* created a simple average power model for superscalar processors 80960JF and 80960HD [66]. In this model, the power is modelled with a constant parameter: the average power consumption of each type of instructions. Thus, the energy consumption of the program is only related with the runtime of the program.

Based on the data from Figure 5.4, and Figure 5.5, the average power consumption of each type of instruction is $0.1733W$.

However, from Figure 5.5, it is clear that the instruction ALU, Load, and Store consume different power. It is not suitable to the assumption in [66]. Thus, we modify this model as the following equation:

$$\begin{aligned} P_{average} &= \sum_i P_i \times p_i \\ &= P_{ALU} \times p_{ALU} + P_{Load} \times p_{Load} + P_{Store} \times p_{Store}, \end{aligned} \tag{5.4}$$

where $Power_{average}$, P_{ALU}, P_{Load}, and P_{store} are the total average power consumption, the power consumption of the ALU instruction, Load, and Store, respectively. p_{ALU}, p_{Load}, and P_{Store} are the ALU instruction percentage, the Load percentage, and the Store of a program, respectively.

On the other hand, the overhead power consumption between different instruction classes could be still important. Although the definition of the base energy cost is not available anymore because of the dual-issue constraints, we can assume that the overhead cost is the power different when instructions are always run in parallel. Thus, similar with Equation 4.15, the following equations can be derived:

$$\begin{aligned} E_{overhead} &= ((P_{logic\_load} - \frac{P_{logic} + P_{load}}{2}) \times p_{load} + (P_{logic\_store} - \frac{P_{logic} + P_{store}}{2}) \times p_{store}) \\ &\quad \times 2 \times T_{clk} \times T \times F \times IPC \\ P_{overhead} &= ((P_{logic\_load} - \frac{P_{logic} + P_{load}}{2}) \times p_{load} + (P_{logic\_store} - \frac{P_{logic} + P_{store}}{2}) \times p_{store}) \\ &\quad \times 2, \ \times IPC \end{aligned} \tag{5.5}$$

where P_{logic_load} is the average test to run instruction pairs *Logic* and *Load*. p_{load}, p_{store} are the percentage of instruction *Load* and *Store* in the program.

Table 5.16: The estimation of the modified constant power model (Equation 5.4) and overhead power cost(W).

module name	String.search	susan.conrner	susan.edges	Bitcount	Sha
The modified constant model	0.1742	0.1752	0.1754	0.1753	0.1751
The overhead power	0.0198	0.0234	0.0225	0.0062	0.0233
	Fib	Tak	Fir	Qs	Han
The modified constant model	0.1748	0.1748	0.1756	0.1761	0.1750
The overhead power	0.0196	0.0232	0.0288	0.0258	0.0180

Table 5.16 shows the power consumption of the modified constant model (Equation 5.4) and the overhead. The data of P_{ALU}, P_{Load}, and P_{store} is fromFigure 5.5, and the data of p_{ALU}, p_{Load}, and p_{store} is from Figure 5.17. The data of P_{logic_load}, and P_{logic_load} is from test3 and test5 in Figure 5.13 in Section 5.8.1.

Table 5.17: The errors of the constant power model ([66]) and our method.

Power Method	Stringsearch	susan.conrner	susan.edges	Bitcount	Sha
The original constant model	-18.68%	-17.10%	-18.77%	-14.69%	-19.52%
The modified constant model	-18.23%	-16.21%	-17.76%	-13.72%	-18.69%
The modified constant model2	-8.96%	-4.99%	-7.2%	-10.73%	-7.82%
Our method	-6.06%	-3.36%	-4.72%	-0.06%	-3.88%
	Fib	Tak	Fir	Qs	Han
The original constant model	-12.84%	-13.25%	-10.31%	-10.05%	-18.81%
The modified constant model	-12.09%	-12.49%	-9.11%	-8.55%	-18.03%
The modified constant model2	-2.21%	-0.87%	5.78%	4.85%	-9.58%
Our method	-1.37%	-2.28%	2.28%	2.63%	-6.69%

Table 5.17 shows the errors of the original constant power model presented in [66], the modified constant power model, the modified constant power model2 (modified model1+overhead power), and our method, respectively. It is clear that neither the original power model nor the modified1 model apply to the ARM Cortex-A8. However, after considering the power consumption effect of different instructions (the overhead power), the modified constant model has a better performance. On top of this the best performance is still our model.

Furthermore, the average absolute error of the original constant power model, the modified model1, the modified model2 one, and our method is 15.41%, 14.49%, 6.3%, and 3.33% respectively. The reason of this mismatch could be that the effect of cache misses for different instructions is not the same, which is presented in Figure 5.5. The constant power model does not consider the effect of the cache miss.

Our model has shown a good estimation base on the ten benchmarks, and there are three main reasons for this:

1. The Cortex-A8 uses a lot of low power design methods, which make the power consumption low and stable. For example, the L1 cache misses cannot affect the power very much (the difference between the average power consumption of the 4kB and 36kB is 3.07%). L2 cache misses are more rare because of the branch prediction and the instruction pre-fetch strategy.

2. The power model shows good performance, as shown above, because it is easier to create than an energy model. On the other hand, in our method, part of the input data (the runtime of a program) comes from measurement rather than modelling, which makes the overall estimation better than for a pure energy model. The comparison results also show the advantage of our method which divides a complex question (energy model) into two simple questions: a power model and the runtime of a program. The runtime of a program is one of the easiest variables to measure.

3. The power model considers the effect of cache misses and pipeline stalls. Data dependency and cache misses can stall the pipeline, hence will affect the energy and

power consumption of a program. We take IPC into account for these factors and make the model concise but accurate. However, these factors are not considered very much in [5, 11].

Compared with other models, as the discussion in Section 4.8.1 shows, one of the benefits of this method is that the overhead power/energy of two adjacent instructions does not need to be considered separately. We do not need to create extra test to measure this effect.

Another benefit of our model is that it is easy to create. In order to generate the power model, we use sixteen tests (eight group of tests and each group has two different cases) as the training tests to achieve a model with an average absolute error of 2.7%. However, sophisticated design of training tests is required for the energy model of the Mep processor because it has to take the standard deviation of each variable value into consideration [103]. The minimum and maximum errors of that model are 2% and 16%. Bazzaz et al. created a model for the ARM7TDMI with 60 specialized tests which are used to analyse the coefficients of each energy sensitive factor. On top of this, the model has 35 parameters as the input data including: register bank bit flip, instruction word Hamming distance and the ARM7 instruction set [15].

5.11 Discussion: Low Energy Software

Based on the power and energy model, we will discuss how the power model might be applied to writing low energy software for a superscalar processor. Combining the EPI Equation 4.20 and the superscalar power model (Equation 5.2) yields

$$\begin{aligned} EPI =& \frac{P}{IPC \times F} \\ =& \frac{0.1842 + 0.0005 \times p_{ALU} + 0.0026 \times p_{Load} + 0.0155 \times IPC}{IPC \times F} \\ =& \frac{C_1}{IPC \times F} + \frac{0.0155}{F}, \end{aligned} \tag{5.6}$$

where C_1 is $0.1842 + 0.0005 \times p_{ALU} + 0.0026 \times p_{Load}$. Therefore, the EPI is inversely proportional to IPC. This result is the same as the ARM11 processor (Section 4.8.3).

Table 5.18: The benchmarks ranked by EPI and IPC.

	Stringsearch	Sussan.corner	Susan.edges	Bitcount	Sha	Fib	Tak	Fir	QS	Han
EPI	6	7	8	9	10	1	2	3	4	5
IPC	5	4	3	2	1	10	9	8	7	6

Figure 5.19 shows the energy per instruction and IPC of each test, and Figure 5.18 ranks the workloads by EPI and IPC in detail. They demonstrate the conclusion proposed by

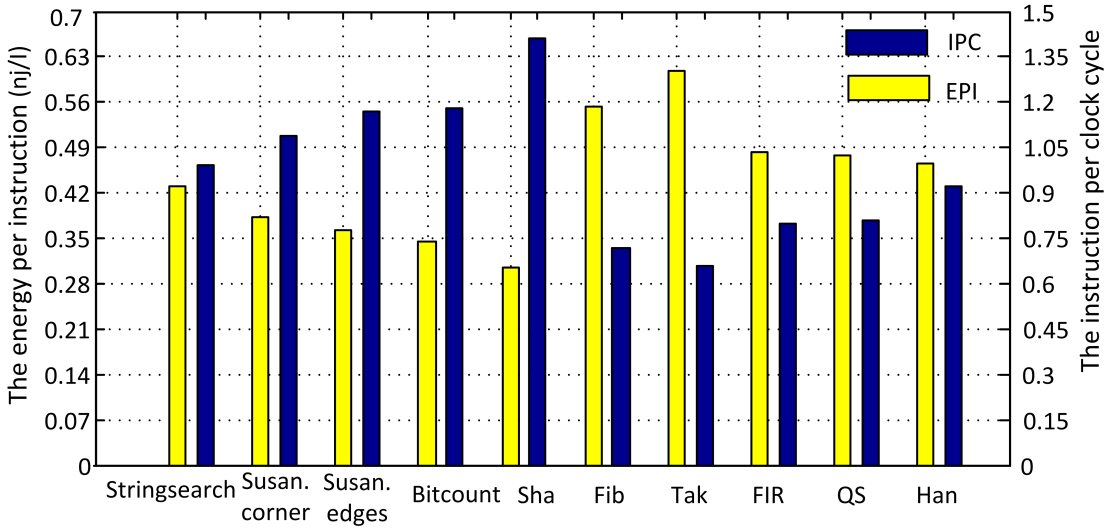


Figure 5.19: The energy per operation VS operation per clock cycle.

Equation 4.21 and Equation 5.6. Sha has the fastest IPC but has the lowest EPI. In contrast, Fib is the slowest but has the biggest EPI. Thus, increasing the IPC without changing the size of a program much may reduce the EPI.

5.12 Conclusion

In this chapter we present an instruction-level power model for an in order superscalar processor ARM Cortex-A8. Firstly, we analyze the power consumption of the processor under various conditions, including: how the power consumption of a processor is affected by L1/L2 instruction and data cache misses; by different instruction types; by dual-issue restriction; by the Hamming distance between the operands of two consecutive instructions; and by the overhead power cost of two adjacent instructions.

We show that the power is related to both the instructions per clock cycle (IPC) and the instruction types of the program. Pipeline stalls are modelled by the IPC instead of the cache miss rate. We extend the power model to estimate energy. The performance of the model is tested in several embedded applications from the MiBench benchmark suite. The results show the maximum estimation error is 6.69% and the average absolute estimation error is 3.33%.

In the power model, instead of studying each instruction individually, the instructions are divided into three types: arithmetic/logic instructions, load and store. Moreover, we find that the speed of a program can affect the power and we take the IPC into consideration. The IPC can reflect the factors which can make the pipeline stall, such as cache miss and data dependency. This model is very concise because it does not consider the overhead energy as an independent fatcor or the operand Hamming distance of two consecutive instructions.

On top of this, we extend the power model to a method of estimating the energy. Compared with energy model, this method has two advantages: it is easy to create and is accurate. Finally, we show that energy per instruction (EPI) is inversely proportional to the instructions per clock cycle (IPC).

Chapter 6

ARM Cortex-A9 Dual-core Processor

As the discussion in the last chapter shows, a superscalar processor has a better performance than a scalar processor. However, it has several disadvantages which are hard to solve.

Firstly, instruction level parallelism (ILP) has it own disadvantages. For example, the multi-issue design increases the complexity of the instruction decoding stages in the front end of the pipeline. In order to achieve higher parallelism, the fetch width could become bigger. However, the fetched instructions could have more than one branch or have data dependencies. In order to make the pipeline usage more efficient, more complex designs have to be developed for the front end of the pipeline. Thus, the pipeline design becomes more and more difficult, since the instruction fetch width increases.

Secondly, it is hard to achieve high speed performance via high instruction level parallelism. The reason is Read after Write (RAW) data dependencies are a the bottle neck in a program, which limits the IPC of the program. For example, if a program contains 100 instructions and 40 of them have internal data dependencies, at least 40 clock cycles are needed to finish, no matter how big the instruction fetch width is, or how many ALUs there are.

Thirdly, the clock frequency of single core is close to the pipeline limits. In order to achieve better performance, one solution is to increase the clock speed. However, the clock speed is also limited by physical materials and a high clock speed also means high power and temperature. Nowadays, low power has become more and more important. The power grows super linearly with higher clock speed and more complex Out of Order logic design.

Between 2000 and 2005, designers attempted to exploit more ILP, but it turned out to be inefficient. The reason is the power and silicon costs grew faster than performance [21]. However, thread-level parallelism (TLP) is another way to increase performance.

On the other hand, multiprocessing is more and more important and reflects several major factors [21]:

- A increasing interest in high-end servers such as cloud computing.
- A growing interesting in data-intensive applications related to massive amounts of data.
- A better understanding about how to benefit from multiprocessors effectively, such as server environments where there is significant natural parallelism.

In order to get better performance in TLP, multi-core processors have been designed. Compared with ILP, parallel program techniques can benefit from TLP directly, such as pthreads. TLP could be more cost-effective because a thread has its own instructions and data. There are a lot of applications where thread-level parallelism occurs naturally, such as server applications [102].

6.1 Target Processor

The Cortex-A9 processor is designed to maximize performance while considering the price sensitivities of embedded devices. Firstly, for low power consumption, the power efficiency is increased with higher performance. Secondly, for most applications, the peak performance is increased. Lastly, it has the ability to share tools and investments in software with different products. Thus, it can make a good solution for any design requiring high performance in cost sensitive, lower-power, single processor-based design. Compared with the existing ARM11 processor, it can provide better performance with similar silicon cost and power budget [111].

Figure 6.1 shows a top level diagram of the Cortex-A9 processor. It contains six main stages: instruction prefetch stage, dual-instruction decode stage, register rename stage, dispatch stage, execute stage and OoO write back stage. The following is the pipeline description [111]:

1. Instruction prefetch stage: up to four instruction cache lines can be prefetched, which unblocks the potential memory latency caused by branches.
2. Dual-instruction decode stage: two full instructions can be decoded per cycle and dispatched into the register rename stage.

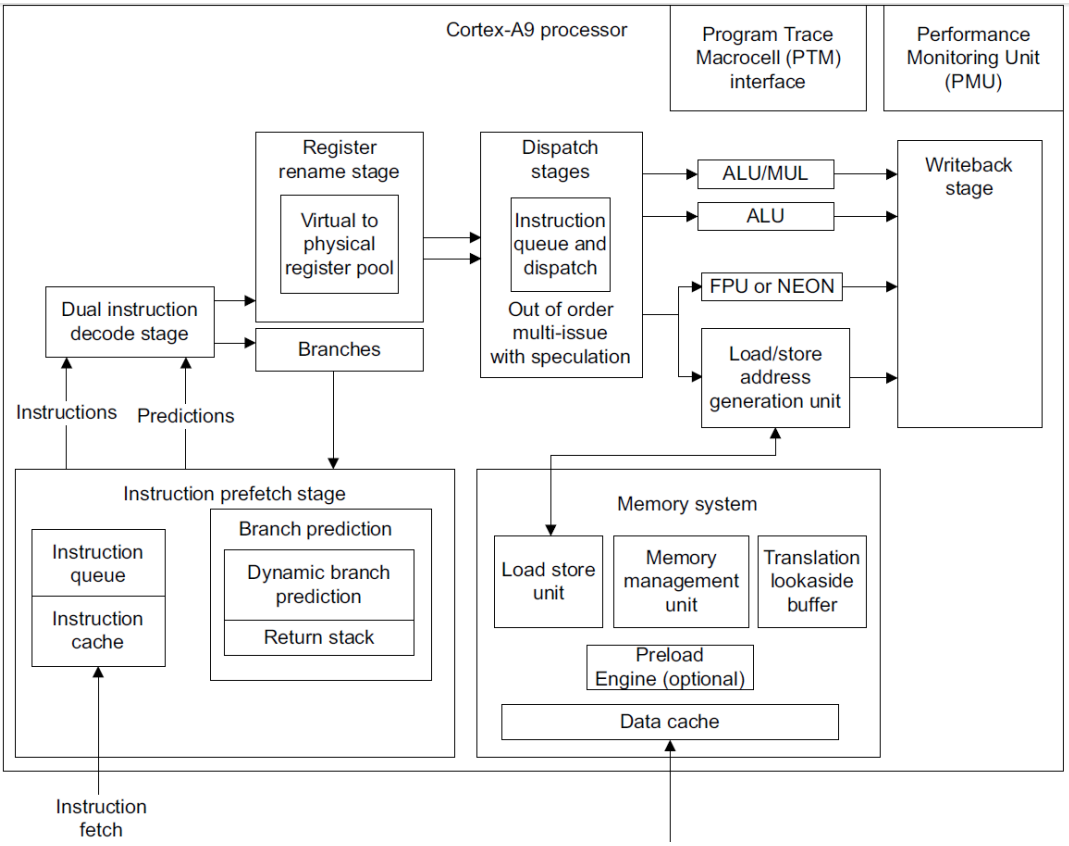


Figure 6.1: The architecture of Cortex-A9 [112].

3. Register rename stage: renames physical registers into a virtual register pool. This stage is used to remove the data dependency between consecutive instructions.

4. Dispatch stage: dispatches instruction to four pipelines (ALU/MUL, ALU, FPU/-NEON, Address). Once the resources of an instruction are ready, it will be dispatched into one of the execute pipelines according to its opcode type.

5. Execute stage: any of the four pipelines (ALU/MUL, ALU, FPU/NEON,Address) can select instructions from the issue queue. The selected instruction are out of order and this increases the pipeline utilization.

6. OoO write back stage: the order in which pipeline resources are released is independent of the order in which the data are provided by system.

For an in-order processor, after a instruction is fetched, the instruction is dispatched to the appropriate functional block only when the input operands are available, otherwise the pipeline will stall and wait until they are available. However, for an OoO processor, instructions will be dispatched to an instruction queue rather than functional blocks. Once the inputs of any instructions in the queue are ready, they will leave the queue and be sent to the appropriate functional unit to execute. Thus, out of order means the execution sequence does not have to be the same as the instruction fetch sequence. The

advantage of OoO design is it avoids a class of pipeline stalls and makes the pipeline usage more efficient.

In order to achieve an OoO design, data hazards have to be considered and one solution is dynamic scheduling with register renaming. Furthermore, the method is to rename all destination registers, such as write for an earlier instruction, and make the out of order write without affecting any instructions that relate to an earlier value of an operand [21].

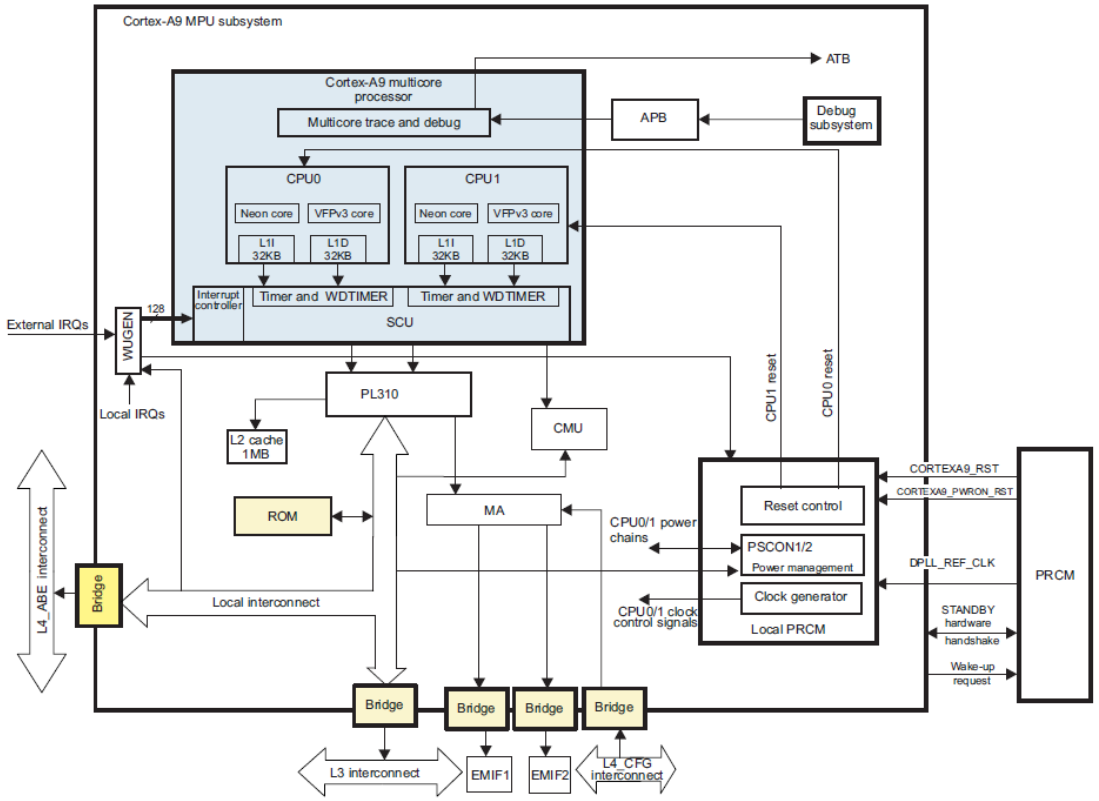


Figure 6.2: The multi-core processor system diagram [113].

Figure 6.2 shows the architecture of the dual-core processor. It is based on a symmetric multiprocessor (SMP) architecture, which means both of the cores have the same architecture [113]. The dual Cortex A9 system includes two Cortex-A9 processors, one L2 cache shared between the two CPUs, and one PLZ310 as an L2 cache controller. On the other hand, each processor has its private 32kB L1 instruction and data cache and a separate, dedicated power domain. Compared with the Cortex-A8, the dual-core processor can run up to four instructions in one clock cycle.

The following are the parameters of Texas Instruments omap4460 [112, 114]:

- 45-nm technology;
- SMP architecture;
- Superscalar, dynamic multiple issue technology with an efficient 8-stage pipeline;

- Out-of-order(OoO) instruction dispatch and completion;
- 32kB L1 instruction and 32-kB L1 data cache-32-byte line size, 4-way set associative;
- Memory management unit(MMU);
- PL310 L2 cache controller with 1-MB cache size with 16-way set associative and 32-byte line size.

6.2 Experimental Methodology

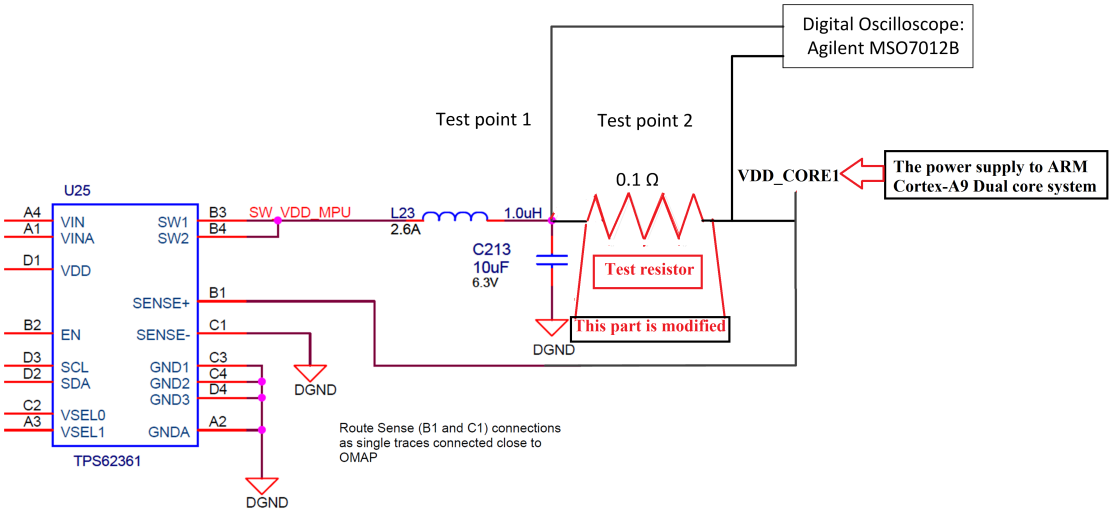


Figure 6.3: The power supply schematic diagram [115].

Figure 6.3 shows the original power supply of the ARM Cortex-A9 dual core processor and how we modified it. To make the necessary power measurements, a 0.1Ω series resistor was included between the power supply and the CPU. Compared with previous tests, the value of the resistor is the lowest because the current of the Panda board is nearly ten times bigger than the Beaglebone board at nearly $1A$. A digitizing oscilloscope, the Agilent MSO7012B, with a sample rate of 2GHz was used to measure the instantaneous power as tests were carried out. We used two probes to measure each side of the resistor. The instant power model, the average power model and the total energy model is the same as Equations 4.1, 4.2, and 4.3, which are defined in Section 4.2.

6.3 Instruction Level Power Model Analysis for a Dual-core

As discussed in Chapter 5, the power is related to the speed of the processor and uses the parameter IPC to reflect the speed. However, for a dual-core processor, the parallel ratio is also important.

Firstly, it will affect the speed of a program. Amdahl's law illustrates how the parallel ratio can affect the speed of a program [10] and it proves that the bottleneck of the speedup of an application is the part of a program which cannot run in parallel. The modern form of Amdahl's law is

$$Speedup = \frac{1}{(1-r) + \frac{r}{n}}, \tag{6.1}$$

where r is the fraction that can run in parallel and n is the number of cores which run r.

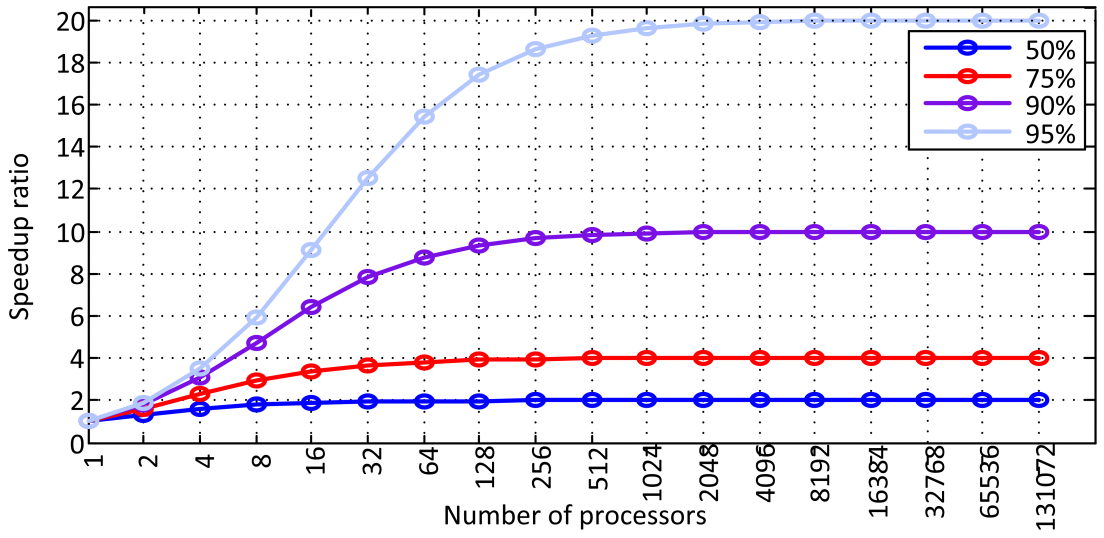
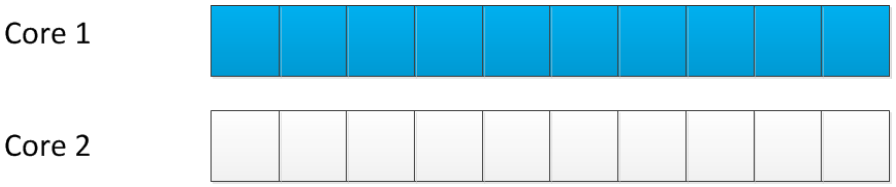


Figure 6.4: Amdahl's law: parallel speedup vs sequential fraction.

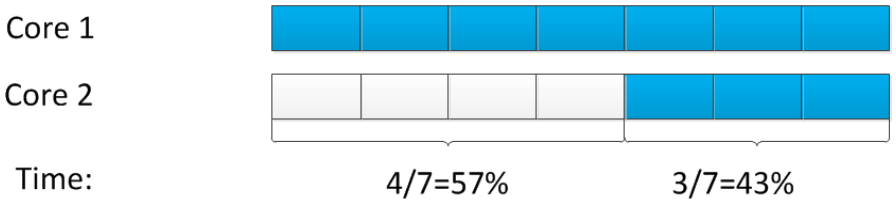
Figure 6.4 shows the relationship between the speedup and the sequential fraction. It is clear that the fraction limits the speedup of a program although there are a lot of processors. For example, if 60% of a program can run in parallel and there are two CPU cores, the speedup will be $Speedup = \frac{1}{(1-60\%)+\frac{60\%}{2}}$=1.43. No matter how many processors there are, the maximum speedup is less than 2. Thus, the speed of a program can be affected by the parallel ratio.

On top of this, the parallel ratio will affect the power of the processor. The reason is that the more cores are active, the more power will be consumed, assuming the same clock speed and supply voltage in each core. Considering that our target processor is a dual core system, the parallel ratio is the factor which reflects the percentage of the

runtime if only one core runs or both cores run. In the previous example, assuming 60% of a program can run in parallel, for 57% of the runtime, only one core works and the other core is idle. For the other 43% of the runtime, the two cores are used together. This can be explain by Figure 6.5.



(a) The case of a single thread program.



(b) The case of a multi-thread program.

Figure 6.5: The parallel ratio will affect the processor usage.

Moreover, the $speedupratio - 1$ represents what percentage of the time the second core is used during the program runs. The reason can be explained by Figure 6.6. From Figure 6.6, assume r is the part of a program that can be run in parallel and there are two cores in the processor. Thus, the time percentage of two core working is $P_{two\_core} = \frac{\frac{r}{2}}{1-\frac{r}{2}}$ and the following equation shows the relationship:

$$\begin{aligned} P_{two\_core}e =& \frac{\frac{r}{2}}{1-\frac{r}{2}} = \frac{r}{2-f} = \frac{2}{2-r} - 1 \\ =& \frac{1}{(1-r+\frac{r}{2})} - 1 = Speedup - 1 \end{aligned} \tag{6.2}$$

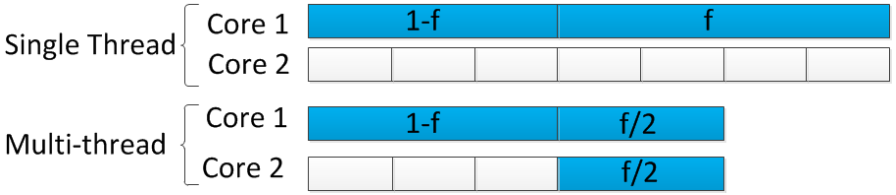


Figure 6.6: The speedup ratio and the second core usage.

Therefore, for a multi-core processor, the parallel ratio can affect both the speed and average power of a program. A power model has to take both these two factors into consideration.

The third factor is the components of each program. As in the discussion in Chapter 5, although two programs may have same parallel ratio and IPC, the power usage may be different if these two programs have different instruction components.

Hence, we derive a model assuming that the average power of a program is related to the parallelism ratio, the IPC and the components of the program.

6.4 Experimental Design

Based on the test results in Chapter 5, we assume that all the different logic and arithmetic instructions consume similar power. Thus, different logic and arithmetic instructions are not considered individually and instructions are classified into three cases: ALU, load, and store. In order to analyse how the characteristics of a program (the components of a program, the parallel ratio and IPC) affect the power, 96 different tests are designed. The main body of each test is a loop but the components and the loop size are changed. Furthermore, based on the different components of a program, the tests are divided into eight different cases. Then, the speedup ratio of a program is divided into six levels. Each of them is also studied in the best case and the worst case, where the L1 cache always hits or misses ($8 \times 6 \times 2 = 96$).

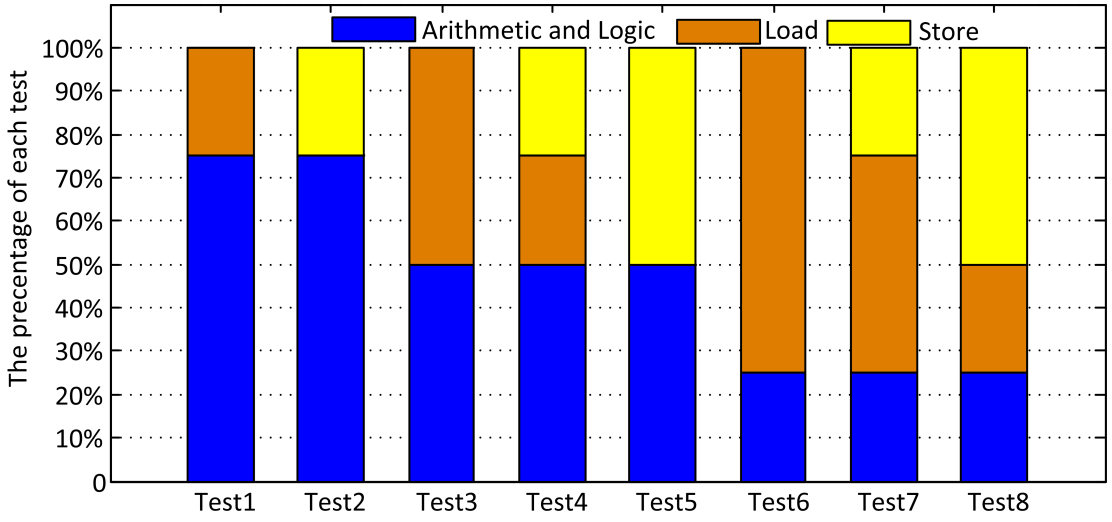


Figure 6.7: The components of the combined tests.

Firstly, eight tests which have different components are shown in Figure 6.7, which is the same as the combined tests in Section 5.8.1.

Secondly, we use the speedup ratio in Equation 6.1 as the parameter to reflect how much of the program can be run in parallel. The speedup ratio describes how many cores are used on average. Each of the eight tests is divided into six different further levels based on the speedup ratio. The speedup ratios were set to 1, 1.2, 1.4, 1.6, 1.8, and 2; This is set by changing the percentage of the instructions which can be run in

parallel. For example, if the speedup ratio is one, it means the program is a purely single thread program. If the speedup ratio is 1.25, it means 1.25 cores are activity on average and 40% of the instructions of a program can be run in parallel with two cores ($1.25 = \frac{1}{(1-40\%)+\frac{40\%}{2}}$).

Lastly, as the previous research shows, the cache hits and misses will affect the power. In order to test the effect, all of the tests are broken down into two cases: the best case and the worst case. For the best case, the length of each program is less than half of the L1 cache size, which makes sure all of the instructions and data can be found in the L1 cache and there are no cache misses. However, for the worst case, the size of each program is twice as much as the size of the L1 cache. Thus, nearly every instruction will be fetched from the L2 cache.

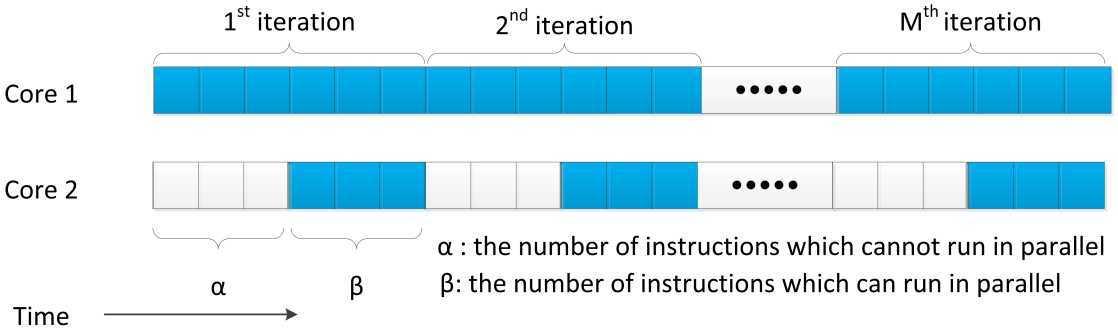


Figure 6.8: The speedup ratio and the second core usage.

Figure 6.8 shows the structure of the test program. A coloured box or a blank box shows whether the core is active or not, respectively. For each iteration of the test, the first part is a single threaded program, thus only one core is active. The second part is a multi-threaded function, which can be run in parallel and two cores are active. The components of the two parts are the same. The speedup ratio is set by changing the instruction numbers of the first part. The following code is an example of a test:

```
#include <stdio.h>
#include <stdlib.h>
#include <pthread.h>

#define THREADS 2
#define N 20000  // the loop number of the second part (multi-thread function )
#define LOOP_TIME 1000  // the total loop number of the test.

void *multi(void *val)  // the multi-thread part, which is a loop
{
    int thread_id = (int) val;
    int i;
    for (i = thread_id; i<N; i += THREADS)
    {
        asm (" sub r4 , r1,  #0xf  ");
```

```
        asm (" orr r1 , r1,  r2 ");
        asm (" sub r4 , r1,  #0xf  ");
        asm (" eor r6 , r2,  #0xf3  ");
        ............
        asm (" ADD r3 , r1,  #0xf");
    }
}

void main()
{
    pthread_t threads[THREADS];
    int i;
    int k;
    int M=LOOP_TIME;

     while(M>0)
    {
    for (k= 0; k<N/8;k++ )
    // the single thread part, which is a loop
    // the loop size depends on the test( the best case or the worst case)
    // the bigger the k is, the more instructions are run in single thread.
        {
        asm (" sub r4 , r1,  #0xf  ");
        asm (" orr r1 , r1,  r2 ");
        asm (" sub r4 , r1,  #0xf  ");
        asm (" eor r6 , r2,  #0xf3  ");
        ............
        asm (" ADD r3 , r1,  #0xf");
        }
    // k is related with N.

      // call the multi-thread function
    for (i = 0; i < THREADS; i++)
         {
            pthread_create(&threads[i], NULL, multi, (void*) i);
    /* Create independent threads each of which will execute function */
         }
        for (i = 0; i < THREADS; i++)
        {
            pthread_join( threads[i], NULL);
        }
   /* Wait till threads are complete before continues. */
           M--;
    }
}
```

6.5 Experimental Results

6.5.1 The Test Results of the Best Case

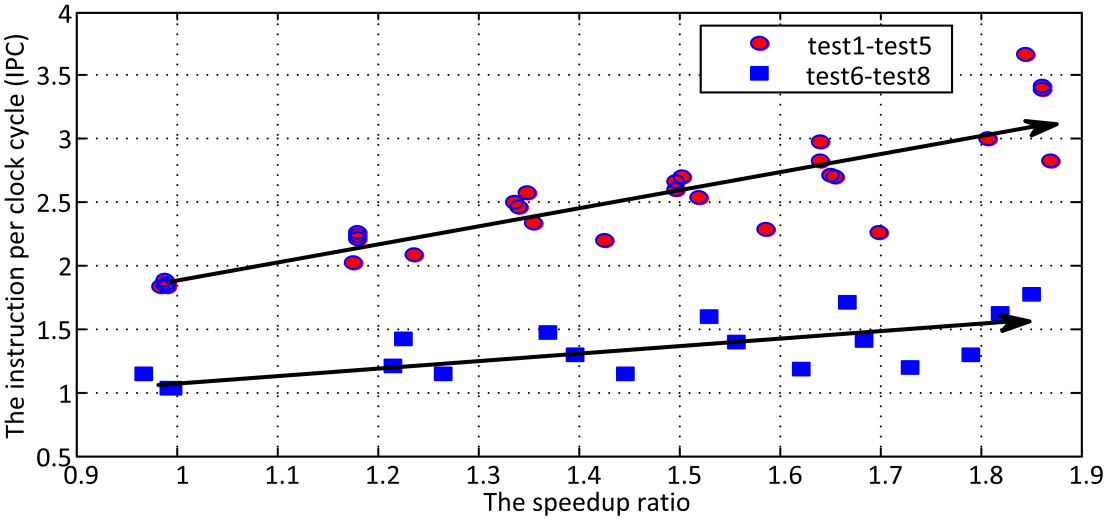


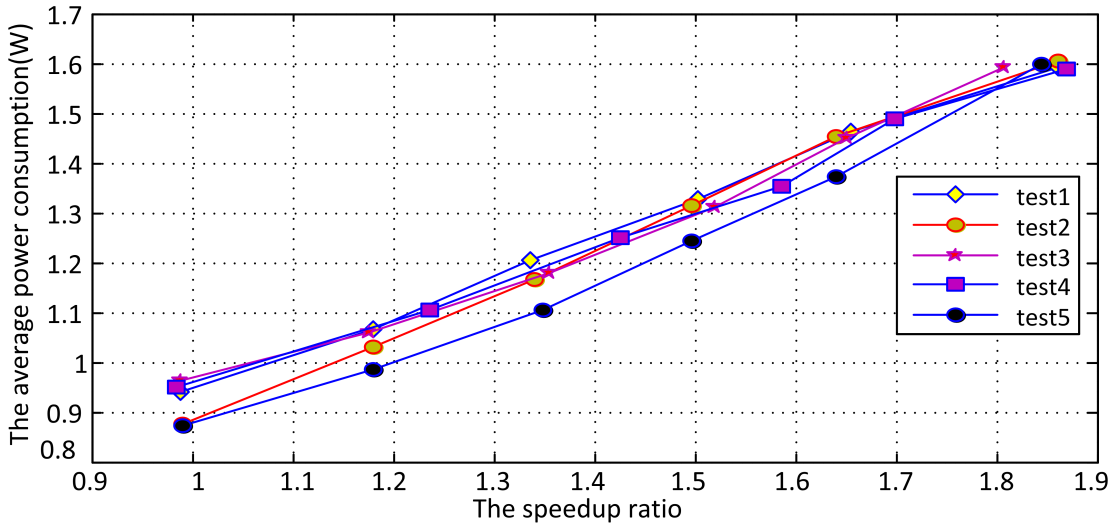
Figure 6.9: The instruction per clock cycle for test 1 to test 8 in the best case.

Figure 6.9 shows the instruction per clock (IPC) of each test with respect to the speedup and L1 cache hits (the best case). The measured speedup ratio is not the same as the ideal setting. The reason may be the effect of the OS and I/O peripherals. It is clear that the IPC is proportional to the parallel speed in all of the eight tests. Test 1 to test 5 have similar gradients (IPC vs the speedup ratio), as do the last three. However, the gradient of test 1 to test 5 is bigger than test 6 to test 8. Moreover, the top speed of test 1 to test 5 (when the speedup ratio is the biggest) is about 3.4 but for test 6 to test 8, it is only about 1.6. The reason is there are two ALU pipelines in the Cortex-A9 but only one load/store pipeline. Thus, when the load or store instructions are too many to consume by the load/store pipeline, the ideal IPC of each core will be less than 2. This is explained by Table 5.11 in Section 5.8.1. Thus, the gradient of test 1 to test 5 is bigger than that of test 6 to test 8.

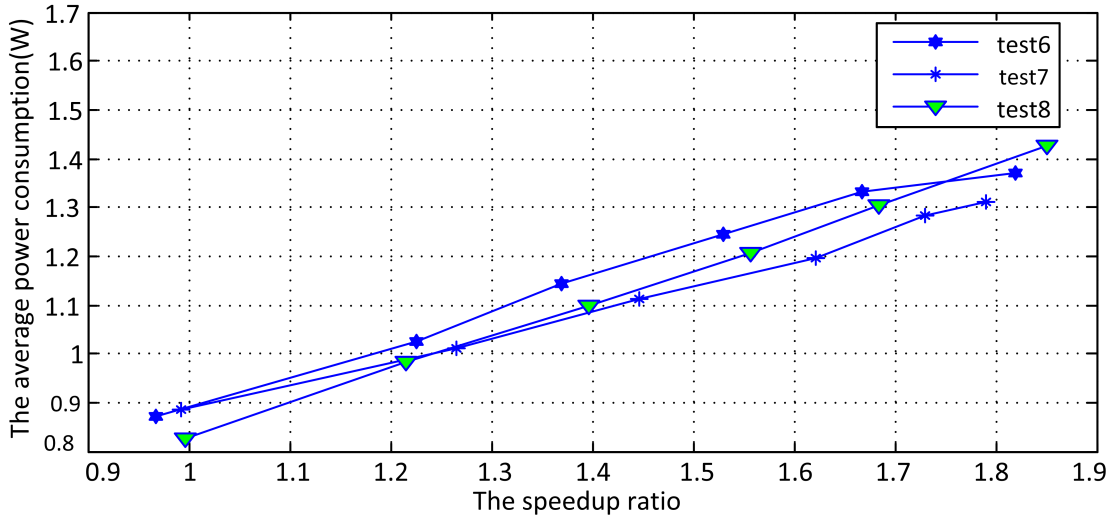
Figure 6.10 shows the power consumption of the eight tests with changing speedup ratio in the best case. The power consumption is proportional to the speedup in all of the eight tests. The power is also affected by the components of the tests but by not much.

Figure 6.10(a) shows the power consumption for test 1 to test 5 in the best case. The power consumption of test 5 is a little smaller than the others but by not much. Thus, they still have similar gradients (power vs the speedup ratio). Similarly, Figure 6.10(b) shows that the power of test 6 to test 8 in the best case are close to each other when they have a similar speedup ratio, thus they also have similar gradients.

However, the peak power (when the speedup ratio is the highest) and the gradients of test1 to test 5 are not the same as for test 6 to test8. For example, test 1 to test 5



(a) The average power consumption of test 1 to test 5.



(b) The average power consumption of test 6 to test 8.

Figure 6.10: The power consumption of the eight tests in the best case.

consume similar peak power, which is around $1.6W$ (Figure 6.10(a)). In contrast to this, the peak power of test 6 to test 8 is only about $1.4W$ (Figure 6.10(b)). The reason for the difference is that the top speed (IPC) is different.

Figure 6.11 shows the speedup ratio, IPC and the average power consumption of test 1 to test 8 together. Firstly, it shows that both the speedup ratio and the IPC can affect the power. For example, when different programs have similar speedup ratios, the power consumption will be different if they have different IPCs, such as test 1-test 5 compared with test 6-test 8. Secondly, both speedup ratio and IPC have positive effects. The bigger the speedup ratio and IPC, the more power will be consumed. Moreover, the IPC has a relation with the speedup ratio, since if the speedup ratio becomes higher,

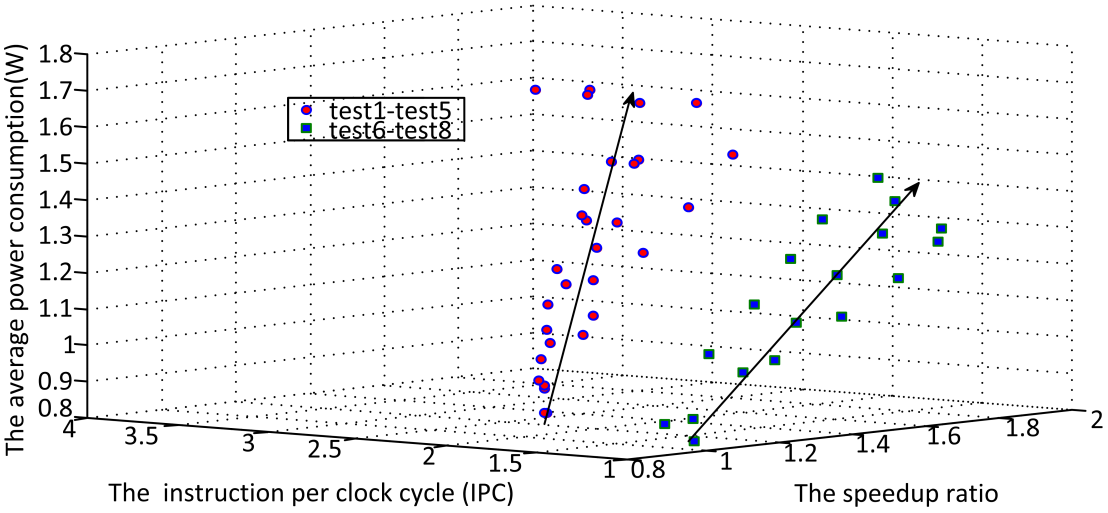


Figure 6.11: The speedup ratio, ICP and average power consumption of test 1 to test 8 in the best case.

more instructions can be run in parallel. Thus, the IPC has a high chance to become bigger.

6.5.2 The Test Result of the Worst Case

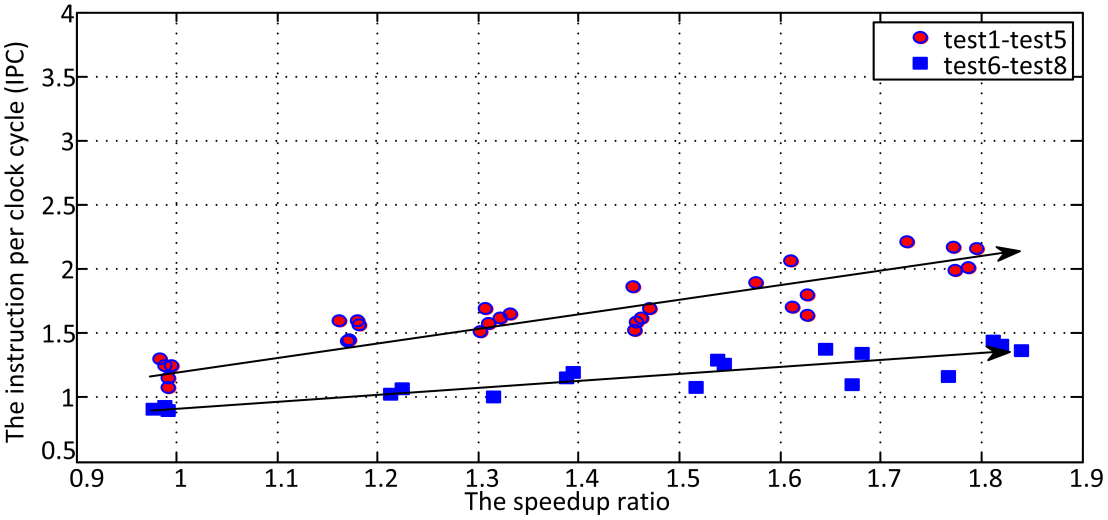
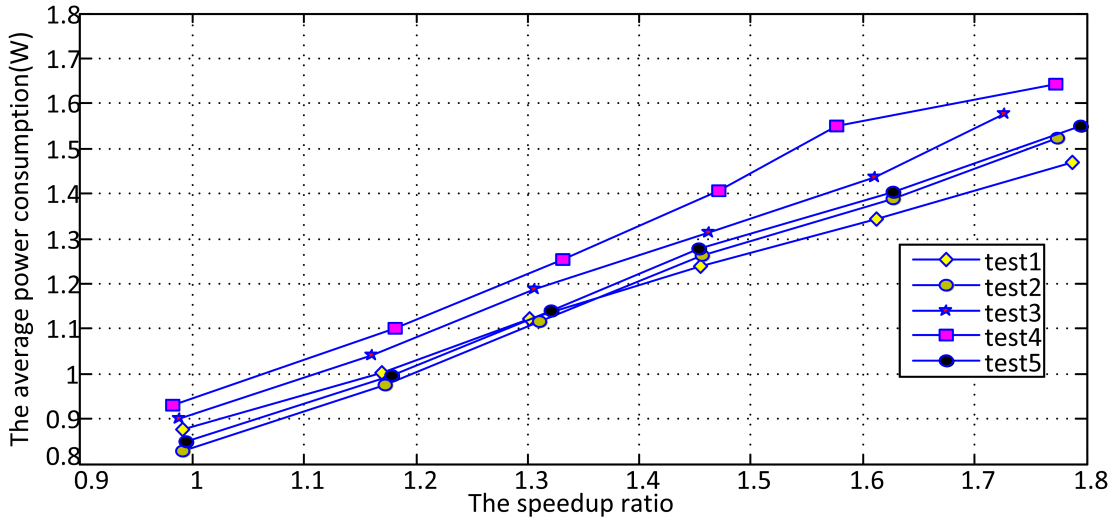
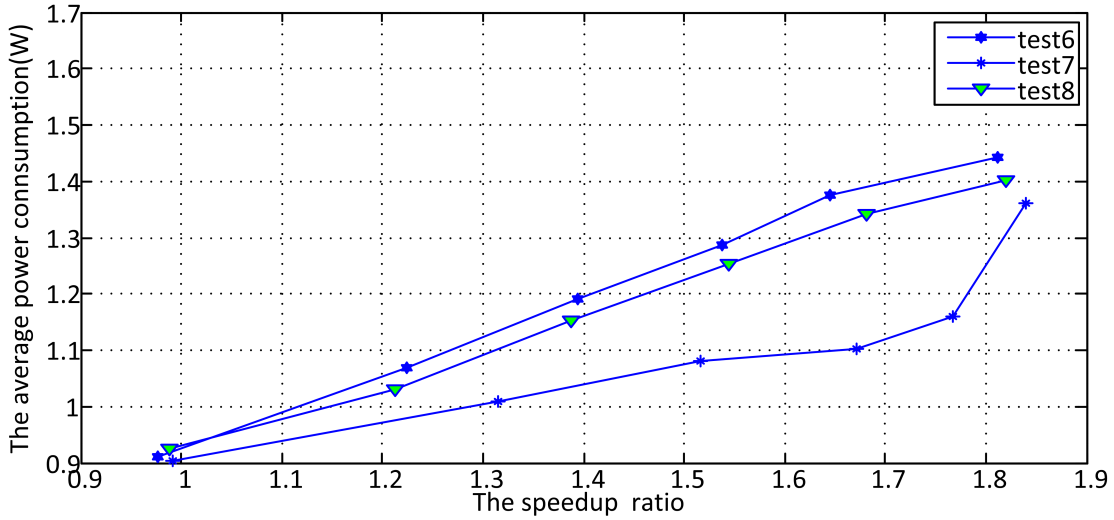


Figure 6.12: The instruction per clock cycle for test 1 to test 8 in the worst case.

Figure 6.12 shows the IPC of the eight tests in the worst case, where nearly all of the instructions have to be fetched from the L2 cache. It is clear that the IPC is still proportional to the speedup ratio of each test. Moreover, compared with Figure 6.9 and Figure 6.12, it is clear that the decrease of IPC in test 1 to test 5 is larger than in test 5-test 7. This effect is the same as in Figure 5.3 in Section 5.3.1, where the reason is



(a) The average power consumption of test 1 to test 5.



(b) The average power consumption of test 6 to test 8.

Figure 6.13: The power consumption of the eight tests in the worst case.

explained. Based on these reasons, test 1 to test 5 consume less power in the worst case but test6 to test 7 consume more.

Figure 6.13 shows the power consumption of the eight tests in the worst case. The power is still proportional to the speedup ratio of each test. In Figure 6.13(a), test 4 consumes the most power but the difference is not much compared with others. On the other hand, in Figure 6.13(b) test 7 consumes the least power. For test 1 to test 5, the peak power of each test in the worse case is a little smaller than that in the best case. For example, compared with Figure 6.13(a) and Figure 6.10(a), the peak power in the worst case is about $1.5W$ but it is about $1.6W$ in the best case. In contrast, for test 6 to test 8, the peak power consumption in the worst case is a little bigger than that in the best case. As for the single core, the pipeline usage will be less efficient and the power

may reduce. However, transferring data from the L2 cache to L1 cache consumes a lot of power. Thus, although the speed of the tests in the worst case is slower, the overall power consumption may be more than in the best case.

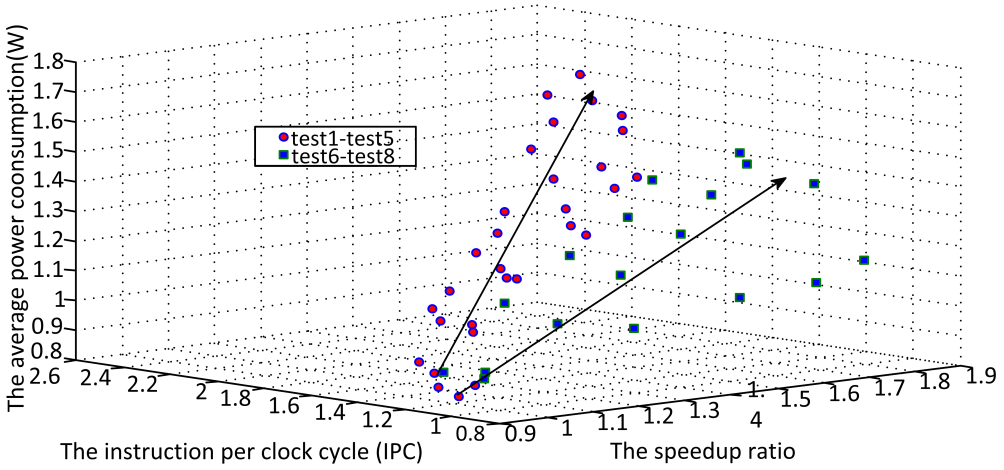


Figure 6.14: The speedup ratio, ICP and average power consumption of test 1 to test 8 in the worst case.

Figure 6.14 shows the speedup ratio, ICP and the average power consumption of all of the tests in the worst case. It shows that both the speedup ratio and the IPC can affect the power just like the best case test. Compared with the best case in Figure 6.11, the data points of test 1 to test 5 are closer to the points of test 6 to test 8. The reason is the difference in the IPC between test 1 to test 5 and test 6 to test 8 in the worst case is smaller than that in best case.

6.5.3 Summary of the Experimental Results

The power consumption in both the best case and the worst case is proportional to the speedup ratio. The IPC and components of the program also affect the power. When the speedup ratio is similar, the power consumption of each test in the best case and the worst case is similar. However, the energy consumption in the worst case is much more than in the best case. The reason is the IPC decreases from the best case to the worst case, thus the runtime of a program is much longer.

Based on the comparison between the IPC of each test in the best case (Figure 6.9) and the worst case (Figure 6.12), it is clear that the instruction pre-fetch unit and branch predictor works well. For example, for the worst case, although a lot of instructions are fetched from the L2 cache, the IPC is still about 1.3 (speedup ratio is one) and the lost speed is only 0.6, which is less than half of the full speed.

6.6 Instruction Level Power Modeling

In order to generate a concise model, we divide the instructions into three classes: ALU logic instructions (including arithmetic and logic instructions), load and store. We assume the power is affected by the speedup ratio, the IPC and the components of a program. Therefore, it can be represented by the following equation:

$$Power_{average} = k_0 + k_1 \times p_{instruction\_distribution} + k_2 \times IPC + k_3 \times Speedup \quad (6.3)$$

We have already analysed 96 different tests. Based on these results, we use linear regression to derive the model which is presented as follows:

$$\begin{aligned} Power_{average} =& 0.0606 + 0.1247 * p_{ALU} + 0.0633 * p_{Load} + \\ &+ 0.0829 \times IPC + 0.6453 \times Speedup, \end{aligned} \quad (6.4)$$

where $Power_{average}$, p_{ALU}, p_{Load}, IPC, and $Speedup$ are the average power consumption, the ALU instruction percentage, the load percentage, the IPC, and the speedup ratio, respectively. $k_1 \times p_{instruction\_distribution}$ in Equation 6.3 is replaced by $0.1247 * p_{ALU} + 0.0633 * p_{Load}$. The reason p_{Load} is missing is because we assume that all of the instructions come from these three cases. Thus, $p_{logic} + p_{store} + p_{Load} = 100\%$, and p_{Load} can be presented by p_{store} and p_{logic} after creating the power model by linear regression, such as $p_{Load} = 100\% - p_{logic} - p_{store}$. For the same reason discussed in Section 4.6 and Section 5.8.2, the p_{load} is represented by p_{logic} and p_{store} after linear regression.

However, the energy is also important and the power model can be extended to a energy model by Equation 4.3. Ubuntu Linux is used as the operating system, thus measuring the runtime of the program is very easy. Figure 6.15 shows the measured power and estimated power. It is clear the power consumption in the best case is estimated accurately. However, there are several bad estimations from the worst case. The power consumption of the processor has a large range, from $0.8W$ to $1.6W$.

Figure 6.16 shows the error between the estimated and measured power. The error is less than 10% in most of the cases. However, there are six predictions whose errors are more than 10% and they all come from the worst case. There are 96 (8 tests $\times$ 6 different speedup level $\times$ best case or worst case =96) estimations in total, thus the bad prediction percentage is only 6.25%. Moreover, the errors of 80 estimations are less than 5%, and the average absolute error value is 2.92% and 4.972% for the best case and worst case, respectively. Therefore, this model is accurate.

Although the model is created based on integer tests, this model can also be used for floating point programs. Firstly, the integer and floating point instructions share a lot

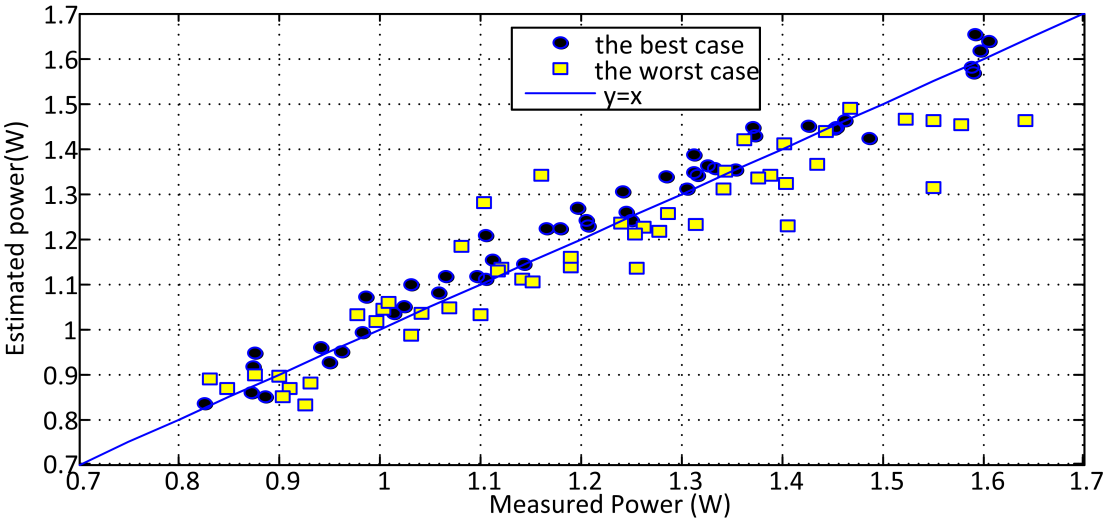


Figure 6.15: The comparison between measured power and estimated power.

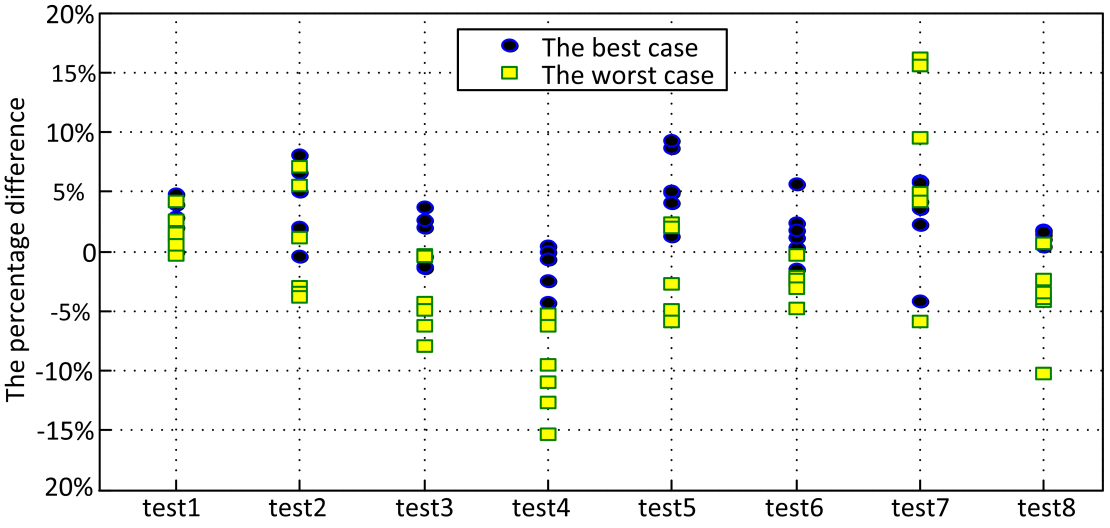


Figure 6.16: The error between predicted and measured power.

of the resources such as L2 cache, L1 cache and TLB. Moreover, they also share most of the pipeline stages such as instruction fetch stage, register rename stage and dispatch stages. The only different stage is the execute stage. Therefore, the different power consumption between an integer and a floating point comes from the execute stage. As the whole system includes 1MB L2 cache, L1 cache, 48 kB ROM, this difference is unlikely to be significant.

Secondly, Load/Store, branch and integer instructions make up most of the instruction types and floating point is less significant. For example, the sum of the percentages of branch, load and store is bigger than 50% (without considering the percentage of integer ALU instructions) in most of the benchmarks in the SPEC2006 floating point benchmark suite [116] and a similar result is presented in [117]. Moreover, the percentage of floating point instructions will be less in other integer benchmarks and programs.

Thirdly, the floating point pipeline consumes more power than an integer, because the floating point pipeline is more complicated than the integer. This may reduce the accuracy of our model to predict floating point programs and make the estimation lower than the real measurement. However, the number of execution stages of a floating point instruction is more than for an integer. Thus, a floating point instruction needs more clock cycles to finish. If a floating point instruction makes the pipeline stall, such as with data dependency, the system will stall for more cycles than for an integer. Therefore, the usage of the pipeline is inefficient, which will reduce the overall average power consumption. Considering these two factors together: one increases the power consumption (the floating point pipeline consumes more power than an integer) but one decreases the power consumption (floating point may stall the pipeline longer than integer), the effect of the floating point pipeline will not reduce the accuracy much.

Thus, considering these three factors, we will confirm this model can also work for floating point programs in the next section.

6.7 Validation

In order to test the performance of the model, we use the Stanford Parallel Application for SHared memory (SPLASH2) as the benchmark [118]. It includes twelve test programs in total and nine of them were compiled and run successfully.

Table 6.1: The setup for SPLASH2 benchmark suit.

Name	Input Value	Thread Number
Barnes	default input file	1,2
Cholesky	cache=32768MB tk15.O	1,2
FMM	input.2048	1,2
LU	444 × 444 matrix	1,2
Ocean	no	1,2
Radix	no	1,2
Raytrace	teapot, global memory=64 MB	1,2
Water-Spatial	default input file	1,2
Water-nsquared	default input file	1,2

Table 6.1 shows the setup of each test. Barnes, Water-Spatial and Water-nsquared use the default input file. Ocean and Radix do not need any input values. For Cholesky, the parameter cache size is set to 32768 Bytes and the file "tk15.O" is chosen as the input. FMM uses the file "input.2048" as the input. The LU test in contiguous blocks case uses 444 × 444 as the decomposition matrix. All of benchmarks were tested with both a single thread and two threads.

There are three input values in our model: speedup ratio, IPC and the components of each program. Before we validated our model, we used the gem5 simulator in full system mode [78] to analyse each program and get the distribution of each instruction type and the total number of instructions in each benchmark.

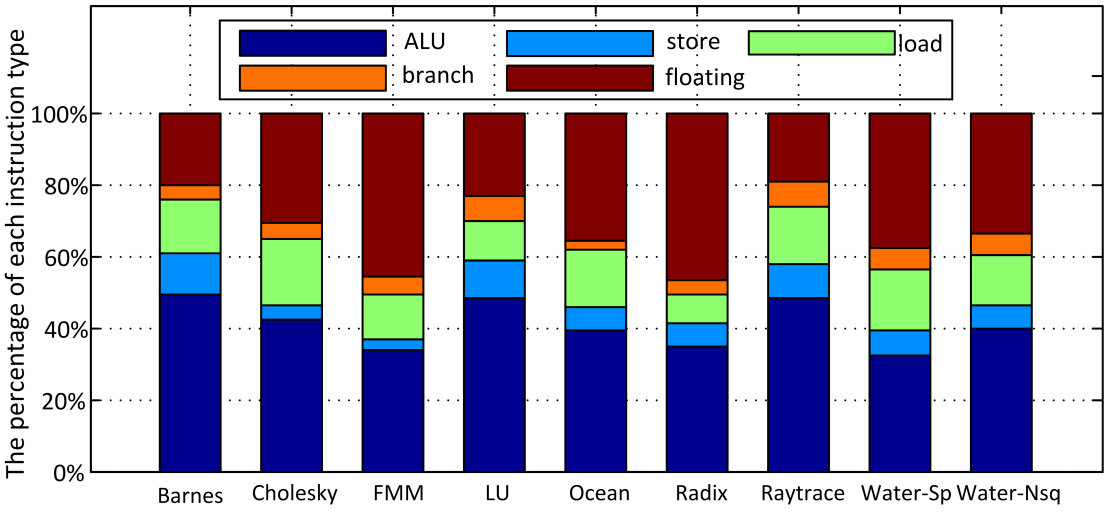


Figure 6.17: The components of each benchmark.

Figure 6.17 shows the components of each test. It is clear that compared with the other three instruction types, floating point and ALU instructions are the most significant parts. Load is the third biggest part and branch is the least significant part. Radix has a bigger floating point distribution than the others, which is 46.39%. However, Raytrace has the least floating point distribution of all of the tests, which is 18.97%. Barnes has the biggest ALU distribution, which is 49.6%, and Water-Nsq has the least ALU distribution, which is 32.56%

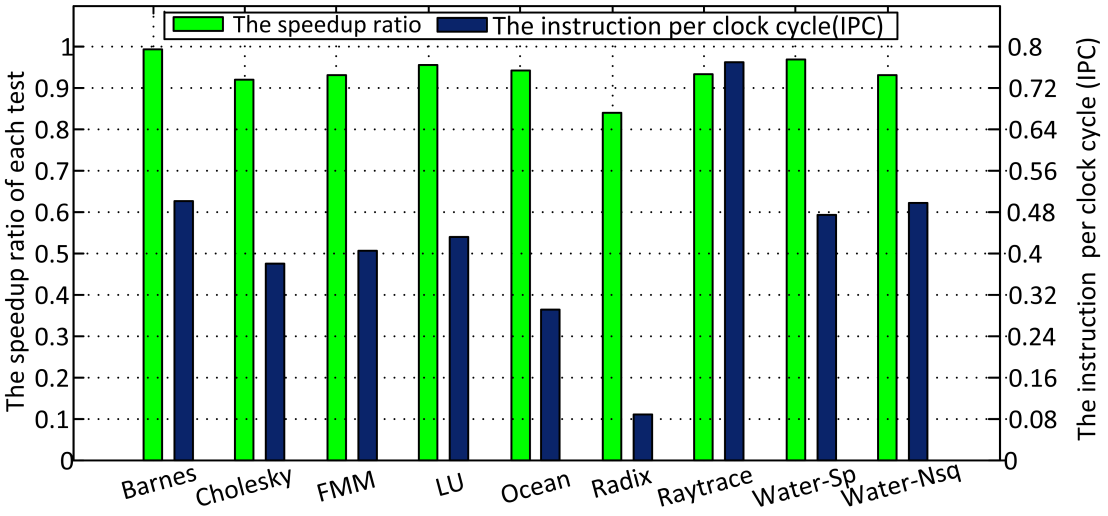


Figure 6.18: The speedup ratio and IPC of each benchmark with a single thread.

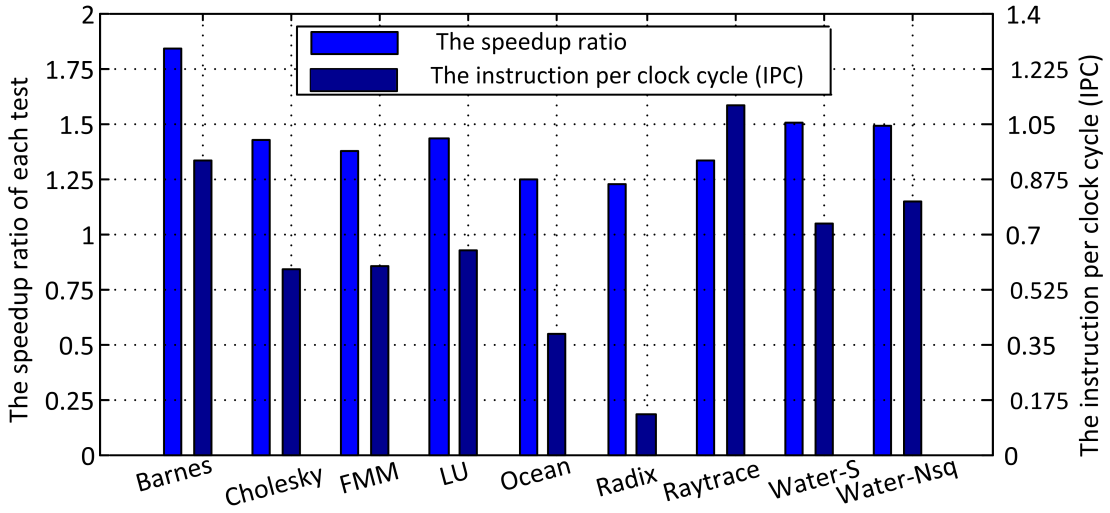


Figure 6.19: The speedup ratio and IPC of each benchmark with two threads.

Figures 6.18 and 6.19 show the speedup ratio and IPC of each test in single thread mode and two thread mode. The timing information of the two thread test comes from the measurement in the OS such as speedup ratio and runtime of a program. IPC is calculated by $\frac{Number\ of\ Instructions}{Clock\ Frequency \times Runtime}$. The speedup ratio of the single thread tests also comes from the measurement but the IPC is calculated by $\frac{IPC_{two}}{speedup_{two}/speedup_{single}}$. We assume that the instruction numbers of the program running in a single thread is the same as in two threads, therefore the IPC is proportional to the speedup ratio.

For tests with a single thread, from Figure 6.18, it is clear that all of the speedup ratios are close to one. The lowest speedup ratio comes from Radix which is 0.842 and the largest is Barnes, which is 0.994. Ideally, the speedup ratio in a single thread should be one. However, because of OS effects, such as task scheduling, the speedup will be a little less than one.

For tests with two threads, from Figure 6.19, Barnes gets the biggest speedup ratio, which is more than 1.75, which means it gets the most benefit from parallel threads. Radix has the lowest speedup ratio, which is less than 1.25. Overall, all of the tests have similar speedup ratios, which is from 1.25 to 1.5. However, the IPC of each test varies considerably. Raytrace has the biggest IPC which is more than 1 but Radix's IPC is less than 0.175. The reason is that Raytrace has the smallest floating point percentage but Radix has the largest. Floating point takes more clock cycles to finish and may stall the pipeline longer than for an integer, as in the true data dependency case.

Figure 6.20 and 6.21 show the test results of the model running the benchmarks with a single thread and two threads respectively. Each figure includes the estimated and measured power, and the estimation error.

For tests with a single thread, from Figure 6.20, it is clear that most of the tests consume similar power, which is about $0.8W$. However, Radix consumes the least and the reason

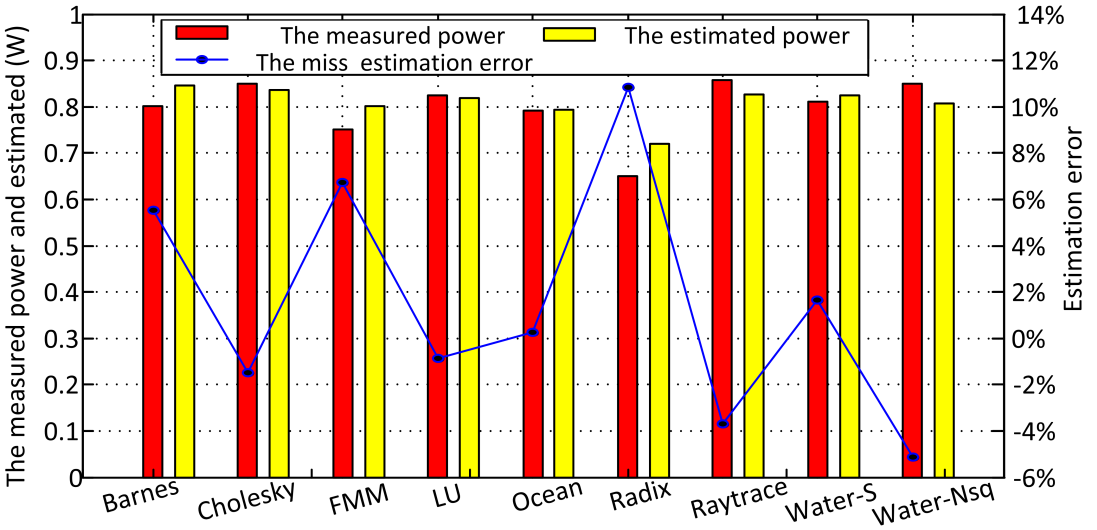


Figure 6.20: The error between predicted power and measured with a single thread.

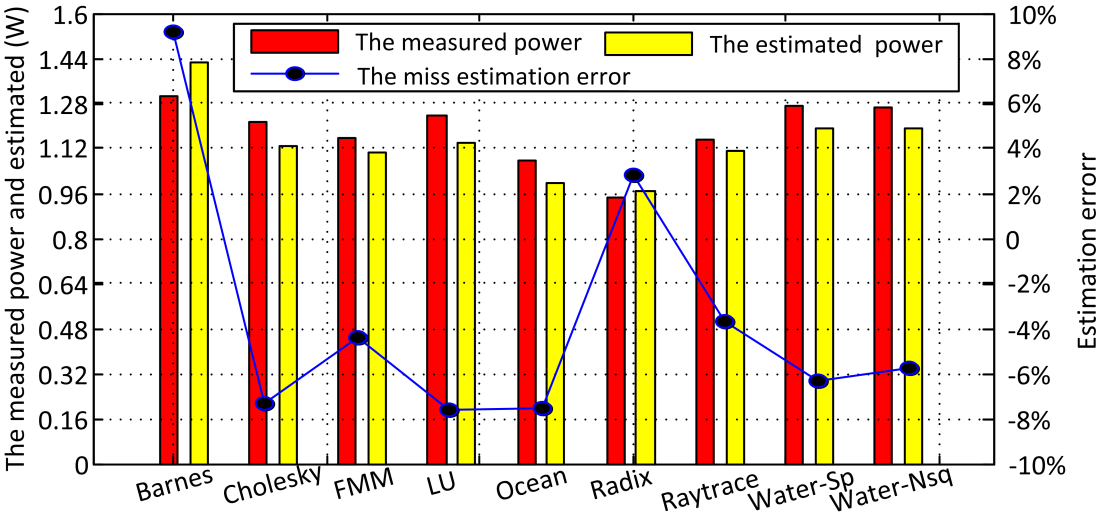


Figure 6.21: The error between predicted power and measured with two threads.

is all of the tests have a similar speedup ratio which is close to one and it has the lowest IPC. The least predicted error is 0.26%, which comes from Ocean and the largest error comes from Radix and its estimation error is 10.83%. Except Radix, all of the absolute estimation errors are less than 8% and the average absolute error value of all tests is 4.6%.

For tests with two threads, from Figure 6.21, Barnes consumes the most power because it has the largest speedup ratio which is the most significant factor in our model. In contrast, Radix consumes the least, as its speedup ratio is the smallest. Overall, the power consumption of all tests is from $0.96W$ to $1.28W$. Although Radix and Ocean have similar speedup ratios, Ocean has a larger IPC and consumes more power than

Radix. The largest error is 9.18% which comes from Branes. Radix has the smallest error which is 1.79%. The average absolute error value of all tests is 5.95%.

Based on the SPLASH2 benchmark results, it is clear that the model works well in both single thread mode and multi-thread mode.

6.8 Discussion: Energy Consumption and Performance

In this subsection, we will discuss the energy consumed by the program in a dual-core processor. Because of the architecture difference between the dual-core and single core, the discussion for the single core (Section 5.11) needs to be proved to apply to a dual core.

6.8.1 Discussion: EPI vs IPC for a Dual-core Processor

For a dual core system, the CPU usage depends on the applications, since a program can run with a single thread or two threads. Thus, we have to analyse the EPI and IPC of an application with both a single thread and two threads.

Based on Equation 4.20 and the power model Equation 7.1, the EPI for a dual core processor can be presented as

$$\begin{aligned} EPI &= \frac{P}{IPC \times F} \\ &= \frac{0.0606 + 0.6453 \times Speedup + 0.0829 \times IPC_s + 0.1247 \times P_{ALU} + 0.0633 \times P_{LD}}{IPC \times F} \\ &= \frac{0.0606 + 0.6453 \times Speedup + 0.1247 \times P_{ALU} + 0.0633 \times P_{LD}}{IPC \times F} + \frac{0.0829}{F} \\ &= \frac{C}{IPC \times F} + \frac{0.0829}{F}, \end{aligned} \tag{6.5}$$

where C is $\frac{0.0606+0.6453\times Speedup+0.1247\times P_{ALU}+0.0633\times P_{LD}}{IPC\times F}$. Thus, C is a constant for a program and the EPI is inversely proportional to the IPC.

Table 6.2: The benchmarks ranked by EPI and IPC with a single thread.

	Barnes	Cholesky	FMM	LU	Ocean	Radix	Raytrace	Water-S	Water-NS
EPI	8	3	5	4	2	1	9	6	7
IPC	2	7	5	6	9	9	1	4	3

Figure 6.22 shows the energy per instruction and IPC of each test with a single thread and Figure 6.2 ranks the workloads by EPI and IPC. They demonstrate that EPI is inversely proportional to IPC for a single thread program running in a dual-core processor. For example, Radix has the biggest EPI (6.17) but the lowest IPC (0.087). In contrast, Raytrace has the lowest EPI (0.921), but the highest IPC (0.776).

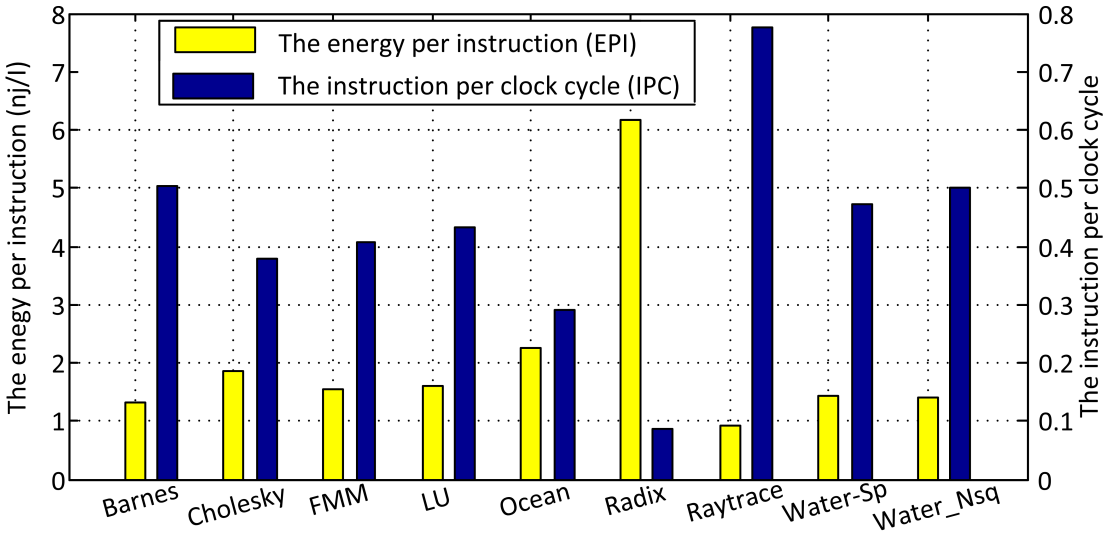


Figure 6.22: The energy per operation VS operation per clock cycle with a single thread.

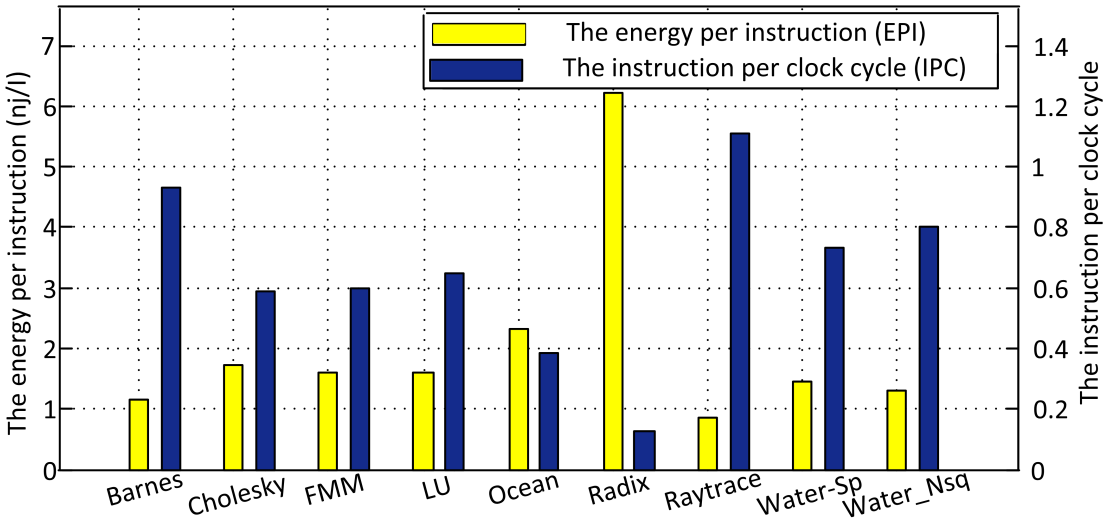


Figure 6.23: The energy per operation VS operation per clock cycle with two threads.

Table 6.3: The benchmarks ranked by EPI and IPC with two threads.

	Barnes	Cholesky	FMM	LU	Ocean	Radix	Raytrace	Water-S	Water-NS
EPI	8	3	5	4	2	1	9	6	7
IPC	2	7	5	6	9	9	1	4	3

Figure 6.23 shows the energy per instruction and IPC of each test with two threads and Table 6.3 ranks the workloads by EPI and IPC. Radix has the biggest EPI (6.22) but the lowest IPC (0.127). In contrast, Raytrace has the lowest EPI (0.868), but the highest IPC (1.107).

Increasing the IPC will lead to a lower EPI and make energy usage more efficient.

Whether the processor is a single core or multi-core, or the program has a single thread or two threads does not matter. Therefore, the ideas presented for the ARM11 in Section 4.8.3 about how to write energy efficient code can be extended to multi-core. For example, if the pipeline stalls can be reduced or the speed of the program is improved, such as by reducing the cache miss rate, the energy usage will be better.

6.8.2 Discussion: the Energy of a Single Thread Program vs a Multi Thread Program

A program can run in a single thread or two threads, but which consumes more energy is not clear. In order to discuss this problem, we assume the energy consumed by a program which runs with a single thread and multi-threads are E_s and E_m respectively.

Based on Equation 7.1, we assume that the speedup ratio in single thread mode is one and the energy ratio between E_s and E_m is

$$\begin{aligned} Ratio =& \frac{E_s}{E_m} = \frac{P_s \times T_s}{P_m \times T_m} = \frac{P_s}{P_m} \times \frac{T_s}{T_m} = \frac{P_s}{P_m} \times Speedup \\ =& \frac{0.0606 + 0.6453 \times Speedup_s + 0.0829 \times IPC_s + 0.1247 \times P_{ALU} + 0.0633 \times P_{LD}}{0.0606 + 0.6453 \times Speedup_m + 0.0829 \times IPC_m + 0.1247 \times P_{ALU} + 0.0633 \times P_{LD}} \\ & \times Speedup \\ =& \frac{C_1 + 0.6453 \times 1 + 0.0829 \times IPC_s}{C1 + 0.6453 \times Speedup + 0.0829 \times IPC_s \times Speedup} \times Speedup \\ =& \frac{C_1 + 0.6453 + 0.0829 \times IPC_s}{\frac{C_1}{Speedup} + 0.6453 + 0.0829 \times IPC_s} \\ =& \frac{C_1 + C_2}{\frac{C_1}{Speedup} + C_2} > 1, \end{aligned} \tag{6.6}$$

where C_1 is a constant $0.0606 + 0.1247 \times p_{ALU} + 0.0633 \times p_{LD}$, and is related to the components of a program. C_2 is $0.6453 + 0.0829 \times IPC_s$ and is related to the IPC of a program with a single thread.

If the number of instructions of a program with a single thread and two threads are the same (in other words, the amount of the work is the same for both cases: a single thread and two threads), based on the analysis of Equation 6.6, the EPI of a program running in two threads will be less than that in a single thread. Furthermore, ideally, the more parts of a program that can run in parallel, the higher the speedup ratio will be and the less energy it will consume.

Figure 6.24 shows the experimental results for the both EPI of each benchmark in single thread mode and multi-thread mode. Barnes, Cholesky, Raytrace and Water-Nsq consume less EPI in multi-thread than single thread and this is the expected result. However, FMM, LU, Ocean, Radix and Water-Sp consume more EPI in multi-thread mode. But the EPI ratio between single and multi-thread is quite close to one. For

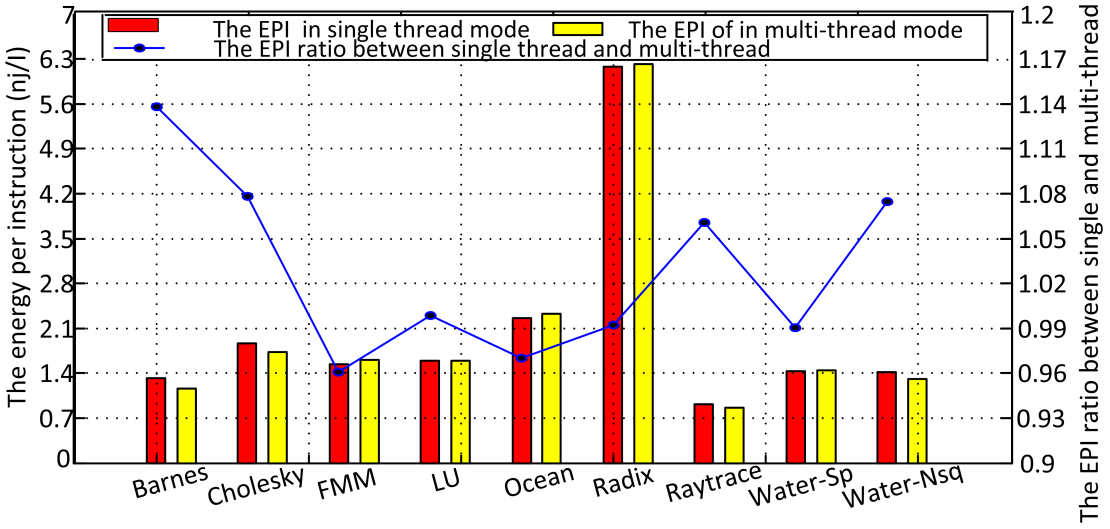


Figure 6.24: The error between predicted power and measured in two thread mode.

example, the ratios of LU, Radix and Water-S are 0.9988, 0.9922 and 0.9901 respectively. The reason is that in Equation 6.6 we assume that the speedup of single thread is one but in fact, it is slightly less than one, thus the IPC of a single thread is also overestimated.

From Figure 6.24, it is clear that the energy consumption of a program with multi-threads is very close to that with a single thread and may even consume less energy. Therefore, multi-threading can reduce the run time of a program without sacrificing energy, and may even save energy. The reason for this can be analysed in another way. A lot of resources of the processor are shared between the two cores such as the L2 cache. Therefore, even if only one core works, these resources will still consume energy. Assume the power consumed by each core is $power_{core}$ and by the other shared resources is $power_{share}$. The energy of a program with a single thread will be $(power_{core} + power_{share}) \times T$ (T is the runtime of the program with a single thread). If the same program is run with two threads and the speedup ratio is $n (n > 1)$, the energy will be $(power_{core} \times n + power_{share}) \times \frac{T}{n}$. Therefore, it will consume less energy.

6.9 Conclusion

In this chapter, we extend our method to a dual-core ARM Cortex-A9. Firstly, we classified the instructions into three classes: load, store, and ALU. We assumed that the power is affected by three factors: IPC, speedup ratio and the components of a program. 96 tests were created to analyze how these three factors affect the power together. Based on the test results, we used linear regression to create a power model. Nine benchmarks from SPLASH2 were used to test the performance of the power model and it shows good performance in both single thread and two thread tests. For example, for each

test running in one thread, the best prediction error is 0.26% (Ocean) and the largest error is 10.83% (Radix). The average absolute error of all tests is 4.6%. For each test running in two threads, the best predication error is 1.79% (Radix) and the largest error is 9.18% (Barnes). The average absolute error of all tests is 5.78%.

Moreover, we proved that the EPI of a program with a single thread is inversely proportional to IPC with different types of the processor, such as simple scalar, super scalar, and multi-core processors. The energy consumed by a program with a single thread is the same as or less than that with multi-threads. Therefore, multi-threading can reduce the runtime of a program without sacrificing energy.

Chapter 7

How To Apply The Model To New Processors

So far, we have proved that our method works in several different RISC processors including ARM11, ARM Cortex-A9 and dual core ARM Cortex-A9. However, there are a lot of different RISC architectures, such as MIPS and SPARC. In this chapter, we will discuss the limitations of our method and explain how to apply our model to other processors.

7.1 The limitation of the method

There are two limitation of the method. Firstly, the method is designed for RISC processors, therefore it may not apply to a CISC processor. The reason is we use instruction per clock cycle (IPC) to model the speed of the processor. This means that a higher IPC implies the processor is busy and consumes higher power. However, for a complex instruction set computer (CISC), IPC cannot reflect the speed of the processor so well. The reason is that for a RISC processor, one instruction normally means one function or one job. However for a CISC processor, one instruction may do a lot of things. Thus, the IPC of a CISC processor is lower compared with a RISC processor, but it does not mean the performance of the CISC processor is lower or consumes less power.

Secondly, before creating our model, we have proved that for our target processors, the overhead power cost of two consecutive arithmetic and logic instructions and the data may not affect the power much. For example, the maximum overhead power of the ARM11 and ARM Cortex-A8 is $0.0101W$ (4.99%) and 0.0058 (0.32%), respectively. Thus, we do not need to design more tests to consider these effects and can ignore them in our model. However, for other processors, especially simple architecture processors,

these factors may affect the power and energy significantly [8]. For a simple processor, a lot of hardware which is used to increase the performance in RISC processors is not used. For example, processors may have smaller cache sizes, shallow pipelines, and no branch predictor. Thus, the data switching rate of the datapath may become more important and affect the power more. Finally, this model may not apply to those processors where the effect of data and the overhead of arithmetic and logic instructions cannot be ignored.

7.2 How to apply the model to new processors

In this section, we will explain how to extend our power model to another ISRC processor. In order to explain more clearly, we will use MIPS as an example only, but we have not tested this in a real MIPS processor.

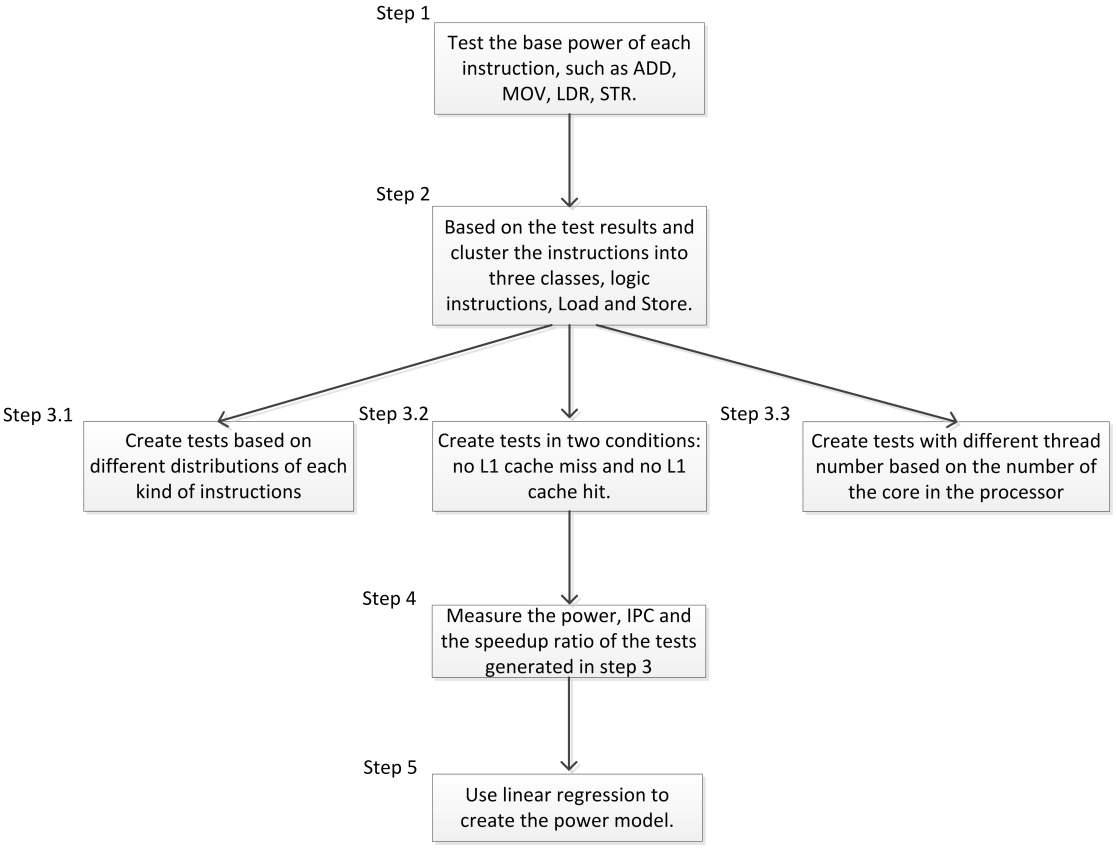


Figure 7.1: How to apply the model to new processors.

Figure 7.1 shows how to apply the model to a new processor. There are five steps to create the power model.

Step 1 Ideally, we need to test the base power of all the arithmetic, load and store instructions, such as ADD, AND, LB, SB in MIPS. However, the instructions from the same category should consume similar power. Thus, if a MIPS processor follows this rule, there is no need to test all of the instructions. On the other hand, it is not harmful

to test all of the instructions. Setting up the test is the same as in the ARM example, Section 4.3.

On the other hand, we need to check that whether the effect of overheads of the instructions from different instruction categories and data can be ignored. The method is to run a loop that only contains the target test instructions. The size of the loop has to be less than the size of the L1 cache. An example of the *AND* test pseudo code to test the effect of the operand is shown below and more details of the test harness are presented in Appendix A.2.1.

```
while(true);
{
asm(" AND r3, r2, r1 ");
asm(" AND r4, r1, r2 ");
asm(" AND r3, r2, r1 ");
asm(" AND r4, r1, r2 ");
 ..............
asm(" AND r3, r2, r1 ");
//  the size of the loop has to be less than the size of L1 cache
}
```

We expect that the difference between the minimum operand switching test, such as 0 bits, and the maximum operand switching, such as 16 bits, should be less 5%. However, we do not know the specific value. Furthermore, the bigger the difference is, the worse the performance of the model. If this number is more than 5%, the case where the average number of switching bits is no more than 10 could be a good choice, because we assume that a low switching rate in a real program is more likely than a high switching rate. Similarly, the overhead of the instructions from the same category, such as all ALU instructions, should be less than 5%. Thus, there is not need to test every pair because different ALU instructions use very similar hardware. Thus, the overhead power or energy should be very little, again around 5%. The set up of the overhead and operand switching tests are the same as for the ARM example, Section 4.5 and Section 4.4, respectively.

Step 2 We then need to cluster the instructions into classes based on the power test results from step 1. We propose that three classes: logic, load and store are a good and common classification, based on their different pipeline usage. The reason is most logic and arithmetic instructions use the the same pipeline, and thus consume similar power. Compared with the logic and arithmetic instructions, load and store use different pipelines and consume different power. Thus, for ARM11, ARM Cortex-A8 and ARM-A9, we cluster the instructions into three classes. Moreover, a lot of RISC processors, such as MIPS, use load and store to access the memory and the other instructions focus on calculations. This suggests our clustering method. On the other hand, if

this clustering method does not work well, Bona *et al.* presented a general method for dividing the instructions into different classes, Section 2.3, [33, 37].

The following table is an example of instruction clusters of MIPS; we focus on integer instructions only.

Table 7.1: An example of clustering MIPS instruction.

Logic	Load	Store
ADD	LB	SB
ADDIU	LBU	SH
ANDI	LH	SW
ORI	LHU	SWL
XORI	LWL	SWR
LUI	LWR	
ADD		
ADDU		
SUB		
SUBU		
AND		
OR		
XOR		
SLL		
SRL		

We only used one class for OpenRISC, not three. The reason was that the OpenRISC was an initial test, used to identify which factors could affect the power usage significantly, and how the pipeline usage was related to power consumption. Moreover, various advanced technologies for front-end RTL design, such as branch prediction, instruction pre-fetch, and back-end P&R (place and route) design were not used. Low power techniques, such as clock gating and power gating, were not included. Thus, these tests could not fully present the real behaviour of a real processor, but they were good enough for finding several useful results, such as that different logic instructions consume similar power. On the other hand, we also found that predicting the timing of a program is hard if only based on the instructions, due to the Out-Of-Order pipeline and branch prediction technology. Thus, we use average power $\times$ timing to predict the energy of a program.

Step 3 In our method, there are three main factors that can affect the power: 1. different distributions of each kind of instruction, 2. different conditions (L1 cache miss and hit), and 3. different speedup ratios (this is used to model multi-core processors). The third step is to create the tests by changing the weight of each factor and to gather the input data for linear regression. The main body of the each test is still a loop and the components of the loop are changed based on the different weights of the three factors:

Step 3.1 In order to analyse how different classes of instructions affect the power, we need to change the percentage of each class in a program. Then, we can test the power consumption and estimate how the different kinds of instruction affect the power.

Firstly, we need to choose a number as the granularity or the step distance between different ratios of each instruction class. Then, the test should change the percentage of each kind of instruction by the step distance to cover all of the different cases. There is no restriction on the step distance, but the smaller it is, the greater the total number of simulations without necessarily increasing the accuracy of the model. We suggest that 20% or 25% would be a good number. For example, for the ARM Cortex-A8 processor, we chose 25% as the step distance, and created eight different settings by decreasing the percentage of arithmetic and logic instructions and increasing the percentage of load and store. This is shown in Figure 5.13 in Section 5.8.1.

Step 3.2 In order to analyse how cache misses affect the power, all of the tests have to be divided into two cases: no L1 cache miss and no L1 cache hit. This can be achieved by changing the size of the loop. For example, for no L1 cache misses, the loop can be less than half of the size of the L1 cache. For no L1 cache hit, the size of the loop can be as much as double the size of the L1 cache.

Step 3.3 For multi-core processors, the speedup ratio can affect the core because a higher speedup ratio means more cores run at the same time. In order to change the speedup ratio, the tests can be divided into two parts: a single threaded part and a multi-threaded part. The speedup ratio can be set by changing the percentage of the instructions in the two parts.

Furthermore, the highest speedup ratio is equal to the number of cores in the system. Assuming there are N cores in the system and the granularity is M (M is used to define the difference of the speedup ratio between each test), there will be $\frac{N-1}{M} + 1$ different settings. For example, we presented the power model for a dual core ARM Cortex-A9 system in Section 6.4 where the granularity is 0.2. Thus, there are six different speedup ratios, which are 1, 1.2, 1.4, 1.6, 1.8 and 2. An example of the pseudo code is shown below:

```
#include <stdio.h>
#include <stdlib.h>
#include <pthread.h>

#define THREADS 2
#define N 20000  // the loop number of the second part (multi-thread function )
#define LOOP_TIME 1000  // the total loop number of the test.

void *multi(void *val)  // the multi-thread part, which is a loop
{
    int thread_id = (int) val;
```

```
    int i;
    for (i = thread_id; i<N; i += THREADS)
    {
        asm (" sub r4 , r1,  #0xf  ");
        asm (" orr r1 , r1,  r2 ");
        asm (" sub r4 , r1,  #0xf  ");
        asm (" eor r6 , r2,  #0xf3  ");
        ............
        asm (" ADD r3 , r1,  #0xf");
    }
}

void main()
{
    pthread_t threads[THREADS];
    int i;
    int k;
    int M=LOOP_TIME;

     while(M>0)
    {
    for (k= 0; k<N/8;k++ )
   // The single thread part, which is a loop
    // The loop size depends on the test( the best case or the worst case)
    // The instructions in this part can be set by changing the loop condition
    (N/8)
    // The bigger the condition is , the more instructions are run in single
    thread.

        {
        asm (" sub r4 , r1,  #0xf  ");
        asm (" orr r1 , r1,  r2 ");
        asm (" sub r4 , r1,  #0xf  ");
        asm (" eor r6 , r2,  #0xf3  ");
        ............
        asm (" ADD r3 , r1,  #0xf");
        }
    // k is related with N.

      // call the multi-thread function
    for (i = 0; i < THREADS; i++)
         {
            pthread_create(&threads[i], NULL, multi, (void*) i);
    /* Create independent threads each of which will execute function */
         }
        for (i = 0; i < THREADS; i++)
        {
```

```
            pthread_join( threads[i], NULL);
        }
   /* Wait till threads are complete before continues. */
          M--;
    }
}
```

Considering the different combinations of the three factors in steps 3.1, 3.2, and 3.3, there are 8 (step 3.1) $\times$ 2 (step 3.2) $\times$ ($\frac{N-1}{M}+1$) (step 3.3) tests in total. For example, there are $8 \times 2 \times 6 = 96$ tests in a dual core ARM Cortex-A9 system in Chapter 6.

For a single core processor, step 3.3 can be ignored because the speedup ration is always one.

Step 4 The fourth step is to measure the power, IPC and the speedup ratio of each test created in step 3. The current can be measured by an oscilloscope and hence the power can be calculated. The IPC can be measured as $\frac{number\ of\ instructions}{number\ of\ clock\ cycle}$. The number of instructions is equal to the product of the loop size and the loop time. The speedup ratio can be measured by the operating system.

Step 5 The last step is to create the power model based on the test results in step 4. The input factors are the distributions of each kind of instruction, the IPC, the speedup ratio, and the measured power. The generated model will be

$$\begin{aligned} Power_{average} =& \alpha * p_{ALU} + \beta * p_{load\_or\_store} \\ &+ \gamma \times Speedup + \delta \times IPC + \varepsilon, \end{aligned} \tag{7.1}$$

where α, β, γ, δ, and ε are the coefficients of each type of instruction (we assume that $p_{ALU} + p_{load} + p_{store}$=1, thus only two of the three coefficients are needed), the speedup ratio, the IPC and a constant, respectively. A single core processor, such as ARM11 and ARM Cortex-A8, only has one core, and the speed is always the same as its original speed. Thus the *Speedup* is always one. Then, γ can be merged into the constant ε. Thus, this model covers the ARM11, ARM Cortex-A8 and ARM dual Cortex -A9.

7.3 Conclusion

In this chapter, we discussed how to apply this method to a new RISC processor. Firstly, we need to test the base power of each instruction, then cluster them into three classes: Load, Store, and Logic. Then, we need to create training tests based on three factors: 1, the distribution of different instruction types; 2, the L1 cache always hits and the L1 cache never hits; 3, the *Speedup* ratio if it is a multi-core processor. Finally, based on the power measurement of the training tests, and different factors, an instruction level

power model can be created by using Linear regression. The instruction distribution and instruction number can be found from an ISA simulator, such as gem5, and hardware components, such as performance counters. Thus, the input values can be generated easily and this method is easy to use.

Chapter 8

Conclusion and Future Work

8.1 Conclusion

Nowadays, the power or energy consumed by a chip has become a primary design constraint for embedded systems and is largely affected by software. The aims vary with the application domain, the best program is sometimes the most power or energy efficient one. However, there is a gap between software and hardware that makes it hard to predict which code consumes the least power/energy before measuring it. Therefore, it is vital to discover which factors can affect a program's energy consumption and to create a concise model to estimate it.

In this thesis, we have created instruction level power models for different processors. We demonstrate that several factors which are considered by previous work may no longer be important. For example, the base power/energy cost of different arithmetic and logic instructions are similar to each other, since they use similar blocks. The Hamming distance between the operands of two consecutive instructions does not affect the power much. Thus, our hypothesis is that the power is related to the IPC and the components of a program. We design tests to analyze how these two factors affect the power. Furthermore, the IPC is affected by the cache miss rate and pipeline stalls. We cluster instructions into three groups: ALU, load and store and change the percentage of each group in different tests. Finally, we use a linear regression method to create a power model for different processors. It is not necessary to track and find out what the specific instructions are but just to know which group they belong to. Thus, we do not need a cycle accurate simulator to get this data and so save a lot of effort. Moreover, the model does not consider the overhead power/ energy cost independently. Thus, this model is concise and easy to use (objective 3). Moreover, we extend this method to a dual core processor and the power is affected by the IPC, speedup ratio and the components of a program.

This method has been tested by different types of processors including: a scalar processor ARM11 (Chapter 4), a super scalar ARM Cortex-A8 (Chapter 5), and dual-core processor (each core is ARM Cortex-A9 processor Chapter 6) and mosty of the estimation errors are less than 10% (objective 1).

On the other hand, the tests for creating the model are easily designed and few (objective 2). For example, nine tests and sixteen tests are used to create the power model for the ARM11 and ARM Cortex-A8 respectively.

On top of this, we have extended the power model to the energy model (objective 4). The previous work concentrated on estimating the energy of a program. However, there are a lot of factors that can affect the energy consumed by a program, such as cache misses, and pipeline stalls. Thus, it is hard and inefficient to consider each factor separately. The energy model becomes more and more complex without improving the accuracy much. Instead of establishing the energy model directly, we split this complex work into two simple steps: 1) create the power model, and 2) measure the runtime. The energy is simply estimated by multiplying the average power by the runtime. We prove this method with a simple processor OpenRISC (Chapter 3), a single scalar processor ARM11 (Chapter 4), a superscalar processor ARM Cortex-A8 (Chapter 5), and a ARM Cortex-A9 dual-core processor (Chapter 6).

Moreover, we prove that the EPI of the program is inversely proportional to the IPC based on three different processors: an ARM11, an ARM Cortex-A8, and a dual-core processor. Thus, it is important to make the pre-fetch unit and branch predictor run more efficiently to reduce the cache misses and pipeline stalls. We also prove that a program with two threads will not consume more energy and possibly even less than one with a single thread. Thus, multi-thread technology can reduce the run time of a program without sacrificing energy in a dual-core processor. The reason is that although multi-threading may increase the amount of work, such as creating and deleting threads by the OS, a lot of the resources of the processor are shared between the two cores, such as the L2 cache. Even if only one core works, these resources will still consume energy. Thus, multi-threading, which uses two cores, can make the system run more efficiently; when two cores run, the power increases, the runtime will reduce, and the energy may be less than for a single thread (Section 6.8.2).

8.2 Future Work

8.2.1 Apply to more complex systems

The first piece of future work is to extend this method to more sophisticated and complex processors. In our tests, the multi-core processor only has two cores. Thus it is important to test whether this method could be extended to a processor which has more cores, such

as four. On the other hand, the multi-processor and system network becomes more and more important. Other future work is to test whether this method can be extended to a multiprocessor system. If it works, it may present some hints as to how to write low power/energy cost programs.

8.2.2 Reduce the energy consumption of the system

Another piece of future work is to use this method to make the energy usage of a program more efficient. For example, there are some multi-core processors which have a powerful core with a high energy cost and a simple core with less energy cost. If we can use the energy model to predict how to schedule the programs, the overall energy cost will be lower. On the other hand, General-Purpose computing on Graphics Processing Units (GPGPU) can be used to perform computations in applications which are traditionally handled by the CPU. Based on the energy prediction, if the cost for a GPGPU is lower than for a CPU, the work can be handled by the GPGPU and save energy.

8.2.3 Static program analysis

In order to create the energy model, the run time is necessary. Although we create a method to predict the average power of a program on various platforms, the run time of the program comes from measurement (running the program on the platform), such as the OS. Thus, this is not convenient to fill the gap between the high-level program structure and the low-level energy models. The power model will be more useful if the runtime can be analysed without running the application since the energy can be estimated at the compile time.

Static program analysis and static timing analysis can solve this problem well. This is a method to analyse the computer software without actually executing it [119]. This method can predict how many clock cycles are used to finish the program, which can be used to extend the power model to an energy model easily.

Eder *et al.* used static timing analysis and an energy model to study the static energy estimation [120]. The target processor is a multi-threaded architecture, XMOSXS1-L. The created energy model for XMOSXS1-L can be presented as:

$$E_p = P_{base} N_{idle} T_{clk} + \sum_{t=1}^{N_t} \sum_{i \epsilon ISA} ((M_t P_i O + P_{base}) N_{i,t} T_{clk}, \tag{8.1}$$

where P_{base} is the base power in both active and idle periods. N_{idle} and T_{clk} are the number of idle periods and clock period, respectively. t, i, P_i, O are the number of concurrent threads, each instruction in the ISA, the instruction power, and a constant inter-instruction power overhead, respectively. M_t is a concurrency cost because of the

level of concurrency at which the processor is executing. N_i is the number of times the instruction exists at this concurrency level.

All of the input values of the energy model come from the static program analysis, and this method is tested in five benchmarks: factorial(N) (Calculates N!), fibonacci(N) (Nth Fibonacci no.), square(N) (Computes N^2), poweroftwo(N) (Calculates 2^N), and power($base$,exp) (Calculates $base^{exp}$). However, the authors only presented the results for factorial(N), and the maximum error is -15% when N equals one. When N is bigger than one, the estimated energy is close to the measurement.

The advantage of this method is that it only needs static program analysis to give a good prediction. On top of this, the model considers the input value of N since the energy is highly related to N. However, the benchmarks used to validate the performance of the method are few and they are not complicated enough.

Jayaseelan *et al* estimated the worst-case energy consumption (WCEC) of an embedded system [121]. However, the processor is not a real processor and the test results come from a simulator, SimpleScalar.

Assuming $B_1, ..., B_N$ is the set of basic blocks of the program, B_i is related to the predicted result of its preceding branch and its cache scenarios. Thus, the set of possible cache scenarios at B_i is presented as Ω. Considering the two cases: correct/wrong prediction of the preceding branch, and the possible cache scenarios, the total energy of a program is described as

$$Energy = \sum_{i=1}^{N} \sum_{j \to i} \sum_{\omega \epsilon \Omega_i} energy_{j \to i}^{c,\omega} * count_{j \to i}^{c,\omega} + energy_{j \to i}^{m,\omega} * count_{j \to i}^{m,\omega}, \tag{8.2}$$

where $energy_{j \to i}^{c,\omega}$ is the WCEC of Bi which is executed under the following cases: 1. B_i is reached from a previous block B_j, 2. B_j does not any conditional jump or the branch prediction is correct, and 3. B_i is run under a cache scenario $\omega \epsilon \Omega_i$. $count_{j \to i}^{c,\omega}$ shows how many times is the block B_i executed under this scenario. Similarly, $energy_{j \to i}^{m,\omega}$ is the WCEC of Bi under the following cases: 1. B_i is reached from a previous block B_j, 2. at the end of B_j, the branch prediction is not correct, and 3. B_i is run under a cache scenario $\omega \epsilon \Omega_i$. $count_{j \to i}^{m,\omega}$ shows how many times is the block B_i executed under this scenario.

Eleven benchmarks are used to test the performance of the model, and they are isort, fft, fdct, ludcmp, matsum, minver, bsearch, des, matmult, qsort, and qurt. The estimated WCEC value is close to the observed WCEC. The minimum error is 6%, which comes from ludcmp, and the biggest error is 1.33 which is from fft.

The two models discussed above show that static program analysis and static timing analysis technology can be integrated into an energy model to predict the energy. Thus, another future work direction is to gather the inputs of the power model and runtime

of the program via static program analysis. This method can create a better bridge between high-level program structure and low-level energy models.

Appendix A

Design Codes and Benchmarks

A.1 The Test Code of Chapter 3

A.1.1 The code of Section 3.3

The following code is the Makefile for generating .c, .elf, .bin, .asm, and vmem file:

```
CC=or32-elf-gcc
CO=or32-elf-objcopy
CD=or32-elf-objdump
CFLAGS=
LFLAGS=-lm
filename= hanoi-executed


build:
        echo "blah" ${CFLAGS}
        ${CC} ${filename}.c -o ${filename}.elf
        ${CO} -O binary  ${filename}.elf ${filename}.bin
        ${CD} ${CFLAGS} -d ${filename}.elf > ${filename}.asm
        ./bin2vmem ${filename}.bin > ${filename}.vmem
clean:
        rm ${filename}.elf
        rm ${filename}.bin
        rm ${filename}.asm
        rm ${filename}.vmem
```

A.1.2 The code of Section 3.4

In this section, we will show the codes for the base test in Section 3.4.

The following code is the input.c code for Section 3.4, which is used to initial the processor. Thus, the main part of the input.c code is not important and a unconditional jump will be executed.

```
#include <stdio.h>
int main()
{
int a;
for (a=0;a<1000;a++)
{
a=5+a;
}
asm volatile("l.nop 0x3\n\t");
return a;
 }
```

Then, the corresponding input.vmem file is modified.

```
00000890 9c210004 8521fffc 44004800 15000000
    00000894 d7e117fc 9c410000 9c21fff8 9c600000\\
//9c410000 = unconditional jump to a new empty address\\
// new address=current address+1767*4\\
00000898 d7e21ff8 00001767 15000000 8462fff8
    0000089c 9c630001 d7e21ff8 8462fff8 9c630001\\
000008a0 d7e21ff8 8462fff8 bda3270f 13fffff8
```

After jumping to a new address, which is an empty space, the test code is run. Instead of write the test machine code by modifying the input.vmem file manually, we change the memory Verilog file and the following is the changed ram_wb_b3.v file.

```
//‘include "synthesis-defines.v"
‘timescale 1ns/1ps
module ram_wb_b3(
                 wb_adr_i, wb_bte_i, wb_cti_i, wb_cyc_i, wb_dat_i, wb_sel_i,
                 wb_stb_i, wb_we_i,

                 wb_ack_o, wb_err_o, wb_rty_o, wb_dat_o,

                 wb_clk_i, wb_rst_i);

   parameter dw = 32;
   parameter aw = 32;
    parameter memory_file = "mytest_for_long3.vmem";
/*************************the following is the source code of
    OpenRISC*********/
   input [aw-1:0]       wb_adr_i;
   input [1:0]          wb_bte_i;
```

```
input [2:0]          wb_cti_i;
input                wb_cyc_i;
input [dw-1:0]       wb_dat_i;
input [3:0]          wb_sel_i;
input                wb_stb_i;
input                wb_we_i;

output               wb_ack_o;
output               wb_err_o;
output               wb_rty_o;
output [dw-1:0]      wb_dat_o;

input                wb_clk_i;
input                wb_rst_i;

// Memory parameters
parameter mem_size_bytes = 32'h000_5000; // 20KBytes
parameter mem_adr_width = 15; //(log2(mem_size_bytes));

parameter bytes_per_dw = (dw/8);
parameter adr_width_for_num_word_bytes = 2; //(log2(bytes_per_dw))
parameter mem_words = (mem_size_bytes/bytes_per_dw);
 //32'h5000/(32/8)=h'1400

// synthesis attribute ram_style of mem is block
reg [dw-1:0]         mem [ 0 : mem_words-1 ]  /**/ /* verilator public */
 /* synthesis ram_style = no_rw_check */;

// Register to address internal memory array
reg [(mem_adr_width-adr_width_for_num_word_bytes)-1:0] adr;//15-2=13

wire [31:0]                    wr_data;

// Register to indicate if the cycle is a Wishbone B3-registered feedback
// type access
reg                            wb_b3_trans;
wire                           wb_b3_trans_start, wb_b3_trans_stop;

// Register to use for counting the addresses when doing burst accesses
reg [mem_adr_width-adr_width_for_num_word_bytes-1:0]  burst_adr_counter;
reg [2:0]                      wb_cti_i_r;
reg [1:0]                      wb_bte_i_r;
wire                           using_burst_adr;
wire                           burst_access_wrong_wb_adr;

// Wire to indicate addressing error
wire                           addr_err;
```

```
// Logic to detect if there's a burst access going on
assign wb_b3_trans_start = ((wb_cti_i == 3'b001)|(wb_cti_i == 3'b010)) &
                           wb_stb_i & !wb_b3_trans;

assign  wb_b3_trans_stop = ((wb_cti_i == 3'b111) &
                           wb_stb_i & wb_b3_trans & wb_ack_o) | wb_err_o;

always @(posedge wb_clk_i)
  if (wb_rst_i)
    wb_b3_trans <= 0;
  else if (wb_b3_trans_start)
    wb_b3_trans <= 1;
  else if (wb_b3_trans_stop)
    wb_b3_trans <= 0;

// Burst address generation logic
always @(/*AUTOSENSE*/wb_ack_o or wb_b3_trans or wb_b3_trans_start
         or wb_bte_i_r or wb_cti_i_r or wb_adr_i or adr)
  if (wb_b3_trans_start)
    // Kick off burst_adr_counter, this assumes 4-byte words when getting
    // address off incoming Wishbone bus address!
    // So if dw is no longer 4 bytes, change this!
    burst_adr_counter = wb_adr_i[mem_adr_width-1:2];
  else if ((wb_cti_i_r == 3'b010) & wb_ack_o & wb_b3_trans)
    // Incrementing burst
    begin
       if (wb_bte_i_r == 2'b00) // Linear burst
         burst_adr_counter = adr + 1;
       if (wb_bte_i_r == 2'b01) // 4-beat wrap burst
         burst_adr_counter[1:0] = adr[1:0] + 1;
       if (wb_bte_i_r == 2'b10) // 8-beat wrap burst
         burst_adr_counter[2:0] = adr[2:0] + 1;
       if (wb_bte_i_r == 2'b11) // 16-beat wrap burst
         burst_adr_counter[3:0] = adr[3:0] + 1;
    end // if ((wb_cti_i_r == 3'b010) & wb_ack_o_r)

always @(posedge wb_clk_i)
  wb_bte_i_r <= wb_bte_i;

// Register it locally
always @(posedge wb_clk_i)
  wb_cti_i_r <= wb_cti_i;

assign using_burst_adr = wb_b3_trans;
```

```
   assign burst_access_wrong_wb_adr = (using_burst_adr &
                                       (adr != wb_adr_i[mem_adr_width-1:2]));

   // Address registering logic
   always@(posedge wb_clk_i)
     if(wb_rst_i)
       adr <= 0;
     else if (using_burst_adr)
       adr <= burst_adr_counter;
     else if (wb_cyc_i & wb_stb_i)
       adr <= wb_adr_i[mem_adr_width-1:2];

   /* Memory initialisation.
    If not Verilator model, always do load, otherwise only load when called
    from SystemC testbench.
    */
// synthesis translate_off

`ifdef verilator

   task do_readmemh;
      // verilator public
      $readmemh(memory_file, mem);
   endtask // do_readmemh

`else

   initial
     begin
        $readmemh(memory_file, mem);
     end

`endif // !`ifdef verilator

//synthesis translate_on

   assign wb_rty_o = 0;

   // mux for data to ram, RMW on part sel != 4'hf
   assign wr_data[31:24] = wb_sel_i[3] ? wb_dat_i[31:24] : wb_dat_o[31:24];
   assign wr_data[23:16] = wb_sel_i[2] ? wb_dat_i[23:16] : wb_dat_o[23:16];
   assign wr_data[15: 8] = wb_sel_i[1] ? wb_dat_i[15: 8] : wb_dat_o[15: 8];
   assign wr_data[ 7: 0] = wb_sel_i[0] ? wb_dat_i[ 7: 0] : wb_dat_o[ 7: 0];

   wire ram_we;
   assign ram_we = wb_we_i & wb_ack_o;
```

```
assign wb_dat_o = mem[adr];

// Write logic
always @ (posedge wb_clk_i)
  begin
     if (ram_we)
       mem[adr] <= wr_data;
  end

// Ack Logic
reg wb_ack_o_r;

assign wb_ack_o = wb_ack_o_r & wb_stb_i &
                  !(burst_access_wrong_wb_adr | addr_err);

always @ (posedge wb_clk_i)
  if (wb_rst_i)
    wb_ack_o_r <= 1'b0;
  else if (wb_cyc_i) // We have bus
    begin
       if (addr_err & wb_stb_i)
         begin
            wb_ack_o_r <= 1;
         end
       else if (wb_cti_i == 3'b000)
         begin
            // Classic cycle acks
            if (wb_stb_i)
              begin
                 if (!wb_ack_o_r)
                   wb_ack_o_r <= 1;
                 else
                   wb_ack_o_r <= 0;
              end
         end // if (wb_cti_i == 3'b000)
       else if ((wb_cti_i == 3'b001) | (wb_cti_i == 3'b010))
         begin
            // Increment/constant address bursts
            if (wb_stb_i)
              wb_ack_o_r <= 1;
            else
              wb_ack_o_r <= 0;
         end
       else if (wb_cti_i == 3'b111)
         begin
            // End of cycle
            if (!wb_ack_o_r)
```

```
              wb_ack_o_r <= wb_stb_i;
            else
              wb_ack_o_r <= 0;
        end
    end // if (wb_cyc_i)
  else
    wb_ack_o_r <= 0;


//
// Error signal generation
//

// Error when out of bounds of memory - skip top nibble of address in case
// this is mapped somewhere other than 0x0.
assign addr_err  = wb_cyc_i & wb_stb_i & (|wb_adr_i[aw-1-4:mem_adr_width]);

// OR in other errors here...
assign wb_err_o =  wb_ack_o_r & wb_stb_i &
                   (burst_access_wrong_wb_adr | addr_err);

//
// Access functions
//

// Function to access RAM (for use by Verilator).
function [31:0] get_mem32;
   // verilator public
   input [aw-1:0]            addr;
   get_mem32 = mem[addr];
endfunction // get_mem32

// Function to access RAM (for use by Verilator).
function [7:0] get_mem8;
   // verilator public
   input [aw-1:0]            addr;
         reg [31:0]          temp_word;
   begin
      temp_word = mem[{addr[aw-1:2],2'd0}];
      // Big endian mapping.
      get_mem8 = (addr[1:0]==2'b00) ? temp_word[31:24] :
                 (addr[1:0]==2'b01) ? temp_word[23:16] :
                 (addr[1:0]==2'b10) ? temp_word[15:8] : temp_word[7:0];
      end
endfunction // get_mem8

// Function to write RAM (for use by Verilator).
```

```
   function set_mem32;
      // verilator public
      input [aw-1:0]            addr;
      input [dw-1:0]            data;
      mem[addr] = data;
   endfunction // set_mem32

/*****************the following is our own code************/
   reg [7:0] mem_t [ 0 : 100000 ]   ;
  reg [32:0] data;
/*********************************************low_switch*****************************************
//initialize begins
initial
begin
int i;
i=0;
// move 00000011 (3) to r5
         mem_t[32835]= 8'b00000011;
         mem_t[32834]= 8'b00000000;
         mem_t[32833]= 8'b10100000;
         mem_t[32832]= 8'b00011000;
   //move 00001111(15) to r7
         mem_t[32839]= 8'b00001111;
         mem_t[32838]= 8'b00000000;
         mem_t[32837]= 8'b11100000;
         mem_t[32836]= 8'b00011000;
   // move 00001100 (12)to r6
   mem_t[32843]= 8'b00001100;
   mem_t[32842]= 8'b00000000;
   mem_t[32841]= 8'b11000000;
   mem_t[32840]= 8'b00011000;

// shift the logic to the right place
// "store" has their own shift logic be careful
//move 00000111 to r7
mem_t[32847]= 8'b10010000;
mem_t[32846]= 8'b00000000;
mem_t[32845]= 8'b11100111;
mem_t[32844]= 8'b10111000;
// move 00010101 to r5
mem_t[32851]= 8'b10010000;
mem_t[32850]= 8'b00000000;
mem_t[32849]= 8'b10100101;
mem_t[32848]= 8'b10111000;
// move 00110001 to r6
mem_t[32855]= 8'b10010000;
mem_t[32854]= 8'b00000000;
```

```
mem_t[32853]= 8'b11000110;
mem_t[32852]= 8'b10111000;
 //the following test is for ADD
 for (i=32856;(i<50000);i+=8)
    begin

   mem_t[i+7]= 8'b00000000;
   mem_t[i+6]= 8'b00101000;
   mem_t[i+5]= 8'b01100110;
   mem_t[i+4]= 8'b11100000;

   mem_t[i+3]= 8'b00000000;
   mem_t[i+2]= 8'b00111000;
   mem_t[i+1]= 8'b01000101;
   mem_t[i+0]= 8'b11100000;
      end
/*
 //the following test is for movhi        (415236101 417333258)
 for (i=32856;(i<50000);i+=8)
  begin
 mem_t[i+7]= 8'b00001010;
 mem_t[i+6]= 8'b00000000;
 mem_t[i+5]= 8'b11100000;
 mem_t[i+4]= 8'b00011000;

 mem_t[i+3]= 8'b00000101;
 mem_t[i+2]= 8'b00000000;
 mem_t[i+1]= 8'b11000000;
 mem_t[i+0]= 8'b00011000;
 end
*/
/*
// the following test is for Addi         (2621767685&2623930383)
      for (i=32856;(i<50000);i+=8)
    begin
   mem_t[i+7]= 8'b00001111;
   mem_t[i+6]= 8'b00000000;
   mem_t[i+5]= 8'b01100110;
   mem_t[i+4]= 8'b10011100;

   mem_t[i+3]= 8'b00000101;
   mem_t[i+2]= 8'b00000000;
   mem_t[i+1]= 8'b01000101;
   mem_t[i+0]= 8'b10011100;
      end
*/
/*
```

```
  // the following test is for Mul      (3762694918&376430630)
    for (i=32856;(i<50000);i+=8)
    begin
   mem_t[i+7]= 8'b00000110;
   mem_t[i+6]= 8'b00000000;
   mem_t[i+5]= 8'b01100110;
   mem_t[i+4]= 8'b11100000;

   mem_t[i+3]= 8'b00000110;
   mem_t[i+2]= 8'b00111011;
   mem_t[i+1]= 8'b01100101;
   mem_t[i+0]= 8'b11100000;
      end
*/
/*
//the following test is for Muli        (2959409167& 2965831686)
  for (i=32856;(i<50000);i+=8)
  begin
 mem_t[i+7]= 8'b00000110;
 mem_t[i+6]= 8'b00000000;
 mem_t[i+5]= 8'b11000111;
 mem_t[i+4]= 8'b10110000;

 mem_t[i+3]= 8'b00001111;
 mem_t[i+2]= 8'b00000000;
 mem_t[i+1]= 8'b01100101;
 mem_t[i+0]= 8'b10110000;
    end
    */
/*
// the following test is for AND
for (i=32856;(i<50000);i+=8)
begin
mem_t[i+7]= 8'b00000011;
mem_t[i+6]= 8'b00101000;
mem_t[i+5]= 8'b01000110;
mem_t[i+4]= 8'b11100000;

mem_t[i+3]= 8'b00000011;
mem_t[i+2]= 8'b00111000;
mem_t[i+1]= 8'b01100101;
mem_t[i+0]= 8'b11100000;
end
*/
/*
// the following test is for ANDi
  for (i=32856;(i<50000);i+=8)
```

```
    begin

   mem_t[i+7]= 8'b00001010;
   mem_t[i+6]= 8'b00000000;
   mem_t[i+5]= 8'b01000110;
   mem_t[i+4]= 8'b10100100;

   mem_t[i+3]= 8'b00011111;
   mem_t[i+2]= 8'b00000000;
   mem_t[i+1]= 8'b01100101;
   mem_t[i+0]= 8'b10100100;
      end
      */
/*
// the following test is for OR
    for (i=32856;(i<50000);i+=8)
          begin

         mem_t[i+7]= 8'b00000100;
         mem_t[i+6]= 8'b00101000;
         mem_t[i+5]= 8'b01000110;
         mem_t[i+4]= 8'b11100000;

         mem_t[i+3]= 8'b00000100;
         mem_t[i+2]= 8'b00111000;
         mem_t[i+1]= 8'b01100101;
         mem_t[i+0]= 8'b11100000;
            end
*/
/*uuuu
for (i=32856;(i<50000);i+=8)
    begin
   mem_t[i+7]= 8'b00011111;
   mem_t[i+6]= 8'b00000000;
   mem_t[i+5]= 8'b01000110;
   mem_t[i+4]= 8'b10101000;

   mem_t[i+3]= 8'b00010000;
   mem_t[i+2]= 8'b00000000;
   mem_t[i+1]= 8'b01100101;
   mem_t[i+0]= 8'b10101000;
      end
*/
/*
// the following test is for XOR
            for (i=32856;(i<50000);i+=8)
          begin
```

```
         mem_t[i+7]= 8'b00000101;
         mem_t[i+6]= 8'b00101000;
         mem_t[i+5]= 8'b01000110;
         mem_t[i+4]= 8'b11100000;

         mem_t[i+3]= 8'b00000101;
         mem_t[i+2]= 8'b00111000;
         mem_t[i+1]= 8'b01100101;
         mem_t[i+0]= 8'b11100000;
            end
 */
/*
// the following test is for XOR
for (i=32856;(i<50000);i+=8)
    begin

   mem_t[i+7]= 8'b00001110;
   mem_t[i+6]= 8'b00000000;
   mem_t[i+5]= 8'b01000110;
   mem_t[i+4]= 8'b10101100;

   mem_t[i+3]= 8'b00000101;
   mem_t[i+2]= 8'b00000000;
   mem_t[i+1]= 8'b01100101;
   mem_t[i+0]= 8'b10101100;
      end
*/

/*
// the following test is for lbs        (2420506631&2422538250)

for (i=32856;(i<50000);i+=8)
    begin

   mem_t[i+7]= 8'b00000111;
   mem_t[i+6]= 8'b00000000;
   mem_t[i+5]= 8'b01000110; //EA(12+7)->r2
   mem_t[i+4]= 8'b10010000;

   mem_t[i+3]= 8'b00001010;
   mem_t[i+2]= 8'b00000000;
   mem_t[i+1]= 8'b01100101;  // EA(3+10)->r3
   mem_t[i+0]= 8'b10010000;
      end

*/
/*
```

```
// the following test is for sub
 for (i=32856;(i<50000);i+=8)
    begin
   mem_t[i+7]= 8'b00000010;
   mem_t[i+6]= 8'b00110000;
   mem_t[i+5]= 8'b01100111;
   mem_t[i+4]= 8'b11100000;

   mem_t[i+3]= 8'b00000010;
   mem_t[i+2]= 8'b00101000;
   mem_t[i+1]= 8'b01000110;
   mem_t[i+0]= 8'b11100000;
      end
*/
/*
// the following test is for sh
// the target address has to be a even
//move 00000111 (15)to r7
mem_t[32847]= 8'b01010000;
mem_t[32846]= 8'b00000000;
mem_t[32845]= 8'b11100111;
mem_t[32844]= 8'b10111000;
// move 000000011 to (3)r5
mem_t[32851]= 8'b01010000;
mem_t[32850]= 8'b00000000;
mem_t[32849]= 8'b10100101;
mem_t[32848]= 8'b10111000;
// move 00110001 to  (12)r6
mem_t[32855]= 8'b01010000;
mem_t[32854]= 8'b00000000;
mem_t[32853]= 8'b11000110;
mem_t[32852]= 8'b10111000;
for (i=32856;(i<50000);i+=8)
begin
mem_t[i+7]= 8'b00000101;
mem_t[i+6]= 8'b00110000;
mem_t[i+5]= 8'b00000101;//r6(12)->EA(5+(3R5)==8+1)
mem_t[i+4]= 8'b11011100;

mem_t[i+3]= 8'b00000111;
mem_t[i+2]= 8'b00101000;
mem_t[i+1]= 8'b00000111;
mem_t[i+0]= 8'b11011100;        //r5(3)->EA(7+(15R7)==22+1)
  end
  */
/*
// the following test is for sb(3624347663&3624216576)
```

```
//move 00000111 (7)to r7
mem_t[32847]= 8'b01010000;
mem_t[32846]= 8'b00000000;
mem_t[32845]= 8'b11100111;
mem_t[32844]= 8'b10111000;
// move 00010101 to (21)r5
mem_t[19551]= 8'b01010000;
mem_t[32850]= 8'b00000000;
mem_t[32849]= 8'b10100101;
mem_t[32848]= 8'b10111000;
// move 00110001 to  (49)r6
mem_t[19555]= 8'b01010000;
mem_t[32854]= 8'b00000000;
mem_t[32853]= 8'b11000110;
mem_t[32852]= 8'b10111000;

for (i=32856;(i<50000);i+=8)
begin

mem_t[i+7]= 8'b00000000;
mem_t[i+6]= 8'b00101000;
mem_t[i+5]= 8'b00000101;
mem_t[i+4]= 8'b11011000;

mem_t[i+3]= 8'b00001111;
mem_t[i+2]= 8'b00101000;
mem_t[i+1]= 8'b00000111;
mem_t[i+0]= 8'b11011000;
  end
*/
/*************************************************high switch
    ***********************************************/
/*
//initialize begins
initial
begin
int i;
i=0;
   //move 3157 to r7
         mem_t[32835]= 8'b01010101;
         mem_t[32834]= 8'b00001100;
         mem_t[32833]= 8'b11100000;
         mem_t[32832]= 8'b00011000;
         // move 12458 (15) to r5
         mem_t[32839]= 8'b10101010;
         mem_t[32838]= 8'b00110000;
         mem_t[32837]= 8'b10100000;
```

```
        mem_t[32836]= 8'b00011000;
   // move 853 (12)to r6
   mem_t[32843]= 8'b01010101;
   mem_t[32842]= 8'b00000011;
   mem_t[32841]= 8'b11000000;
   mem_t[32840]= 8'b00011000;
// shift the logic to the right place
// store has their own shift logic be careful
     //move 00000111 to r7
 mem_t[32847]= 8'b10010000;
 mem_t[32846]= 8'b00000000;
 mem_t[32845]= 8'b11100111;
 mem_t[32844]= 8'b10111000;
 // move 00010101 to r5
 mem_t[32851]= 8'b10010000;
 mem_t[32850]= 8'b00000000;
 mem_t[32849]= 8'b10100101;
 mem_t[32848]= 8'b10111000;
 // move 00110001 to r6
 mem_t[32855]= 8'b10010000;
 mem_t[32854]= 8'b00000000;
 mem_t[32853]= 8'b11000110;
 mem_t[32852]= 8'b10111000;
  */
   /*
   // the following is for movhi
 for (i=32856;(i<50000);i+=8)
  begin
 mem_t[i+7]= 8'b01010101;
 mem_t[i+6]= 8'b00010101;
 mem_t[i+5]= 8'b11100000;
 mem_t[i+4]= 8'b00011000;

 mem_t[i+3]= 8'b10101010;
 mem_t[i+2]= 8'b00101010;
 mem_t[i+1]= 8'b11000000;
 mem_t[i+0]= 8'b00011000;
 end
*/
/*
// the following test is for Add        (3764791296&3762632704)
 for (i=32856;(i<50000);i+=8)
    begin
   mem_t[i+7]= 8'b00000000;
   mem_t[i+6]= 8'b00101000;
   mem_t[i+5]= 8'b01100110;
   mem_t[i+4]= 8'b11100000;
```

```
   mem_t[i+3]= 8'b00000000;
   mem_t[i+2]= 8'b00111000;
   mem_t[i+1]= 8'b01000101;
   mem_t[i+0]= 8'b11100000;
      end
*/
/*
// the following test is for Addi               (262346751&2621767680)
      for (i=32856;(i<50000);i+=8)
    begin
   mem_t[i+7]= 8'b11111111;
   mem_t[i+6]= 8'b00111111;
   mem_t[i+5]= 8'b01100110;
   mem_t[i+4]= 8'b10011100;

   mem_t[i+3]= 8'b00000000;
   mem_t[i+2]= 8'b00000000;
   mem_t[i+1]= 8'b01000101;
   mem_t[i+0]= 8'b10011100;
      end
*/
/*
  // the following test is for Mul      (3764730630&3762694918)
     for (i=32856;(i<50000);i+=8)
    begin
   mem_t[i+7]= 8'b00000110;
   mem_t[i+6]= 8'b00101011;
   mem_t[i+5]= 8'b01000110;
   mem_t[i+4]= 8'b11100000;

   mem_t[i+3]= 8'b00000110;
   mem_t[i+2]= 8'b00111011;
   mem_t[i+1]= 8'b01100101;
   mem_t[i+0]= 8'b11100000;
      end
*/
/*
//the following test is for Muli        (2965839871&2959409152)
  for (i=32856;(i<50000);i+=8)
  begin
 mem_t[i+7]= 8'b11111111;
 mem_t[i+6]= 8'b00011111;
 mem_t[i+5]= 8'b11000111;
 mem_t[i+4]= 8'b10110000;

 mem_t[i+3]= 8'b00000000;
```

```
 mem_t[i+2]= 8'b00000000;
 mem_t[i+1]= 8'b01100101;
 mem_t[i+0]= 8'b10110000;
    end
*/
/*
// the following test is for AND
for (i=32856;(i<50000);i+=8)
begin
mem_t[i+7]= 8'b00000011;
mem_t[i+6]= 8'b00101000;
mem_t[i+5]= 8'b01000110;
mem_t[i+4]= 8'b11100000;

mem_t[i+3]= 8'b00000011;
mem_t[i+2]= 8'b00111000;
mem_t[i+1]= 8'b01100101;
mem_t[i+0]= 8'b11100000;
end
*/
/*
// the following test is for ANDi
for (i=32856;(i<50000);i+=8)
    begin
   mem_t[i+7]= 8'b11000111;
   mem_t[i+6]= 8'b00111001;
   mem_t[i+5]= 8'b01000110;
   mem_t[i+4]= 8'b10100100;

   mem_t[i+3]= 8'b00111000;
   mem_t[i+2]= 8'b00000110;
   mem_t[i+1]= 8'b01100101;
   mem_t[i+0]= 8'b10100100;
      end
*/
/*
// the following test is for OR
     for (i=32856;(i<50000);i+=8)
          begin
         mem_t[i+7]= 8'b00000100;
         mem_t[i+6]= 8'b00101000;
         mem_t[i+5]= 8'b01000110;
         mem_t[i+4]= 8'b11100000;

         mem_t[i+3]= 8'b00000100;
         mem_t[i+2]= 8'b00111000;
         mem_t[i+1]= 8'b01100101;
```

```
         mem_t[i+0]= 8'b11100000;
            end
*/
/*
// the following test is for ORi
 for (i=32856;(i<50000);i+=8)
    begin

   mem_t[i+7]= 8'b11111111;
   mem_t[i+6]= 8'b00111000;
   mem_t[i+5]= 8'b01000110;
   mem_t[i+4]= 8'b10101000;

   mem_t[i+3]= 8'b00000000;
   mem_t[i+2]= 8'b00000111;
   mem_t[i+1]= 8'b01100101;
   mem_t[i+0]= 8'b10101000;
      end
*/
/*
// the following test is for XOR
 for (i=32856;(i<50000);i+=8)
          begin

         mem_t[i+7]= 8'b00000101;
         mem_t[i+6]= 8'b00101000;
         mem_t[i+5]= 8'b01000110;
         mem_t[i+4]= 8'b11100000;

         mem_t[i+3]= 8'b00000101;
         mem_t[i+2]= 8'b00111000;
         mem_t[i+1]= 8'b01100101;
         mem_t[i+0]= 8'b11100000;
            end
  */
/*
// the following test is for XORi
   for (i=32856;(i<50000);i+=8)
    begin

   mem_t[i+7]= 8'b11110000;
   mem_t[i+6]= 8'b00000111;
   mem_t[i+5]= 8'b01000110;
   mem_t[i+4]= 8'b10101100;

   mem_t[i+3]= 8'b00001111;
   mem_t[i+2]= 8'b00111000;
```

```
   mem_t[i+1]= 8'b01100101;
   mem_t[i+0]= 8'b10101100;
      end
*/
/*
// the following test is for lbs        (2420506624 & 2422538495)
    for (i=32856;(i<50000);i+=8)
    begin

   mem_t[i+7]= 8'b00000000;
   mem_t[i+6]= 8'b00000000;
   mem_t[i+5]= 8'b01000110; //49+7->r2
   mem_t[i+4]= 8'b10010000;

   mem_t[i+3]= 8'b11111111;
   mem_t[i+2]= 8'b00000000;
   mem_t[i+1]= 8'b01100101;  // 21+10->r3
   mem_t[i+0]= 8'b10010000;
      end
          */

/*
// the following test is for sub
    //move 00000111 (7)to r7
 mem_t[19511]= 8'b01010000;
 mem_t[32846]= 8'b00000000;
 mem_t[32845]= 8'b11100111;
mem_t[32844]= 8'b10111000;
 // move 00010101 to (21)r5
 mem_t[19503]= 8'b01010000;
 mem_t[32850]= 8'b00000000;
 mem_t[32849]= 8'b10100101;
 mem_t[32848]= 8'b10111000;
 // move 00110001 to  (49)r6
 mem_t[19507]= 8'b01001111;
 mem_t[32854]= 8'b00000000;
 mem_t[32853]= 8'b11000110;
 mem_t[32852]= 8'b10111000;
 for (i=32856;(i<50000);i+=8)
    begin
   mem_t[i+7]= 8'b00000010;
   mem_t[i+6]= 8'b00101000;
   mem_t[i+5]= 8'b01100110;
   mem_t[i+4]= 8'b11100000;

   mem_t[i+3]= 8'b00000010;
   mem_t[i+2]= 8'b00111000;
```

```
   mem_t[i+1]= 8'b01000101;
   mem_t[i+0]= 8'b11100000;
      end
*/
/*
// the following test is for sh

for (i=32856;(i<50000);i+=8)
begin
mem_t[i+7]= 8'b00000111;
mem_t[i+6]= 8'b00101000;
mem_t[i+5]= 8'b00000111;//RB21-->EA7+7=14
mem_t[i+4]= 8'b11011100;

mem_t[i+3]= 8'b00000111;
mem_t[i+2]= 8'b00110000;
mem_t[i+1]= 8'b00000110;//RB49->>EA49+7=56
mem_t[i+0]= 8'b11011100;
  end
*/

/*
// the following test is for sb

for (i=32856;(i<50000);i+=8)
begin

mem_t[i+7]= 8'b11111111;
mem_t[i+6]= 8'b00101000;
mem_t[i+5]= 8'b00000101;
mem_t[i+4]= 8'b11011000;

mem_t[i+3]= 8'b00000000;
mem_t[i+2]= 8'b00101000;
mem_t[i+1]= 8'b00000111;
mem_t[i+0]= 8'b11011000;
  end
*/

/*****************************************************************************************************
mem_t[40835]= 8'b00110000;// branch, jump back about 2000 insturciton
mem_t[40834]= 8'b11111000;
mem_t[40833]= 8'b11111111;
mem_t[40832]= 8'b00000011;

mem_t[40839]= 8'b00110000;// branch, jump back about 2000 insturciton
mem_t[40838]= 8'b11111111;//bf
```

```
mem_t[40837]= 8'b11111111;
mem_t[40836]= 8'b00010011;
/*
mem_t[40835]= 8'b10011100;// branch, jump back about 100 insturciton
mem_t[40834]= 8'b11111111;
mem_t[40833]= 8'b11111111;
mem_t[40832]= 8'b00000011;

mem_t[40835]= 8'10011100;// branch, jump back about 100 insturciton
mem_t[40834]= 8'b11111111;
mem_t[40833]= 8'b11111111;
mem_t[40832]= 8'b00000011;
  */
mem_t[40843]= 8'b00000000;//nop
mem_t[40842]= 8'b00000000;
mem_t[40841]= 8'b00000000;
mem_t[40840]= 8'b00010101;

mem_t[40847]= 8'b00000011;
mem_t[40846]= 8'b00000000;//nop
mem_t[40845]= 8'b00000000;
mem_t[40844]= 8'b00010101;
end
////////////////////////////////////////////////////////////////////////////////////////////////
//integer memout;
//initial
//begin
//  int i;
//  #150000
// for (i = 0; i<2097; i=i+1) //i stands for the lines in the mem
//  begin
//  memout = $fopen ("memory.txt");
//  $fwriteb (memout, mem[i], "\n");
//  end
//  $fclose(memout);
//end
//
//  ////////////////////////////////////////////////////////////////////////
initial
begin
int i;
int j;
i = 0;
j = 32832/4;
for (i = 32832; i<40848; i=i+4)
begin
```

```
data[7:0] =  mem_t[i+3];
data[15:8] = mem_t[i+2];
data[23:16] = mem_t[i+1];
data[31:24] = mem_t[i];
mem [j]=data;
j = j+1;
end

end

endmodule // ram_wb_b3
```

A.1.3 The code of Section 3.6

The method to create these tests is similar to the base power cost tests in Section 3.4. The only difference is that firstly we create a input.c files whose main body is a loop. Then we modified the contents of the machine code of the corresponding input.vmem file directly, rather than jumping to a new space.

The following code is the source c code for test G1.1.

```
#include <stdio.h>
int main()
{
int a;
for (a=0;a<500;a++)
{
a=1+a;
}
asm volatile("l.nop 0x3\n\t");
return a;
 }
```

The following code is main part of the source input.vmem code for test G1.1:

```
@00000894 d7e117fc 9c410000 9c21fff8 9c600000
@00000898 d7e21ff8 00001767 15000000 8462fff8
@0000089c 9c630001 d7e21ff8 8462fff8 9c630001
@000008a0 d7e21ff8 8462fff8 bda3270f 13fffff8
@000008a4 15000000 15000003 8462fff8 a9630000
@000008a8 a8220000 8441fffc 44004800 15000000
}
```

The following code is the source c code for test G1.2, G1.3, and G1.4 but be careful that the input.vmem is the final input file of the Modelsim simulation and the input.c source file is used for the initialization. The contents of input.vmem is changed manually:

```
#include <stdio.h>
int main()
{
int a;
int b=0;
int c=0;
int d=0;
int e=0;
int f=0;
int g=0;
for (a=0;a<50;a++)
{
#/*1*/
a=1+a;
b=b+c;
c=c+d;
d=d+e;
e=e+f;
f=f+g;
g=g+a;//repeat these seven lines another two times (3 times totally)
......
/*3*/
a=1+a;
......
g=g+a;//There are 21 (7*3) lines totally
}
asm volatile("l.nop 0x3\n\t");
return a;
 }
```

The following code is main part of the source input.vmem code for test G1.2:

```
@00000890 9c210004 8521fffc 44004800 15000000
@00000894 d7e117fc 9c410000 9c21ffe0 9c600000
@00000898 d7e21ff4 9c600000 d7e21ff0 9c600000
@0000089c d7e21fec 9c600000 d7e21fe8 9c600000
@000008a0 d7e21fe4 9c600000 d7e21fe0 9c600000
@000008a4 d7e21ff8 00000056 15000000 8462fff8
@000008a8 9c630001 d7e21ff8 8482fff4 8462fff0
@000008ac e0641800 d7e21ff4 8482fff0 8462ffec
@000008b0 e0641800 d7e21ff0 8482ffec 8462ffe8
@000008b4 e0641800 d7e21fec 8482ffe8 8462ffe4
@000008b8 e0641800 d7e21fe8 8482ffe4 8462ffe0
```

```
@000008bc e0641800 d7e21fe4 8482ffe0 8462fff8
@000008c0 e0641800 d7e21fe0 8462fff8 9c630001
@000008c4 d7e21ff8 8482fff4 8462fff0 e0641800
@000008c8 d7e21ff4 8482fff0 8462ffec e0641800
@000008cc d7e21ff0 8482ffec 8462ffe8 e0641800
@000008d0 d7e21fec 8482ffe8 8462ffe4 e0641800
@000008d4 d7e21fe8 8482ffe4 8462ffe0 e0641800
@000008d8 d7e21fe4 8482ffe0 8462fff8 e0641800
@000008dc d7e21fe0 8462fff8 9c630001 d7e21ff8
@000008e0 8482fff4 8462fff0 e0641800 d7e21ff4
@000008e4 8482fff0 8462ffec e0641800 d7e21ff0
@000008e8 8482ffec 8462ffe8 e0641800 d7e21fec
@000008ec 8482ffe8 8462ffe4 e0641800 d7e21fe8
@000008f0 8482ffe4 8462ffe0 e0641800 d7e21fe4
@000008f4 8482ffe0 8462fff8 e0641800 d7e21fe0
@000008f8 8462fff8 9c630001 d7e21ff8 8462fff8
@000008fc bda30031 13ffffaa 15000000 15000003
@00000900 8462fff8 a9630000 a8220000 8441fffc
@00000904 44004800 15000000 d7e14ffc 9c21fffc
```

The following code is main part of the source input.vmem code for test G1.3:

```
@00000890 9c210004 8521fffc 44004800 15000000
@00000894 d7e117fc 9c410000 9c21ffe0 9c600000
@00000898 d7e21ff4 9c600000 d7e21ff0 9c600000
@0000089c d7e21fec 9c600000 d7e21fe8 9c600000
@000008a0 d7e21fe4 9c600000 d7e21fe0 9c600000
@000008a4 d7e21ff8 00000056 15000000 9ce6000f
@000008a8 9ce6000f 9ce6000f e0a62000 9ce6000f
@000008ac e0a62000 9ce6000f e0a62000 9ce6000f
@000008b0 e0a62000 9ce6000f e0a62000 9ce6000f
@000008b4 e0a62000 9ce6000f e0a62000 9ce6000f
@000008b8 e0a62000 9ce6000f e0a62000 9ce6000f
@000008bc e0a62000 9ce6000f e0a62000 9ce6000f
@000008c0 e0a62000 9ce6000f e0a62000 9ce6000f
@000008c4 e0a62000 9ce6000f e0a62000 9ce6000f
@000008c8 e0a62000 9ce6000f e0a62000 9ce6000f
@000008cc e0a62000 9ce6000f e0a62000 9ce6000f
@000008d0 e0a62000 9ce6000f e0a62000 9ce6000f
@000008d4 e0a62000 9ce6000f e0a62000 9ce6000f
@000008d8 e0a62000 9ce6000f e0a62000 9ce6000f
@000008dc e0a62000 9ce6000f e0a62000 9ce6000f
@000008e0 e0a62000 9ce6000f e0a62000 9ce6000f
@000008e4 e0a62000 9ce6000f e0a62000 9ce6000f
@000008e8 e0a62000 9ce6000f e0a62000 9ce6000f
@000008ec e0a62000 9ce6000f e0a62000 9ce6000f
@000008f0 e0a62000 9ce6000f e0a62000 9ce6000f
```

```
@000008f4 8482ffe0 8462fff8 e0641800 d7e21fe0
@000008f8 8462fff8 9c630001 d7e21ff8 8462fff8
@000008fc bda30031 13ffffaa 15000000 15000003
@00000900 8462fff8 a9630000 a8220000 8441fffc
@00000904 44004800 15000000 d7e14ffc 9c21fffc
```

The following code is main part of the source input.vmem code for test G1.4:

```
@00000890 9c210004 8521fffc 44004800 15000000
@00000894 d7e117fc 9c410000 9c21ffe0 9c600000
@00000898 d7e21ff4 9c600000 d7e21ff0 9c600000
@0000089c d7e21fec 9c600000 d7e21fe8 9c600000
@000008a0 d7e21fe4 9c600000 d7e21fe0 9c600000
@000008a4 d7e21ff8 00000056 15000000 9ce6000f
@000008a8 e0a62000 9ce6000f e0a62000 9ce6000f
@000008ac e0a62000 9ce6000f e0a62000 9ce6000f
@000008b0 e0a62000 9ce6000f e0a62000 9ce6000f
@000008b4 e0a62000 9ce6000f e0a62000 9ce6000f
@000008b8 e0a62000 9ce6000f e0a62000 9ce6000f
@000008bc e0a62000 9ce6000f e0a62000 9ce6000f
@000008c0 e0a62000 9ce6000f e0a62000 9ce6000f
@000008c4 e0a62000 9ce6000f e0a62000 9ce6000f
@000008c8 e0a62000 9ce6000f e0a62000 9ce6000f
@000008cc e0a62000 9ce6000f e0a62000 9ce6000f
@000008d0 e0a62000 8482ffe8 8462ffe4 e0641800
@000008d4 d7e21fe8 8482ffe4 8462ffe0 e0641800
@000008d8 d7e21fe4 8482ffe0 8462fff8 e0641800
@000008dc d7e21fe0 8462fff8 9c630001 d7e21ff8
@000008e0 8482fff4 8462fff0 e0641800 d7e21ff4
@000008e4 8482fff0 8462ffec e0641800 d7e21ff0
@000008e8 8482ffec 8462ffe8 e0641800 d7e21fec
@000008ec 8482ffe8 8462ffe4 e0641800 d7e21fe8
@000008f0 8482ffe4 8462ffe0 e0641800 d7e21fe4
@000008f4 8482ffe0 8462fff8 e0641800 d7e21fe0
@000008f8 8462fff8 9c630001 d7e21ff8 8462fff8
@000008fc bda30031 13ffffaa 15000000 15000003
@00000900 8462fff8 a9630000 a8220000 8441fffc
@00000904 44004800 15000000 d7e14ffc 9c21fffc
```

The following code is the source c code for test G2.1, G2.2, and G2.3:

```
#include <stdio.h>
int main()
{
int a;
int b=0;
```

```
int c=0;
int d=0;
int e=0;
int f=0;
int g=0;
for (a=0;a<50;a++)
{
#/*1*/
a=1+a;
b=b+c;
c=c+d;
d=d+e;
e=e+f;
f=f+g;
g=g+a;//repeat these seven lines another four times (5 times totally)
......
/*5*/
a=1+a;
......
g=g+a;//There are 35(7*5) lines totally
}
asm volatile("l.nop 0x3\n\t");
return a;
 }
```

The following code is main part of the source input.vmem code for test G2.1:

```
@00000890 9c210004 8521fffc 44004800 15000000
@00000894 d7e117fc 9c410000 9c21ffe0 9c600000
@00000898 d7e21ff4 9c600000 d7e21ff0 9c600000
@0000089c d7e21fec 9c600000 d7e21fe8 9c600000
@000008a0 d7e21fe4 9c600000 d7e21fe0 9c600000
@000008a4 d7e21ff8 0000008c 15000000 9ce6000f
@000008a8 e0a62000 9ce6000f e0a62000 9ce6000f
@000008ac e0a62000 9ce6000f e0a62000 9ce6000f
@000008b0 e0a62000 9ce6000f e0a62000 9ce6000f
@000008b4 e0a62000 9ce6000f e0a62000 9ce6000f
@000008b8 e0a62000 9ce6000f e0a62000 9ce6000f
@000008bc e0a62000 9ce6000f e0a62000 9ce6000f
@000008c0 e0a62000 9ce6000f e0a62000 9ce6000f
@000008c4 e0a62000 9ce6000f e0a62000 9ce6000f
@000008c8 e0a62000 9ce6000f e0a62000 9ce6000f
@000008cc e0a62000 9ce6000f e0a62000 9ce6000f
@000008d0 e0a62000 9ce6000f e0a62000 9ce6000f
@000008d4 e0a62000 9ce6000f e0a62000 9ce6000f
@000008d8 e0a62000 9ce6000f e0a62000 9ce6000f
@000008dc e0a62000 9ce6000f e0a62000 9ce6000f
```

```
@000008e0 e0a62000 9ce6000f e0a62000 9ce6000f
@000008e4 e0a62000 9ce6000f e0a62000 9ce6000f
@000008e8 e0a62000 9ce6000f e0a62000 9ce6000f
@000008ec e0a62000 9ce6000f e0a62000 9ce6000f
@000008f0 e0a62000 9ce6000f e0a62000 9ce6000f
@000008f4 e0a62000 9ce6000f e0a62000 9ce6000f
@000008f8 8462fff8 9c630001 d7e21ff8 8482fff4
@000008fc 8462fff0 e0641800 d7e21ff4 8482fff0
@00000900 8462ffec e0641800 d7e21ff0 8482ffec
@00000904 8462ffe8 e0641800 d7e21fec 8482ffe8
@00000908 8462ffe4 e0641800 d7e21fe8 8482ffe4
@0000090c 8462ffe0 e0641800 d7e21fe4 8482ffe0
@00000910 8462fff8 e0641800 d7e21fe0 8462fff8
@00000914 9c630001 d7e21ff8 8482fff4 8462fff0
@00000918 e0641800 d7e21ff4 8482fff0 8462ffec
@0000091c e0641800 d7e21ff0 8482ffec 8462ffe8
@00000920 e0641800 d7e21fec 8482ffe8 8462ffe4
@00000924 e0641800 d7e21fe8 8482ffe4 8462ffe0
@00000928 e0641800 d7e21fe4 8482ffe0 8462fff8
@0000092c e0641800 d7e21fe0 8462fff8 9c630001
@00000930 d7e21ff8 8462fff8 bda30031 13ffff74
@00000934 15000000 15000003 8462fff8 a9630000
@00000938 a8220000 8441fffc 44004800 15000000
```

The following code is main part of the source input.vmem code for test G2.2:

```
@00000894 d7e117fc 9c410000 9c21ffe0 9c600000
@00000898 d7e21ff4 9c600000 d7e21ff0 9c600000
@0000089c d7e21fec 9c600000 d7e21fe8 9c600000
@000008a0 d7e21fe4 9c600000 d7e21fe0 9c600000
@000008a4 d7e21ff8 0000008c 15000000 9ce6000f
@000008a8 e0a62000 9ce6000f e0a62000 9ce6000f
@000008ac e0a62000 9ce6000f e0a62000 9ce6000f
@000008b0 e0a62000 9ce6000f e0a62000 9ce6000f
@000008b4 e0a62000 9ce6000f e0a62000 9ce6000f
@000008b8 e0a62000 9ce6000f e0a62000 9ce6000f
@000008bc e0a62000 9ce6000f e0a62000 9ce6000f
@000008c0 e0a62000 9ce6000f e0a62000 9ce6000f
@000008c4 e0a62000 9ce6000f e0a62000 9ce6000f
@000008c8 e0a62000 9ce6000f e0a62000 9ce6000f
@000008cc e0a62000 9ce6000f e0a62000 9ce6000f
@000008d0 e0a62000 9ce6000f e0a62000 9ce6000f
@000008d4 e0a62000 9ce6000f e0a62000 9ce6000f
@000008d8 e0a62000 9ce6000f e0a62000 9ce6000f
@000008dc e0a62000 9ce6000f e0a62000 9ce6000f
@000008e0 e0a62000 9ce6000f e0a62000 9ce6000f
@000008e4 e0a62000 9ce6000f e0a62000 9ce6000f
```

```
@000008e8 e0a62000 9ce6000f e0a62000 9ce6000f
@000008ec e0a62000 9ce6000f e0a62000 9ce6000f
@000008f0 e0a62000 9ce6000f e0a62000 9ce6000f
@000008f4 e0a62000 9ce6000f e0a62000 9ce6000f
@000008f8 e0a62000 9ce6000f e0a62000 9ce6000f
@000008fc e0a62000 9ce6000f e0a62000 9ce6000f
@00000900 e0a62000 9ce6000f e0a62000 9ce6000f
@00000904 e0a62000 9ce6000f e0a62000 9ce6000f
@00000908 e0a62000 9ce6000f e0a62000 9ce6000f
@0000090c e0a62000 9ce6000f e0a62000 9ce6000f
@00000910 e0a62000 9ce6000f e0a62000 9ce6000f
@00000914 e0a62000 9ce6000f e0a62000 9ce6000f
@00000918 e0a62000 9ce6000f e0a62000 9ce6000f
@0000091c e0a62000 9ce6000f e0a62000 9ce6000f
@00000920 e0a62000 9ce6000f e0a62000 9ce6000f
@00000924 e0a62000 9ce6000f e0a62000 9ce6000f
@00000928 e0641800 d7e21fe4 8482ffe0 8462fff8
@0000092c e0641800 d7e21fe0 8462fff8 9c630001
@00000930 d7e21ff8 8462fff8 bda30031 13ffff74
@00000934 15000000 15000003 8462fff8 a9630000
@00000938 a8220000 8441fffc 44004800 15000000
```

The following code is main part of the source input.vmem code for test G2.3:

```
@00000890 9c210004 8521fffc 44004800 15000000
@00000894 d7e117fc 9c410000 9c21ffe0 9c600000
@00000898 d7e21ff4 9c600000 d7e21ff0 9c600000
@0000089c d7e21fec 9c600000 d7e21fe8 9c600000
@000008a0 d7e21fe4 9c600000 d7e21fe0 9c600000
@000008a4 d7e21ff8 0000008c 15000000 8462fff8
@000008a8 9c630001 d7e21ff8 8482fff4 8462fff0
@000008ac e0641800 d7e21ff4 8482fff0 8462ffec
@000008b0 e0641800 d7e21ff0 8482ffec 8462ffe8
@000008b4 e0641800 d7e21fec 8482ffe8 8462ffe4
@000008b8 e0641800 d7e21fe8 8482ffe4 8462ffe0
@000008bc e0641800 d7e21fe4 8482ffe0 8462fff8
@000008c0 e0641800 d7e21fe0 8462fff8 9c630001
@000008c4 d7e21ff8 8482fff4 8462fff0 e0641800
@000008c8 d7e21ff4 8482fff0 8462ffec e0641800
@000008cc d7e21ff0 8482ffec 8462ffe8 e0641800
@000008d0 d7e21fec 8482ffe8 8462ffe4 e0641800
@000008d4 d7e21fe8 8482ffe4 8462ffe0 e0641800
@000008d8 d7e21fe4 8482ffe0 8462fff8 e0641800
@000008dc d7e21fe0 8462fff8 9c630001 d7e21ff8
@000008e0 8482fff4 8462fff0 e0641800 d7e21ff4
@000008e4 8482fff0 8462ffec e0641800 d7e21ff0
@000008e8 8482ffec 8462ffe8 e0641800 d7e21fec
```

```
@000008ec 8482ffe8 8462ffe4 e0641800 d7e21fe8
@000008f0 8482ffe4 8462ffe0 e0641800 d7e21fe4
@000008f4 8482ffe0 8462fff8 e0641800 d7e21fe0
@000008f8 8462fff8 9c630001 d7e21ff8 8482fff4
@000008fc 8462fff0 e0641800 d7e21ff4 8482fff0
@00000900 8462ffec e0641800 d7e21ff0 8482ffec
@00000904 8462ffe8 e0641800 d7e21fec 8482ffe8
@00000908 8462ffe4 e0641800 d7e21fe8 8482ffe4
@0000090c 8462ffe0 e0641800 d7e21fe4 8482ffe0
@00000910 8462fff8 e0641800 d7e21fe0 8462fff8
@00000914 9c630001 d7e21ff8 8482fff4 8462fff0
@00000918 e0641800 d7e21ff4 8482fff0 8462ffec
@0000091c e0641800 d7e21ff0 8482ffec 8462ffe8
@00000920 e0641800 d7e21fec 8482ffe8 8462ffe4
@00000924 e0641800 d7e21fe8 8482ffe4 8462ffe0
@00000928 e0641800 d7e21fe4 8482ffe0 8462fff8
@0000092c e0641800 d7e21fe0 8462fff8 9c630001
@00000930 d7e21ff8 8462fff8 bda30031 13ffff74
@00000934 15000000 15000003 8462fff8 a9630000
@00000938 a8220000 8441fffc 44004800 15000000
```

The following code is the source c code for test G3.1, G3.2, and G3.3:

```
#include <stdio.h>
int main()
{
int a;
int b=0;
int c=0;
int d=0;
int e=0;
int f=0;
int g=0;
for (a=0;a<50;a++)
{
#/*1*/
a=1+a;
b=b+c;
c=c+d;
d=d+e;
e=e+f;
f=f+g;
g=g+a;//repeat these seven lines another four times (6 times totally)
......
/*6*/
a=1+a;
......
```

```
g=g+a;//There are 35(7*6) lines totally
}
asm volatile("l.nop 0x3\n\t");
return a;
 }
```

The following code is main part of the source input.vmem code for test G3.1:

```
@00000890 9c210004 8521fffc 44004800 15000000
@00000894 d7e117fc 9c410000 9c21ffe0 9c600000
@00000898 d7e21ff4 9c600000 d7e21ff0 9c600000
@0000089c d7e21fec 9c600000 d7e21fe8 9c600000
@000008a0 d7e21fe4 9c600000 d7e21fe0 9c600000
@000008a4 d7e21ff8 000000a7 15000000 8462fff8
@000008a8 9c630001 d7e21ff8 8482fff4 8462fff0
@000008ac e0641800 d7e21ff4 8482fff0 8462ffec
@000008b0 e0641800 d7e21ff0 8482ffec 8462ffe8
@000008b4 e0641800 d7e21fec 8482ffe8 8462ffe4
@000008b8 e0641800 d7e21fe8 8482ffe4 8462ffe0
@000008bc e0641800 d7e21fe4 8482ffe0 8462fff8
@000008c0 e0641800 d7e21fe0 8462fff8 9c630001
@000008c4 d7e21ff8 8482fff4 8462fff0 e0641800
@000008c8 d7e21ff4 8482fff0 8462ffec e0641800
@000008cc d7e21ff0 8482ffec 8462ffe8 e0641800
@000008d0 d7e21fec 8482ffe8 8462ffe4 e0641800
@000008d4 d7e21fe8 8482ffe4 8462ffe0 e0641800
@000008d8 d7e21fe4 8482ffe0 8462fff8 e0641800
@000008dc d7e21fe0 8462fff8 9c630001 d7e21ff8
@000008e0 8482fff4 8462fff0 e0641800 d7e21ff4
@000008e4 8482fff0 8462ffec e0641800 d7e21ff0
@000008e8 8482ffec 8462ffe8 e0641800 d7e21fec
@000008ec 8482ffe8 8462ffe4 e0641800 d7e21fe8
@000008f0 8482ffe4 8462ffe0 e0641800 d7e21fe4
@000008f4 8482ffe0 8462fff8 e0641800 d7e21fe0
@000008f8 8462fff8 9c630001 d7e21ff8 8482fff4
@000008fc 8462fff0 e0641800 d7e21ff4 8482fff0
@00000900 8462ffec e0641800 d7e21ff0 8482ffec
@00000904 8462ffe8 e0641800 d7e21fec 8482ffe8
@00000908 8462ffe4 e0641800 d7e21fe8 8482ffe4
@0000090c 8462ffe0 e0641800 d7e21fe4 8482ffe0
@00000910 8462fff8 e0641800 d7e21fe0 8462fff8
@00000914 9c630001 d7e21ff8 8482fff4 8462fff0
@00000918 e0641800 d7e21ff4 8482fff0 8462ffec
@0000091c e0641800 d7e21ff0 8482ffec 8462ffe8
@00000920 e0641800 d7e21fec 8482ffe8 8462ffe4
@00000924 e0641800 d7e21fe8 8482ffe4 8462ffe0
@00000928 e0641800 d7e21fe4 8482ffe0 8462fff8
```

```
@0000092c e0641800 d7e21fe0 8462fff8 9c630001
@00000930 d7e21ff8 8482fff4 8462fff0 e0641800
@00000934 d7e21ff4 8482fff0 8462ffec e0641800
@00000938 d7e21ff0 8482ffec 8462ffe8 e0641800
@0000093c d7e21fec 8482ffe8 8462ffe4 e0641800
@00000940 d7e21fe8 8482ffe4 8462ffe0 e0641800
@00000944 d7e21fe4 8482ffe0 8462fff8 e0641800
@00000948 d7e21fe0 8462fff8 9c630001 d7e21ff8
@0000094c 8462fff8 bda30031 13ffff59 15000000
@00000950 15000003 8462fff8 a9630000 a8220000
@00000954 8441fffc 44004800 15000000 d7e14ffc
```

The following code is main part of the source input.vmem code for test G3.2:

```
@00000890 9c210004 8521fffc 44004800 15000000
@00000894 d7e117fc 9c410000 9c21ffe0 9c600000
@00000898 d7e21ff4 9c600000 d7e21ff0 9c600000
@0000089c d7e21fec 9c600000 d7e21fe8 9c600000
@000008a0 d7e21fe4 9c600000 d7e21fe0 9c600000
@000008a4 d7e21ff8 000000a7 15000000 9ce6000f
@000008a8 e0a62000 9ce6000f e0a62000 9ce6000f
@000008ac e0a62000 9ce6000f e0a62000 9ce6000f
@000008b0 e0a62000 9ce6000f e0a62000 9ce6000f
@000008b4 e0a62000 9ce6000f e0a62000 9ce6000f
@000008b8 e0a62000 9ce6000f e0a62000 9ce6000f
@000008bc e0a62000 9ce6000f e0a62000 9ce6000f
@000008c0 e0a62000 9ce6000f e0a62000 9ce6000f
@000008c4 e0a62000 9ce6000f e0a62000 9ce6000f
@000008c8 e0a62000 9ce6000f e0a62000 9ce6000f
@000008cc e0a62000 9ce6000f e0a62000 9ce6000f
@000008d0 e0a62000 9ce6000f e0a62000 9ce6000f
@000008d4 e0a62000 9ce6000f e0a62000 9ce6000f
@000008d8 e0a62000 9ce6000f e0a62000 9ce6000f
@000008dc e0a62000 9ce6000f e0a62000 9ce6000f
@000008e0 e0a62000 9ce6000f e0a62000 9ce6000f
@000008e4 e0a62000 9ce6000f e0a62000 9ce6000f
@000008e8 e0a62000 9ce6000f e0a62000 9ce6000f
@000008ec e0a62000 9ce6000f e0a62000 9ce6000f
@000008f0 e0a62000 9ce6000f e0a62000 9ce6000f
@000008f4 e0a62000 9ce6000f e0a62000 9ce6000f
@000008f8 e0a62000 9ce6000f e0a62000 9ce6000f
@000008fc e0a62000 9ce6000f e0a62000 9ce6000f
@00000900 e0a62000 9ce6000f e0a62000 9ce6000f
@00000904 e0a62000 9ce6000f e0a62000 9ce6000f
@00000908 e0a62000 9ce6000f e0a62000 9ce6000f
@0000090c e0a62000 9ce6000f e0a62000 9ce6000f
@00000910 e0a62000 9ce6000f e0a62000 9ce6000f
```

```
@00000914 e0a62000 9ce6000f e0a62000 9ce6000f
@00000918 e0a62000 9ce6000f e0a62000 9ce6000f
@0000091c e0a62000 9ce6000f e0a62000 9ce6000f
@00000920 e0a62000 9ce6000f e0a62000 9ce6000f
@00000924 e0a62000 9ce6000f e0a62000 9ce6000f
@00000928 e0a62000 9ce6000f e0a62000 9ce6000f
@0000092c e0a62000 9ce6000f e0a62000 9ce6000f
@00000930 e0a62000 9ce6000f e0a62000 9ce6000f
@00000934 e0a62000 9ce6000f e0a62000 9ce6000f
@00000938 e0a62000 9ce6000f e0a62000 9ce6000f
@0000093c e0a62000 9ce6000f e0a62000 9ce6000f
@00000940 e0a62000 9ce6000f e0a62000 9ce6000f
@00000944 e0a62000 9ce6000f e0a62000 9ce6000f
@00000948 d7e21fe0 8462fff8 9c630001 d7e21ff8
@0000094c 8462fff8 bda30031 13ffff59 15000000
@00000950 15000003 8462fff8 a9630000 a8220000
@00000954 8441fffc 44004800 15000000 d7e14ffc
```

The following code is main part of the source input.vmem code for test G3.3:

```
@00000890 9c210004 8521fffc 44004800 15000000
@00000894 d7e117fc 9c410000 9c21ffe0 9c600000
@00000898 d7e21ff4 9c600000 d7e21ff0 9c600000
@0000089c d7e21fec 9c600000 d7e21fe8 9c600000
@000008a0 d7e21fe4 9c600000 d7e21fe0 9c600000
@000008a4 d7e21ff8 000000a7 15000000 9ce6000f
@000008a8 e0a62000 9ce6000f e0a62000 9ce6000f
@000008ac e0a62000 9ce6000f e0a62000 9ce6000f
@000008b0 e0a62000 9ce6000f e0a62000 9ce6000f
@000008b4 e0a62000 9ce6000f e0a62000 9ce6000f
@000008b8 e0a62000 9ce6000f e0a62000 9ce6000f
@000008bc e0a62000 9ce6000f e0a62000 9ce6000f
@000008c0 e0a62000 9ce6000f e0a62000 9ce6000f
@000008c4 e0a62000 9ce6000f e0a62000 9ce6000f
@000008c8 e0a62000 9ce6000f e0a62000 9ce6000f
@000008cc e0a62000 9ce6000f e0a62000 9ce6000f
@000008d0 e0a62000 9ce6000f e0a62000 9ce6000f
@000008d4 e0a62000 9ce6000f e0a62000 9ce6000f
@000008d8 e0a62000 9ce6000f e0a62000 9ce6000f
@000008dc e0a62000 9ce6000f e0a62000 9ce6000f
@000008e0 e0a62000 9ce6000f e0a62000 9ce6000f
@000008e4 e0a62000 9ce6000f e0a62000 9ce6000f
@000008e8 e0a62000 9ce6000f e0a62000 9ce6000f
@000008ec e0a62000 9ce6000f e0a62000 9ce6000f
@000008f0 e0a62000 9ce6000f e0a62000 9ce6000f
@000008f4 e0a62000 9ce6000f e0a62000 9ce6000f
@000008f8 e0a62000 9ce6000f e0a62000 9ce6000f
```

```
@000008fc e0a62000 9ce6000f e0a62000 9ce6000f
@00000900 e0a62000 9ce6000f e0a62000 9ce6000f
@00000904 e0a62000 9ce6000f e0a62000 9ce6000f
@00000908 e0a62000 9ce6000f e0a62000 9ce6000f
@0000090c e0a62000 9ce6000f e0a62000 9ce6000f
@00000910 e0a62000 9ce6000f e0a62000 8462fff8
@00000914 9c630001 d7e21ff8 8482fff4 8462fff0
@00000918 e0641800 d7e21ff4 8482fff0 8462ffec
@0000091c e0641800 d7e21ff0 8482ffec 8462ffe8
@00000920 e0641800 d7e21fec 8482ffe8 8462ffe4
@00000924 e0641800 d7e21fe8 8482ffe4 8462ffe0
@00000928 e0641800 d7e21fe4 8482ffe0 8462fff8
@0000092c e0641800 d7e21fe0 8462fff8 9c630001
@00000930 d7e21ff8 8482fff4 8462fff0 e0641800
@00000934 d7e21ff4 8482fff0 8462ffec e0641800
@00000938 d7e21ff0 8482ffec 8462ffe8 e0641800
@0000093c d7e21fec 8482ffe8 8462ffe4 e0641800
@00000940 d7e21fe8 8482ffe4 8462ffe0 e0641800
@00000944 d7e21fe4 8482ffe0 8462fff8 e0641800
@00000948 d7e21fe0 8462fff8 9c630001 d7e21ff8
@0000094c 8462fff8 bda30031 13ffff59 15000000
@00000950 15000003 8462fff8 a9630000 a8220000
@00000954 8441fffc 44004800 15000000 d7e14ffc
```

The following code is the source c file for test G4:

```
#include <stdlib.h>
int main()
{
int m1[3][3];
int m2[3][3];
int m3[3][3];
int i=0;

m1[0][0]=1;
m1[0][1]=2;
m1[0][2]=3;
m1[1][0]=4;
m1[1][1]=5;
m1[1][2]=6;
m1[2][0]=7;
m1[2][1]=8;
m1[2][2]=9;

m2[0][0]=1;
m2[0][1]=1;
m2[0][2]=1;
```

```
m2[1][0]=2;
m2[1][1]=2;
m2[1][2]=2;
m2[2][0]=3;
m2[2][1]=3;
m2[2][2]=3;
for (i=0;i<3;i++)
{
m3[i][0]=m1[i][0]*m2[0][0]+m1[i][1]*m2[1][0]+m1[i][2]*m2[2][0];
m3[i][1]=m1[i][0]*m2[0][1]+m1[i][1]*m2[1][1]+m1[i][2]*m2[2][1];
m3[i][2]=m1[i][0]*m2[0][2]+m1[i][1]*m2[1][2]+m1[i][2]*m2[2][2];
}

  //for(i=0; i<3; i++)
  //   printf(" %i\t %i\t %i\n", m3[i][0], m3[i][1], m3[i][2]);
    return 0;
}
```

The following code is main part of the source input.vmem code for test G3.3:

```
@00000894 d7e117fc 9c410000 9c21ff8c 9c600000
@00000898 d7e21ff8 9c600001 d7e21fd4 9c600002
@0000089c d7e21fd8 9c600003 d7e21fdc 9c600004
@000008a0 d7e21fe0 9c600005 d7e21fe4 9c600006
@000008a4 d7e21fe8 9c600007 d7e21fec 9c600008
@000008a8 d7e21ff0 9c600009 d7e21ff4 9c600001
@000008ac d7e21fb0 9c600001 d7e21fb4 9c600001
@000008b0 d7e21fb8 9c600002 d7e21fbc 9c600002
@000008b4 d7e21fc0 9c600002 d7e21fc4 9c600003
@000008b8 d7e21fc8 9c600003 d7e21fcc 9c600003
@000008bc d7e21fd0 9c800000 d7e227f8 00000089
@000008c0 15000000 8482fff8 a8640000 e0631800
@000008c4 e0632000 b8630002 9c82fffc e0641800
@000008c8 9c63ffd8 84830000 8462ffb0 e0a41b06
@000008cc 8482fff8 a8640000 e0631800 e0632000
@000008d0 b8630002 9c82fffc e0641800 9c63ffdc
@000008d4 84830000 8462ffbc e0641b06 e0a51800
@000008d8 8482fff8 a8640000 e0631800 e0632000
@000008dc b8630002 9c82fffc e0641800 9c63ffe0
@000008e0 84830000 8462ffc8 e0641b06 e0a51800
@000008e4 8482fff8 a8640000 e0631800 e0632000
@000008e8 b8630002 9c82fffc e0641800 9c63ff90
@000008ec d4032800 8482fff8 a8640000 e0631800
@000008f0 e0632000 b8630002 9c82fffc e0641800
@000008f4 9c63ffd8 84830000 8462ffb4 e0a41b06
@000008f8 8482fff8 a8640000 e0631800 e0632000
@000008fc b8630002 9c82fffc e0641800 9c63ffdc
```

```
@00000900 84830000 8462ffc0 e0641b06 e0a51800
@00000904 8482fff8 a8640000 e0631800 e0632000
@00000908 b8630002 9c82fffc e0641800 9c63ffe0
@0000090c 84830000 8462ffcc e0641b06 e0a51800
@00000910 8482fff8 a8640000 e0631800 e0632000
@00000914 b8630002 9c82fffc e0641800 9c63ff94
@00000918 d4032800 8482fff8 a8640000 e0631800
@0000091c e0632000 b8630002 9c82fffc e0641800
@00000920 9c63ffd8 84830000 8462ffb8 e0a41b06
@00000924 8482fff8 a8640000 e0631800 e0632000
@00000928 b8630002 9c82fffc e0641800 9c63ffdc
@0000092c 84830000 8462ffc4 e0641b06 e0a51800
@00000930 8482fff8 a8640000 e0631800 e0632000
@00000934 b8630002 9c82fffc e0641800 9c63ffe0
@00000938 84830000 8462ffd0 e0641b06 e0a51800
@0000093c 8482fff8 a8640000 e0631800 e0632000
@00000940 b8630002 9c82fffc e0641800 9c63ff98
@00000944 d4032800 8462fff8 9c630001 d7e21ff8
@00000948 8462fff8 bda30002 13ffff77 15000000
@0000094c 9c600000 a9630000 a8220000 8441fffc
@00000950 44004800 15000000 d7e14ffc 9c21fffc
```

A.1.4 The code of Section 3.7

The following codes are main part of the source input.vmem code for Fibonacci:

```
#include <stdlib.h>
#include <stdio.h>
int fib (int n)
{
        if (n<2)
        return n;
        else
        return (fib(n-1)+fib(n-2));
}
int main()
{
int a;
a=fib (15);
return a;
}
```

The following codes are main part of the source input.vmem code for FIR:

```
#include <stdio.h>
```

```
#define F_LENGTH 20
#define K_LENGTH 5

void firFixed( int *coeffs, int *input, int *output, int length, int
    filterLength )
{
    int acc;     // accumulator for MACs
    int *coeffp; // pointer to coefficients
    int *inputp; // pointer to input samples
    int n;
    int k;

    // apply the filter to each input sample
    for ( n = 0; n < length; n++ ) {
        // calculate output n
        coeffp = coeffs;
        inputp = &input[n];
        acc = 0;
        // perform the multiply-accumulate
        for ( k = 0; k < filterLength; k++ ) {
            acc += (*coeffp++) * (*inputp--);
        }
        output[n] = acc;
    }
}
void main()
{
  int input[] = {0, 0, 1, 1, 1, 1, 1, 5, 1, 1,2,3,4,2,1,4,2,1,5,2};
  int output[F_LENGTH];
  int coeffs[] = { 0, 100, 500, 100, 200};
  firFixed(coeffs, input, output, F_LENGTH,K_LENGTH );

}
```

The following codes are main part of the source input.vmem code for Quicksort:

```
#include <stdio.h>
#include <stdlib.h>
void quicksort(int list[],int m,int n)
{
   int key,i,j,k,temp;
   if( m < n)
   {
      k = (m+n)/2;
      //swap(&list[m],&list[k]);
    temp=list[m];
```

```
    list[m]=list[k];
    list[k]=temp;
      key = list[m];
      i = m+1;
      j = n;
      while(i <= j)
      {
         while((i <= n) && (list[i] <= key))
                i++;
         while((j >= m) && (list[j] > key))
                j--;
         if( i < j)
          //      swap(&list[i],&list[j]);
  {  temp=list[i];
    list[i]=list[j];
    list[j]=temp;}
      }
      // swap two elements
    temp=list[m];
    list[m]=list[j];
    list[j]=temp;
      // recursively sort the lesser list
      quicksort(list,m,j-1);
      quicksort(list,j+1,n);
   }
}
void main()
{ const int MAX_ELEMENTS = 15;
   //int list[MAX_ELEMENTS];
int i = 0;
    int
    list[25]={27,74,17,33,94,18,46,83,65,2,32,53,28,85,99,11,68,67,29,82,21,62,90,59,63};
     // sort the list using quicksort
asm volatile("l.nop 0x3\n\t");
   quicksort(list,0,MAX_ELEMENTS-1);
 asm volatile("l.nop 0x3\n\t");
   // print the result
   //printf("The list after sorting using quicksort algorithm:\n");
   //printlist(list,MAX_ELEMENTS);

}
```

The following codes are main part of the source input.vmem code for Tak:

```
#include <stdio.h>
int tak(int x, int y, int z)
{
```

```
  int a1, a2, a3;
  if (x <= y) return z;
  a1 = tak(x-1,y,z);
  a2 = tak(y-1,z,x);
  a3 = tak(z-1,x,y);
  return tak(a1,a2,a3);
}
main()
{
 take(10,5,3);
 //printf("%d\n", tak(10, 5, 3));
}
```

The following codes are main part of the source input.vmem code for Hanoi:

```
#include <stdio.h>
int Hanoi(int from, int to , int use, int howmany)
{
if (howmany ==1)
{
//printf("Moving a piece from %d to %d \n", from , to);
return 1;
}
else
{
int imovs =0;
imovs += Hanoi(from, use, to, howmany -1);
imovs += Hanoi(from, to , use, 1);
imovs += 1;
imovs += Hanoi(use, to, from, howmany-1);
//printf("imovs %d \n", imovs);
return imovs;
}
}
void main()
{
  Hanoi(3,2,1,5);
}
```

The following codes are module: test_all_top_orpsoc, which aims to test the average register switching bit. More specifically, it is used to recode each register value when it changes.

```
`timescale 1ns/1ps
// synopsys translate_on
`include "or1200_defines.v"
```

```
////////////////////////////////////////////////////////////////////////////////
// Company:
// Engineer:
//
// Create Date:   12:47:42 02/21/2011
// Design Name:   all_top
// Module Name:   D:/openrisc_test_all/test_all_top.v
// Project Name:  openrisc_test_all
// Target Device:
// Tool versions:
// Description:
//
// Verilog Test Fixture created by ISE for module: all_top
//
// Dependencies:
//
// Revision:
// Revision 0.01 - File Created
// Additional Comments:
//
////////////////////////////////////////////////////////////////////////////////

module test_all_top_orpsoc;

        // Inputs
        reg clk;
        reg rst;
`include "orpsoc-defines.v"

        // Instantiate the Unit Under Test (UUT)
        orpsoc_top uut (
      .clk_pad_i   (clk),

      .rst_n_pad_i                      (rst)

        );


always
  #4.5 clk <= ~clk;

               initial
        begin
           rst=1;
           clk=0;
           #200   rst=0;
```

```
    #225.75 rst = 1;

          end

        initial
          begin

    //  #110000       $dumpfile("openrisc_st_65nm_cache1.vcd");
            $dumpvars(0,uut.or1200_top0);
            end

integer
    f0,f1,f2,f3,f4,f5,f6,f7,f8,f9,f10,f11,f12,f13,f14,f15,f16,f17,f18,f19,f20,f21,f22,f23,f24,f25,f26

initial begin
  #100000
  f0 = $fopen("f0.txt","a");
  $fmonitor(f0, "%b", uut.or1200_top0.or1200_cpu.or1200_rf.rf_a.mem[0]);

  f1 = $fopen("f1.txt","a");
  $fmonitor(f1, "%b", uut.or1200_top0.or1200_cpu.or1200_rf.rf_a.mem[1]);

   f2 = $fopen("f2.txt","a");
  $fmonitor(f2, "%b", uut.or1200_top0.or1200_cpu.or1200_rf.rf_a.mem[2]);

   f3 = $fopen("f3.txt","a");
  $fmonitor(f3, "%b", uut.or1200_top0.or1200_cpu.or1200_rf.rf_a.mem[3]);

   f4 = $fopen("f4.txt","a");
  $fmonitor(f4, "%b", uut.or1200_top0.or1200_cpu.or1200_rf.rf_a.mem[4]);

   f5 = $fopen("f5.txt","a");
  $fmonitor(f5, "%b", uut.or1200_top0.or1200_cpu.or1200_rf.rf_a.mem[5]);

   f6 = $fopen("f6.txt","a");
  $fmonitor(f6, "%b", uut.or1200_top0.or1200_cpu.or1200_rf.rf_a.mem[6]);

   f7 = $fopen("f7.txt","a");
  $fmonitor(f7, "%b", uut.or1200_top0.or1200_cpu.or1200_rf.rf_a.mem[7]);

   f8 = $fopen("f8.txt","a");
  $fmonitor(f8, "%b", uut.or1200_top0.or1200_cpu.or1200_rf.rf_a.mem[8]);

   f9 = $fopen("f9.txt","a");
  $fmonitor(f9, "%b", uut.or1200_top0.or1200_cpu.or1200_rf.rf_a.mem[9]);
```

```
 f10 = $fopen("f10.txt","a");
$fmonitor(f10, "%b", uut.or1200_top0.or1200_cpu.or1200_rf.rf_a.mem[10]);

 f11 = $fopen("f11.txt","a");
$fmonitor(f11, "%b", uut.or1200_top0.or1200_cpu.or1200_rf.rf_a.mem[11]);

 f12 = $fopen("f12.txt","a");
$fmonitor(f12, "%b", uut.or1200_top0.or1200_cpu.or1200_rf.rf_a.mem[12]);

 f13 = $fopen("f13.txt","a");
$fmonitor(f13, "%b", uut.or1200_top0.or1200_cpu.or1200_rf.rf_a.mem[13]);

 f14 = $fopen("f14.txt","a");
$fmonitor(f14, "%b", uut.or1200_top0.or1200_cpu.or1200_rf.rf_a.mem[14]);

 f15 = $fopen("f15.txt","a");
$fmonitor(f15, "%b", uut.or1200_top0.or1200_cpu.or1200_rf.rf_a.mem[15]);

 f16 = $fopen("f16.txt","a");
$fmonitor(f16, "%b", uut.or1200_top0.or1200_cpu.or1200_rf.rf_a.mem[16]);

 f17 = $fopen("f17.txt","a");
$fmonitor(f17, "%b", uut.or1200_top0.or1200_cpu.or1200_rf.rf_a.mem[17]);

 f18 = $fopen("f18.txt","a");
$fmonitor(f18, "%b", uut.or1200_top0.or1200_cpu.or1200_rf.rf_a.mem[18]);

 f19 = $fopen("f19.txt","a");
$fmonitor(f19, "%b", uut.or1200_top0.or1200_cpu.or1200_rf.rf_a.mem[19]);

 f20 = $fopen("f20.txt","a");
$fmonitor(f20, "%b", uut.or1200_top0.or1200_cpu.or1200_rf.rf_a.mem[20]);

 f21 = $fopen("f21.txt","a");
$fmonitor(f21, "%b", uut.or1200_top0.or1200_cpu.or1200_rf.rf_a.mem[21]);

 f22 = $fopen("f22.txt","a");
$fmonitor(f22, "%b", uut.or1200_top0.or1200_cpu.or1200_rf.rf_a.mem[22]);

 f23 = $fopen("f23.txt","a");
$fmonitor(f23, "%b", uut.or1200_top0.or1200_cpu.or1200_rf.rf_a.mem[23]);

 f24 = $fopen("f24.txt","a");
$fmonitor(f24, "%b", uut.or1200_top0.or1200_cpu.or1200_rf.rf_a.mem[24]);

 f25 = $fopen("f25.txt","a");
$fmonitor(f25, "%b", uut.or1200_top0.or1200_cpu.or1200_rf.rf_a.mem[25]);
```

```
 f26 = $fopen("f26.txt","a");
$fmonitor(f26, "%b", uut.or1200_top0.or1200_cpu.or1200_rf.rf_a.mem[26]);

 f27 = $fopen("f27.txt","a");
$fmonitor(f27, "%b", uut.or1200_top0.or1200_cpu.or1200_rf.rf_a.mem[27]);

 f28 = $fopen("f28.txt","a");
$fmonitor(f28, "%b", uut.or1200_top0.or1200_cpu.or1200_rf.rf_a.mem[28]);

 f29 = $fopen("f29.txt","a");
$fmonitor(f29, "%b", uut.or1200_top0.or1200_cpu.or1200_rf.rf_a.mem[29]);

 f30 = $fopen("f30.txt","a");
$fmonitor(f30, "%b", uut.or1200_top0.or1200_cpu.or1200_rf.rf_a.mem[30]);

 f31 = $fopen("f31.txt","a");
$fmonitor(f31, "%b", uut.or1200_top0.or1200_cpu.or1200_rf.rf_a.mem[31]);

#200000
 $fclose(f0);
$fclose(f1);
$fclose(f2);
$fclose(f3);
$fclose(f4);
$fclose(f5);
$fclose(f6);
$fclose(f7);
$fclose(f8);
$fclose(f9);
$fclose(f10);
$fclose(f11);
$fclose(f12);
$fclose(f13);
$fclose(f14);
$fclose(f15);
$fclose(f16);
$fclose(f17);
$fclose(f18);
$fclose(f19);
$fclose(f20);
$fclose(f21);
$fclose(f22);
$fclose(f23);
$fclose(f24);
$fclose(f25);
$fclose(f26);
```

```
  $fclose(f27);
  $fclose(f28);
  $fclose(f29);
  $fclose(f30);
  $fclose(f31);

//  $finish;
end


endmodule
```

The following codes aims to test the average register switching bit. More specifically, it is used to analyse the generated files from test_all_top_orpsoc.v and recode the toggle bits number when register value is changed.

```
#include <iostream>
#include <string>
#include <fstream>
#include <sstream>

static const int LINE_LENGTH = 32;
static const std::string LOGFILE_PREFIX = "f";
static const std::string LOGFILE_SUFFIX = ".txt";

inline std::string separator()
{
#ifdef _WIN32
    return "\\";
#else
    return "/";
#endif
}

std::string int2str(int i) {
    std::ostringstream ss;
    ss << i;
    return ss.str();
}

bool scan_file(std::string filename, int* stat) {
    std::ifstream logFile(filename.c_str());
    if (logFile.fail()) return false;

    std::string previousLine = "";
    std::string currentLine = "";
```

```
    int lineNum = 0;
    while (std::getline(logFile, currentLine)) {
        lineNum++;
        if (previousLine != "") {
            // check toggle bits here
            int toggledBitNum = 0;
            for (int i = 0; i < LINE_LENGTH; i++) {
                if (previousLine[i] != currentLine[i]) {
                    toggledBitNum++;
                }
            }
            stat[toggledBitNum]++;
        }
        previousLine = currentLine;
    }
    std::cout << filename << " has " << lineNum << " lines." << std::endl;
    return true;
}

int main(int argc, char* argv[]) {
    int stat[LINE_LENGTH+1];
    int currentStat[LINE_LENGTH+1];
    int toggle_total_number=0;
        int toggle_totol_times=0;
        float toggle_average=0;
    for (int i = 0; i <= LINE_LENGTH; i++) {
        stat[i] = 0;
    }

    static const std::string logDirPath(argv[1]);

    for (int i = 0; i < LINE_LENGTH; i++) {
        std::string fileNum = int2str(i);
        std::string currentLogFileName =
            LOGFILE_PREFIX + fileNum + LOGFILE_SUFFIX;
        // you can print warning for not existing
        // files according to the return value here
        for (int j = 0; j <= LINE_LENGTH; j++) {
            currentStat[j] = 0;
        }
        std::cout << "============================" << std::endl;
        std::string filePath = logDirPath + separator() + currentLogFileName;
        std::cout << "Scan " << filePath << std::endl;
        if (scan_file(filePath, currentStat)) {
            std::cout << filePath << " scanned." << std::endl;
            for (int k = 0; k <= LINE_LENGTH; k++) {
```

```
                std::cout << k << " bits toggled: " << currentStat[k] <<
    std::endl;
                stat[k] += currentStat[k];
            }
        } else {
            std::cout << filePath << " does not exist." << std::endl;
        }
        std::cout << "==============================" << std::endl;
    }

    std::cout << "\nFinal Stat:" << std::endl;
    std::cout << "==============================" << std::endl;

    for (int i = 0; i <=LINE_LENGTH; i++) {
        std::cout << i << " bits toggled: " << stat[i] << std::endl;
                toggle_total_number=toggle_total_number+stat[i]*i;
                toggle_totol_times= toggle_totol_times+stat[i];
    }
        std::cout << "==============================" << std::endl;
        toggle_average=(float)toggle_total_number/toggle_totol_times;
        std::cout << " average bit toggled: " << toggle_average << std::endl;
    return 0;
}
```

A.2 The Test Code of Chapter 4

A.2.1 The code of Section 4.3

The following codes are an example code for testing basic power of instruction $ADD(r)$. For all of the $ALU(r)$ test, the operand is the same. For the tests of Figure 4.4, the number of instructions is changed in order to have different cache miss rate.

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

int main ( void )
{


int p[20]={1,2,3,4};
size_t r1, r2, r3, r4, r5,r6;
printf ("Now initial finished asdfasdfasdf\n");
int i=1;
```

```
asm(" mov r1, #0x11");  //
asm(" mov r2, #0x05");  //2

while(i<0x7ffff)
{
//400 move instructions

asm(" mov r1, #0x11");  //
asm(" mov r2, #0x05");  //2
asm (" mov r5, %[va]"::[va] "r"(p));   // mov the first address of p to
    register r5
asm (" str r1, [r5]");   /// store r1 to the first address of malloce
asm (" str r2, [r5, #4]");

asm (" ADD r3 , r2,  r1");   //1
asm (" ADD r4 , r1 ,  r2");  //2
asm (" ADD r3 , r2,  r1");   //3
asm (" ADD r4 , r1 ,  r2");  //4
asm (" ADD r3 , r2,  r1");       //5
asm (" ADD r4 , r1 ,  r2");  //6
asm (" ADD r3 , r2,  r1");  //7
 asm (" ADD r4 , r1 ,  r2"); //8
asm (" ADD r3 , r2,  r1"); //9
asm (" ADD r4 , r1 ,  r2"); //10
//...........
// repeat N instructions, N is depend on the what cache miss reate is need.

//10
asm (" ADD r3 , r2,  r1");   //1
asm (" ADD r4 , r1 ,  r2");  //2
asm (" ADD r3 , r2,  r1");   //3
asm (" ADD r4 , r1 ,  r2");  //4
asm (" ADD r3 , r2,  r1");       //5
asm (" ADD r4 , r1 ,  r2");  //6
asm (" ADD r3 , r2,  r1");  //7
 asm (" ADD r4 , r1 ,  r2"); //8
asm (" ADD r3 , r2,  r1"); //9
asm (" ADD r4 , r1 ,  r2"); //10

i++;
}


    return(0);
}
```

The following codes are an example code for testing basic power of instruction $ADD(i)$. For all of the $ALU(i)$ test, the operand is the same. For the tests of Figure 4.4, the number of instructions is changed in order to have different cache miss rate.

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

int main ( void )
{
int p[20]={1,2,3,4};
size_t r1, r2, r3, r4, r5,r6;
printf ("Now initial finished asdfasdfasdf\n");
int i=1;
asm(" mov r1, #0x11");  //
asm(" mov r2, #0x05");  //2

while(i<0x7ffff)
{

asm(" mov r1, #0x11");  //
asm(" mov r2, #0x05");  //2
asm (" mov r5, %[va]"::[va] "r"(p));   // mov the first address of p to
    register r5
asm (" str r1, [r5]");   /// store r1 to the first address of malloce
asm (" str r2, [r5, #4]");


 asm (" ADD r3 , r1,  #0x3f");  //1
asm (" ADD r4, r2,  #0xf3"); //2
 asm (" ADD r3 , r1,  #0x3f");  //3
asm (" ADD r4, r2,  #0xf3"); //4
 asm (" ADD r3 , r1,  #0x3f");   //5
asm (" ADD r4, r2,  #0xf3"); //6
 asm (" ADD r3 , r1,  #0x3f");   //7
asm (" ADD r4, r2,  #0xf3"); //8
 asm (" ADD r3 , r1,  #0x3f");//9
asm (" ADD r4, r2,  #0xf3");//10

//.............
// repeat N instructions, N is depend on the what cache miss reate is need.

 asm (" ADD r3 , r1,  #0x3f");  //1
asm (" ADD r4, r2,  #0xf3"); //2
 asm (" ADD r3 , r1,  #0x3f");  //3
asm (" ADD r4, r2,  #0xf3"); //4
 asm (" ADD r3 , r1,  #0x3f");   //5
```

```
asm (" ADD r4, r2,  #0xf3"); //6
 asm (" ADD r3 , r1,  #0x3f");   //7
asm (" ADD r4, r2,  #0xf3"); //8
 asm (" ADD r3 , r1,  #0x3f");//9
asm (" ADD r4, r2,  #0xf3");//10
i++;

}


    return(0);
}
```

The following codes are an example code for testing basic power of instruction *Load*. For the tests of Figure 4.4, the number of instructions is changed in order to have different cache miss rate.

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

int main ( void )
{


int p[20]={1,2,3,4};
size_t r1, r2, r3, r4, r5,r6;
printf ("Now initial finished asdfasdfasdf\n");
int i=1;
asm(" mov r1, #0x11");  //
asm(" mov r2, #0x05");  //2
int a=5,b=10,c=15;
while(i<0x7ffff)
{
//400 move instructions

asm(" mov r1, #0x11");  //
asm(" mov r2, #0x05");  //2
asm (" mov r5, %[va]"::[va] "r"(p));   // mov the first address of p to
    register r5
asm (" str r1, [r5]");   /// store r1 to the first address of malloce
asm (" str r2, [r5, #4]");

asm (" ldr r3,  [r5]");    //1
asm (" ldr r4,  [r5, #4]");      //2
asm (" ldr r3,  [r5]");    //3
```

```
asm (" ldr r4,  [r5, #4]");      //4
asm (" ldr r3,  [r5]");          //5
asm (" ldr r4,  [r5, #4]");      //6
asm (" ldr r3,  [r5]");   //7
asm (" ldr r4,  [r5, #4]");     //8
asm (" ldr r3,  [r5]");    //9
asm (" ldr r4,  [r5, #4]");     //10

//.........
// repeat N instructions, N is depend on the what cache miss reate is need.


asm (" ldr r3,  [r5]");    //1
asm (" ldr r4,  [r5, #4]");      //2
asm (" ldr r3,  [r5]");    //3
asm (" ldr r4,  [r5, #4]");      //4
asm (" ldr r3,  [r5]");          //5
asm (" ldr r4,  [r5, #4]");      //6
asm (" ldr r3,  [r5]");   //7
asm (" ldr r4,  [r5, #4]");     //8
asm (" ldr r3,  [r5]");    //9
asm (" ldr r4,  [r5, #4]");     //10

i++;

}


    return(0);
}
```

The following codes are an example code for testing basic power of instruction *Store*. For the tests of Figure 4.4, the number of instructions is changed in order to have different cache miss rate.

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


int main ( void )
{


int i=1;
```

```
printf ("Now initial the program \n");

size_t r1, r2, r3, r4, r5,r6;
int p[10];
int t;
int *mm=&t;
int *a;
a= (int*)malloc(10*sizeof(int));
printf("the address of a is %u \n", a);

printf("the address of p is %u \n",p);
printf("the address of mm is %u \n",mm);

/*********initial paramter*******/
asm(" mov r1, #0x11");  //
asm(" mov r2, #0x3f");  //2
asm (" mov r5, %[va]"::[va] "r"(p));   // mov the first address of malloc to
    register r5
asm (" str r1, [r5]");   /// store r1 to the first address of malloce
asm (" str r2, [r5, #4]");



while(i<0x7ffff)
{
asm(" mov r1, #0x11");  //
asm(" mov r2, #0x3f");  //2

asm (" mov r5, %[va]"::[va] "r"(p));   // mov the first address of malloc to
    register r5
asm (" str r1, [r5]");   /// store r1 to the first address of malloce
asm (" str r2, [r5, #4]");
asm (" ldr r6, [r5]");

asm (" str r1, [r5]");    //1
asm (" str r2, [r5, #4]");      //2
asm (" str r1, [r5]");    //3
asm (" str r2, [r5, #4]");      //4
asm (" str r1, [r5]");           //5
asm (" str r2, [r5, #4]");      //6
asm (" str r1, [r5]");   //7
asm (" str r2, [r5, #4]");     //8
asm (" str r1, [r5]");    //9
asm (" str r2, [r5, #4]");     //10

//.........
// repeat N instructions, N is depend on the what cache miss reate is need.
```

```
asm (" str r1, [r5]");    //1
asm (" str r2, [r5, #4]");      //2
asm (" str r1, [r5]");    //3
asm (" str r2, [r5, #4]");      //4
asm (" str r1, [r5]");           //5
asm (" str r2, [r5, #4]");      //6
asm (" str r1, [r5]");   //7
asm (" str r2, [r5, #4]");     //8
asm (" str r1, [r5]");    //9
asm (" str r2, [r5, #4]");     //10

i++;

}


    return(0);
}
```

A.2.2 The code of Section 4.4

The following codes is an example to test the Hamming distance 4. The test case main body is the same as Section A.2.1.

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

int main ( void )
{


int p[20]={1,2,3,4};
size_t r1, r2, r3, r4, r5,r6;
printf ("Now initial finished asdfasdfasdf\n");
int i=1;
asm(" mov r1, #0x11");  //
asm(" mov r2, #0x05");  //2

while(i<0x7ffff)
{
//400 move instructions
```

```
asm(" mov r1, #0x11");  //
asm(" mov r2, #0x05");  //2
asm (" mov r5, %[va]"::[va] "r"(p));   // mov the first address of p to
    register r5
asm (" str r1, [r5]");   /// store r1 to the first address of malloce
asm (" str r2, [r5, #4]");

asm (" ADD r3 , r2,  r1");   //1
asm (" ADD r4 , r1 ,  r2");  //2
asm (" ADD r3 , r2,  r1");   //3
asm (" ADD r4 , r1 ,  r2");  //4
asm (" ADD r3 , r2,  r1");       //5
asm (" ADD r4 , r1 ,  r2");  //6
asm (" ADD r3 , r2,  r1");  //7
 asm (" ADD r4 , r1 ,  r2"); //8
asm (" ADD r3 , r2,  r1"); //9
asm (" ADD r4 , r1 ,  r2"); //10
//...........
// repeat N instructions, N is depend on the what cache miss reate is need.

//10
asm (" ADD r3 , r2,  r1");   //1
asm (" ADD r4 , r1 ,  r2");  //2
asm (" ADD r3 , r2,  r1");   //3
asm (" ADD r4 , r1 ,  r2");  //4
asm (" ADD r3 , r2,  r1");       //5
asm (" ADD r4 , r1 ,  r2");  //6
asm (" ADD r3 , r2,  r1");  //7
 asm (" ADD r4 , r1 ,  r2"); //8
asm (" ADD r3 , r2,  r1"); //9
asm (" ADD r4 , r1 ,  r2"); //10

i++;
}


    return(0);
}
```

A.2.3 The code of Section 4.5

The following codes is an example to test the overhead affect of ADD_SUB. The test case main body is the same as Section A.2.1. However, for each case, we set the instruction number 2000, which means the cache usage is 4kB

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

int main ( void )
{
int p[20]={3,12,3,4};
size_t r1, r2, r3, r4, r5,r6;
printf ("Now initial finished asdfasdfasdf\n");
int i=1;
asm(" mov r1, #0x11");  //
asm(" mov r2, #0x05");  //2
int a=5,b=10,c=15;
while(i<0x7ffff)
//400 move instructions
{
asm (" mov r5, %[va]"::[va] "r"(p));   // mov the first address of p to
    register r5
asm (" str r1, [r5]");   /// store r1 to the first address of malloce
asm (" str r2, [r5, #4]");

asm(" mov r1, #0x0");  //
asm(" mov r2, #0x0");  //2

asm (" ADD r3 , r2,  r1");   //1
asm (" SUB  r4 , r1 ,  r2");  //2
asm (" ADD r3 , r2,  r1");   //3
asm (" SUB  r4 , r1 ,  r2");  //4
asm (" ADD r3 , r2,  r1");       //5
asm (" SUB  r4 , r1 ,  r2");  //6
asm (" ADD r3 , r2,  r1");  //7
 asm (" SUB  r4 , r1 ,  r2"); //8
asm (" ADD r3 , r2,  r1"); //9
asm (" SUB  r4 , r1 ,  r2"); //10
//.........
// 2000 instructions totally
asm (" ADD r3 , r2,  r1");   //1
asm (" SUB  r4 , r1 ,  r2");  //2
asm (" ADD r3 , r2,  r1");   //3
asm (" SUB  r4 , r1 ,  r2");  //4
asm (" ADD r3 , r2,  r1");       //5
asm (" SUB  r4 , r1 ,  r2");  //6
asm (" ADD r3 , r2,  r1");  //7
 asm (" SUB  r4 , r1 ,  r2"); //8
asm (" ADD r3 , r2,  r1"); //9
asm (" SUB  r4 , r1 ,  r2"); //10
```

```
}

    return(0);
}
```

A.2.4 The code of Section 4.6

The following codes are an example code for test2, where 25% of the instructions come from logic and 75% come from ALU logic.

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

int main ( void )
{

int p[20]={1,2,3,4};
size_t r1, r2, r3, r4, r5,r6;
printf ("Now initial finished asdfasdfasdf\n");
int i=1;
asm(" mov r1, #0x11");  //
asm(" mov r2, #0x05");  //2
int a=5,b=10,c=15;
while(i<0x7ffff)
{
//400 move instructions

asm(" mov r1, #0x11");  //
asm(" mov r2, #0x05");  //2
asm (" mov r5, %[va]"::[va] "r"(p));   // mov the first address of p to
    register r5
asm (" str r1, [r5]");   /// store r1 to r[5]
asm (" str r2, [r5, #4]");


asm (" ldr r3, [r5,#4] ");
asm (" orr r1 , r1,  r2 ");
asm ("sub r4 , r1,  #0xf  ");
asm ("eor r6 , r2,  #0xf3 ");
asm (" ldr r6, [r5,#8]");
asm (" ADD r3 , r1,  #0xf");
asm (" mov r6 , r1   ");
asm (" and r3 , r2 ,r1 ");
```

```
...........
//2000 instructions totally
asm (" ldr r3, [r5,#4] ");
asm (" orr r1 , r1,  r2 ");
asm ("sub r4 , r1,  #0xf  ");
asm ("eor r6 , r2,  #0xf3 ");
asm (" ldr r6, [r5,#8]");
asm (" ADD r3 , r1,  #0xf");
asm (" mov r6 , r1   ");
asm (" and r3 , r2 ,r1 ");

i++;
}

printf ("Finish the test finished asdfasdfasdf");


    return(0);
}
```

A.2.5 The code of Section 4.7

The source code of Bitcount and Quicksort come from websiteL: MiBench Version 1.0 (http://wwweb.eecs.umich.edu/mibench/).

The following codes are main part of the source input.vmem code for Fibonacci:

```
#include <stdlib.h>
#include <stdio.h>
int fib (int n)
{
        if (n<2)
        return n;
        else
        return (fib(n-1)+fib(n-2));
}
int main()
{
int a;
a=fib (25);
i++;
//printf("the value is %d\n", a);
return a;
}
```

The following codes are main part of the source input.vmem code for FIR:

```
#include <stdio.h>
#define F_LENGTH 4000
#define K_LENGTH 5
void firFixed( int *coeffs, int *input, int *output, int length, int
    filterLength )
{
    int acc;     // accumulator for MACs
    int *coeffp; // pointer to coefficients
    int *inputp; // pointer to input samples
    int n;
    int k;
    // apply the filter to each input sample
    for ( n = 0; n < length; n++ ) {
        // calculate output n
        coeffp = coeffs;
        inputp = &input[n];
        acc = 0;
        // perform the multiply-accumulate
        for ( k = 0; k < filterLength; k++ ) {
            acc += (*coeffp++) * (*inputp--);
        }
        output[n] = acc;
    }
}
void main()
{
int i=0;
  int input[F_LENGTH];
        for (i=0; i<F_LENGTH; i++)
        {
                input[i]=rand()%(F_LENGTH);
                //printf("%d \n", input[i]);
        }
int coeffs[] = { 0, 100, 500, 100, 200};
int output[F_LENGTH];
firFixed(coeffs, input, output, F_LENGTH,K_LENGTH );
  }
```

The following codes are main part of the source input.vmem code for Quicksort:

```
#include <stdio.h>
#include <stdlib.h>
#define MAX_ELEMENTS 4000
void quicksort(int list[],int m,int n)
{
```

```
   int key,i,j,k,temp;
   if( m < n)
   {
      k = (m+n)/2;
      //swap(&list[m],&list[k]);
    temp=list[m];
    list[m]=list[k];
    list[k]=temp;
    key = list[m];
      i = m+1;
      j = n;
      while(i <= j)
      {
         while((i <= n) && (list[i] <= key))
                i++;
         while((j >= m) && (list[j] > key))
                j--;
         if( i < j)
          //      swap(&list[i],&list[j]);
  {  temp=list[i];
    list[i]=list[j];
    list[j]=temp;}
      }
      // swap two elements
    temp=list[m];
    list[m]=list[j];
    list[j]=temp;
      // recursively sort the lesser list
      quicksort(list,m,j-1);
      quicksort(list,j+1,n);
   }
}
void printlist(int list[],int n)
{
   int i;
   for(i=0;i<n;i++)
      printf("%d\t",list[i]);
}
void main()
{
   //int list[MAX_ELEMENTS];
  const char* input_file_name="input_large.dat";
  FILE* file = fopen (input_file_name, "r");
  rewind (file);
  int i = 0;
  int tmp = 0;
  int list[MAX_ELEMENTS];
```

```
  while (!feof (file))
  {
    if (i==MAX_ELEMENTS) break;
    fscanf (file, "%d", &tmp);
    list[i++]=tmp;
  }
  fclose (file);
   printf("The list before sorting is:\n");
   printlist(list,MAX_ELEMENTS);
   // sort the list using quicksort
quicksort(list,0,MAX_ELEMENTS-1);
   // print the result
   printf("The list after sorting using quicksort algorithm:\n");
   printlist(list,MAX_ELEMENTS);
}
```

The following codes are main part of the source input.vmem code for Tak:

```
#include <stdio.h>
int tak(int x, int y, int z)
{
  int a1, a2, a3;
  if (x <= y) return z;
  a1 = tak(x-1,y,z);
  a2 = tak(y-1,z,x);
  a3 = tak(z-1,x,y);
  return tak(a1,a2,a3);
}
main()
{
 int a;
a=tak(3000,2,3);
 //printf("%d\n",a);
}
```

The following codes are main part of the source input.vmem code for Hanoi:

```
#include <stdio.h>
int Hanoi(int from, int to , int use, int howmany)
{
if (howmany ==1)
{
printf("Moving a piece from %d to %d \n", from , to);
return 1;
}|
else
```

```
{
Hanoi(from, use, to, howmany -1);
printf("Moving a piece from %d to %d \n", from , to);
Hanoi(use, to, from, howmany-1);
}
}
void main()
{
int i=0;
 Hanoi(1,2,3,9);
}
```

A.3 The Test Code of Chapter 5

A.3.1 The code of Section 5.4

The following codes is the test code for $ADD(r)$ in Section 5.7: the power consumption of dual-issue restrictions. To test the other opcodes, the only change is the opcode and leave the operand the same.

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



int main ( void )
{


printf ("Now initial the program asdfasdfasdfasdfasdfasdf \n");



printf ("Now initial finished asdfasdfasdf\n");


int i=1;
asm(" mov r1, #0x11");  //
asm(" mov r0, #0x3f");  //2

while(i<0x7ffff)
{
asm(" mov r1, #0x11");  //
```

```
asm(" mov r0, #0x3f");  //2
asm (" add r2 , r1,  r0");   //1
asm (" add r3 , r2 ,  r1");  //2
asm (" add r4 , r3,  r2");   //3
asm (" add r5 , r4 ,  r3");  //4
asm (" add r6 , r5,  r4");       //5
asm (" add r7 , r6 ,  r5");  //6
asm (" add r8 , r7,  r6");  //7
 asm (" add r9 , r8 ,  r7"); //8


// Repeat N times

asm (" add r2 , r1,  r0");   //1
asm (" add r3 , r2 ,  r1");  //2
asm (" add r4 , r3,  r2");   //3
asm (" add r5 , r4 ,  r3");  //4
asm (" add r6 , r5,  r4");       //5
asm (" add r7 , r6 ,  r5");  //6
asm (" add r8 , r7,  r6");  //7
 asm (" add r9 , r8 ,  r7"); //8
i++;

}


    return(0);
}
```

A.3.2 The code of Section 5.7

The following codes is the setting file for Section 5.7: The Power Consumption of Data Cache.

```
#ifndef COMMON_H
#define COMMON_H

#define MEMORY_SPACE 256*1024 /*1KB*/ /*10485760 10240KB*/
#define BLOCK_NUM 64 /*16 <=64k*/ /*32 128k*/  /*64 256k*/  /*128 512k*/
    /*256 1M*/  /*512 2M*/  /*1024 4M*/
#define BLOCK_SIZE (MEMORY_SPACE/BLOCK_NUM)
#define INSTRUCTION_NUM 200 /*1K*/

#define SUB_BLOCK_NUM   4
```

```
#define MIN_SHIFT_WITHIN_BLOCK ((BLOCK_SIZE)/8) /*The min address shift
    between the loading addresses of two adjacent instructions. */
#define MIN_SHIFT_BETWEEN_BLOCK (BLOCK_NUM*SUB_BLOCK_NUM/8) /*The min index
    shift between the loading addresses of two adjacent instructions. */


#define RANDOM_BLOCK_ARRAY_LENGTH 10000
#define TEST_ARRAY_VALUE_RANGE 0xffff

#endif
```

The following codes, generate_code.c, is used to generate the random target address:

```
#include <stdlib.h>
#include <stdio.h>
#include "common.h"

void main()
{
        int r;
        int last=0;
        int isR8=0;
        for (int i=0;i<INSTRUCTION_NUM;i++)
        {
                r = rand()%(BLOCK_SIZE/4);
                while(abs(last-r)<MIN_SHIFT_WITHIN_BLOCK)
                {
                        r = rand()%(BLOCK_SIZE/4);
                }
                last = r;
                if (isR8==1)
                {
                        printf("asm (\" ldr r8, [r5, #%d]\"); \n", r*4);
                        isR8=0;
                } else
                {
                        printf("asm (\" ldr r9, [r5, #%d]\"); \n", r*4);
                        isR8=1;
                }
        }
}
```

The following codes is an example of the generated file from generate_code.c:

```
asm (" ldr r9, [r5, #36]");
asm (" ldr r8, [r5, #4]");
```

```
asm (" ldr r9, [r5, #60]");
asm (" ldr r8, [r5, #8]");
asm (" ldr r9, [r5, #44]");
asm (" ldr r8, [r5, #12]");
asm (" ldr r9, [r5, #48]");
.............
asm (" ldr r8, [r5, #8]");
asm (" ldr r9, [r5, #52]");
asm (" ldr r8, [r5, #8]");
asm (" ldr r9, [r5, #48]");
asm (" ldr r8, [r5, #12]");
asm (" ldr r9, [r5, #52]");
asm (" ldr r8, [r5, #0]");
asm (" ldr r9, [r5, #44]");
asm (" ldr r8, [r5, #0]");
```

The following codes is the final test file:

```
#include <stdlib.h>
#include <stdio.h>
#include <math.h>
#include <sys/time.h>
#include "common.h"
int main ( void )
{
  struct timeval start, end;
        int test_memory_space[MEMORY_SPACE/sizeof(int)];//generate the target
    memery space
// initial the target memory space and the value is less than 0xffff
        for (int i=0; i<MEMORY_SPACE/sizeof(int); i++)
        {
                test_memory_space[i]=rand()%(TEST_ARRAY_VALUE_RANGE);
                //printf("%d\n", test_memory_space[i]);
        }
//////////////////////////////generate the address//////////////////////////
        unsigned int r=0;
        unsigned int last=0;
        unsigned int baseAddress[RANDOM_BLOCK_ARRAY_LENGTH];
        for (int i=0; i<RANDOM_BLOCK_ARRAY_LENGTH; i++)
        {
                r = (unsigned int)(rand()%(SUB_BLOCK_NUM*(BLOCK_NUM-1)+1));
                while(abs(last-r)<MIN_SHIFT_BETWEEN_BLOCK)
                {
                        r = (unsigned
    int)(rand()%(SUB_BLOCK_NUM*(BLOCK_NUM-1)+1));
                }
                last = r;
```

```
                baseAddress[i]=(unsigned
    int)(test_memory_space)+r*BLOCK_SIZE/SUB_BLOCK_NUM;
                //printf(" the r is %d\n", r);
        //printf("target address is %u\n", baseAddress[i]);
        }

//while(i<0x7ffff)
int N=0;
 printf("Completed array init.\n");
srand( time(NULL) );
 gettimeofday(&start, NULL);
        while(N<400)
        {        // mov the first address of malloc to register r5
                for (int i=0; i<RANDOM_BLOCK_ARRAY_LENGTH; i++)
                {
                        asm("mov r8, #0x11");  //
                        asm(" mov r9, #0x3f");  //2
                        asm(" mov r5, %[va]"::[va] "r"(baseAddress[i]));   //
    mov the first address of malloc to register r5

//The following part comes from the generated file of generate_code.c
                        asm (" str r9, [r5, #36]");
                        asm (" ldr r8, [r5, #360]");
                        asm (" ldr r9, [r5, #2672]");
                        ..............................
                        asm (" ldr r8, [r5, #328]");
                        asm (" ldr r9, [r5, #2776]");
                        asm (" ldr r8, [r5, #304]");
                }
        N++;
        }
gettimeofday(&end, NULL);
printf(" took %d seconds, %d us\n", end.tv_sec - start.tv_sec, end.tv_usec -
    start.tv_usec);
    return(0);
}
```

A.3.3 The code of Section 5.8

The following codes are an example code for test1, where 25% of the instructions come from logic and 75% come from ALU logic.

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

```
int main ( void )
{

int p[20]={1,2,3,4};
size_t r1, r2, r3, r4, r5,r6;
printf ("Now initial finished asdfasdfasdf\n");
int i=1;
asm(" mov r1, #0x11");  //
asm(" mov r2, #0x05");  //2
int a=5,b=10,c=15;
while(i<0x7ffff)
{

asm(" mov r1, #0x11");  //
asm(" mov r2, #0x05");  //2
asm (" mov r5, %[va]"::[va] "r"(p));   // mov the first address of p to
    register r5
asm (" str r1, [r5]");   /// store r1 to r[5]
asm (" str r2, [r5, #4]");


asm (" ldr r3, [r5,#4] ");
asm (" orr r1 , r1,  r2 ");
asm ("sub r4 , r1,  #0xf  ");
asm ("eor r6 , r2,  #0xf3 ");
asm (" ldr r6, [r5,#8]");
asm (" ADD r3 , r1,  #0xf");
asm (" mov r6 , r1   ");
asm (" and r3 , r2 ,r1 ");
...........
//the number of instructions changes for the best case and worst case
// For the best case test, the number is 2000 (size=2000*4B)
// For the worst case test, the number is 16000 (size=16000*4B)
...........
asm (" ldr r3, [r5,#4] ");
asm (" orr r1 , r1,  r2 ");
asm ("sub r4 , r1,  #0xf  ");
asm ("eor r6 , r2,  #0xf3 ");
asm (" ldr r6, [r5,#8]");
asm (" ADD r3 , r1,  #0xf");
asm (" mov r6 , r1   ");
asm (" and r3 , r2 ,r1 ");

i++;
}
```

```
printf ("Finish the test finished asdfasdfasdf");


    return(0);
}
```

A.3.4 The code of Section 5.9

The source c files for each benchmark are the same as Chapter A.2.

A.4 The Test Code of Chapter 6

A.4.1 The code of Section 6.4

The example code of the tests are shown in Section 7.2.

A.4.2 The code of Section 6.7

The source code of SPLASH 2 comes from the website:The Modified SPLASH-2 Home Page (http://www.capsl.udel.edu/splash/Download.html).

References

[1] V. Tiwari and M. Tien-Chien Lee, "Power analysis of a 32-bit embedded microcontroller," in *Design Automation Conference, 1995. Proceedings of the ASP-DAC '95/CHDL '95/VLSI '95., IFIP International Conference on Hardware Description Languages; IFIP International Conference on Very Large Scale Integration., Asian and South Pacific*, pp. 141 –148, Aug-1 Sep 1995.

[2] D. Sarta, D. Trifone, and G. Ascia, "A data dependent approach to instruction level power estimation," in *Low-Power Design, 1999. Proceedings. IEEE Alessandro Volta Memorial Workshop on*, pp. 182 –190, Mar. 1999.

[3] S. Abrar, "Cycle-accurate energy model and source-independent characterization methodology for embedded processors," in *VLSI Design, 2004. Proceedings. 17th International Conference on*, pp. 749–752, 2004.

[4] J. Koomey, S. Berard, M. Sanchez, and H. Wong, "Implications of historical trends in the electrical efficiency of computing," *Annals of the History of Computing, IEEE*, vol. 33, pp. 46–54, March 2011.

[5] N. Kavvadias, P. Neofotistos, S. Nikolaidis, K. Kosmatopoulos, and T. Laopoulos, "Measurements analysis of the software-related power consumption in microprocessors," in *Instrumentation and Measurement Technology Conference, 2003. IMTC '03. Proceedings of the 20th IEEE*, vol. 2, pp. 981 – 986 vol.2, may 2003.

[6] N. Kim, T. Austin, D. Baauw, T. Mudge, K. Flautner, J. Hu, M. Irwin, M. Kandemir, and V. Narayanan, "Leakage current: Moore's law meets static power," *Computer*, vol. 36, pp. 68–75, Dec 2003.

[7] V. Tiwari, S. Malik, A. Wolfe, and M.-C. Lee, "Instruction level power analysis and optimization of software," in *VLSI Design, 1996. Proceedings., Ninth International Conference on*, pp. 326 –328, Jan 1996.

[8] V. Tiwari, S. Malik, and A. Wolfe, "Power analysis of embedded software: a first step towards software power minimization," *Very Large Scale Integration (VLSI) Systems, IEEE Transactions on*, vol. 2, pp. 437 –445, Dec. 1994.

[9] S. Nikolaidis, N. Kavvadias, T. Laopoulos, L. Bisdounis, and S. Blionas, "Instruction level energy modeling for pipelined processors," *Journal of Embedded Computing*, vol. 1, no. 3, pp. 317–324, 2005.

[10] M. Hill and M. Marty, "Amdahl's law in the multicore era," *Computer*, vol. 41, no. 7, pp. 33–38, 2008.

[11] S. Penolazzi, L. Bolognino, and A. Hemani, "Energy and performance model of a sparc leon3 processor," in *Digital System Design, Architectures, Methods and Tools, 2009. DSD '09. 12th Euromicro Conference on*, pp. 651 –656, Aug. 2009.

[12] J. M. Acevedo Pati O. and C. A. A.J., "Static simulation: A method for power and energy estimation in embedded microprocessors," in *Circuits and Systems (MWSCAS), 2010 53rd IEEE International Midwest Symposium on*, pp. 41 –44, aug. 2010.

[13] N. Chang, K. Kim, and H. G. Lee, "Cycle-accurate energy consumption measurement and analysis: Case study of arm7tdmi," in *Proceedings of the 2000 International Symposium on Low Power Electronics and Design*, ISLPED '00, (New York, NY, USA), pp. 185–190, ACM, 2000.

[14] S. Lee, A. Ermedahl, S. Min, and N. Chang, "An accurate instruction-level energy consumption model for embedded risc processors," vol. 36, no. 8, pp. 1–10, 2001.

[15] M. Bazzaz, M. Salehi, and A. Ejlali, "An accurate instruction-level energy estimation model and tool for embedded systems," *Instrumentation and Measurement, IEEE Transactions on*, vol. 62, no. 7, pp. 1927–1934, 2013.

[16] S. Steinke, M. Knauer, L. Wehmeyer, and P. Marwedel, "An accurate and fine grain instruction-level energy model supporting software optimizations," in *Proc. of PATMOS*, Citeseer, 2001.

[17] N. Chang, K. Kim, and H. G. Lee, "Cycle-accurate energy measurement and characterization with a case study of the arm7tdmi [microprocessors]," *Very Large Scale Integration (VLSI) Systems, IEEE Transactions on*, vol. 10, pp. 146–154, April 2002.

[18] V. Konstantakos, A. Chatzigeorgiou, S. Nikolaidis, and T. Laopoulos, "Energy consumption estimation in embedded systems," *Instrumentation and Measurement, IEEE Transactions on*, vol. 57, pp. 797–804, April 2008.

[19] A. of OpenCores, "Openrisc 1000 architecture manual." URL=http://opencores.org/openrisc,or1200, May 2006.

[20] D. Cormie, "The ARM11 Microarchitecture." URL=http://www.cs.uiuc.edu/class/fa05/cs433ug/PROCESSORS/ARM%2011%20MicroArchitecture.pdf, 2002.

[21] J. L. Hennessy and D. A. Patterson, *Computer Architecture, Fifth Edition: A Quantitative Approach*. San Francisco, CA, USA: Morgan Kaufmann Publishers Inc., 5th ed., 2011.

[22] ARM Holdings plc, "Enabling innovation everywhere." URL=http://financialreports.arm.com/pdfs/ARM_StrategicReport_2013.pdf, 2013.

[23] S. Nikolaidis and T. Laopoulos, "Instruction-level power consumption estimation embedded processors low-power applications," in *Intelligent Data Acquisition and Advanced Computing Systems: Technology and Applications, International Workshop on, 2001.*, pp. 139 –142, 2001.

[24] R. Sridhar and K. Schindler, "Instruction level power model and its application to general purpose processors," in *Signals, Systems amp; Computers, 1997. Conference Record of the Thirty-First Asilomar Conference on*, vol. 1, pp. 753 –756 vol.1, nov. 1997.

[25] A. Varma, E. Debes, I. Kozintsev, and B. Jacob, "Instruction-level power dissipation in the intel xscale embedded microprocessor," in *Electronic Imaging 2005*, pp. 1–8, International Society for Optics and Photonics, 2005.

[26] T. Laopoulos, P. Neofotistos, C. Kosmatopoulos, and S. Nikolaidis, "Measurement of current variations for the estimation of software-related power consumption [embedded processing circuits]," *Instrumentation and Measurement, IEEE Transactions on*, vol. 52, pp. 1206 – 1212, aug. 2003.

[27] T. Laopoulos, P. Neofotistos, C. Kosmatopoulos, and S. Nikolaidis, "Current variations measurements for the estimation of software-related power consumption," in *Instrumentation and Measurement Technology Conference, 2002. IMTC/2002. Proceedings of the 19th IEEE*, vol. 2, pp. 1637 – 1642 vol.2, 2002.

[28] H. Mehta, R. Owens, and M. Irwin, "Instruction level power profiling," in *Acoustics, Speech, and Signal Processing, 1996. ICASSP-96. Conference Proceedings., 1996 IEEE International Conference on*, vol. 6, pp. 3326 –3329 vol. 6, may 1996.

[29] M.-C. Lee, V. Tiwari, S. Malik, and M. Fujita, "Power analysis and minimization techniques for embedded dsp software," *Very Large Scale Integration (VLSI) Systems, IEEE Transactions on*, vol. 5, pp. 123–135, March 1997.

[30] B. Klass, D. Thomas, H. Schmit, and D. Nagle, "Modeling inter-instruction energy effects in a digital signal processor," in *Power Driven Microarchitecture Workshop in conjunction with the 25th International Symposium on Computer Architecture*, vol. 2, pp. 1094 –1097, Oct. 1998.

[31] A. Bona, M. Sami, D. Sciuto, C. Silvano, V. Zaccaria, and R. Zafalon, "Reducing the complexity of instruction-level power models for vliw processors," *Design Automation for Embedded Systems*, vol. 10, no. 1, pp. 49–67, 2005.

[32] M. Sami, D. Sciuto, C. Silvano, and V. Zaccaria, "An instruction-level energy model for embedded vliw architectures," *Computer-Aided Design of Integrated Circuits and Systems, IEEE Transactions on*, vol. 21, pp. 998 – 1010, sep 2002.

[33] A. Bona, M. Sami, D. Sciuto, C. Silvano, V. Zaccaria, and R. Zafalon, "An instruction-level methodology for power estimation and optimization of embedded vliw cores," in *Design, Automation and Test in Europe Conference and Exhibition, 2002. Proceedings*, p. 1128, 2002.

[34] M. P. Giuseppe Ascia, Vincenzo Catania and D. Sarta, "An instruction-level power analysis model with data dependency," in *VLSI Design*, vol. 12, pp. 245– 273, Aug. 2001.

[35] MOTOROLA, "Dsp 56000 24-bit digital signal processor family manual." URL=http://cache.freescale.com/files/dsp/doc/inactive/DSP56000UM.pdf, 1995.

[36] ARM, "Arm7tdmi technical reference manual." URL=http://infocenter.arm.com/help/topic/com.arm.doc.ddi0210c/DDI0210B.pdf, 1994-2001.

[37] A. Bona, M. Sami, D. Sciuto, C. Silvano, V. Zaccaria, and R. Zafalon, "Energy estimation and optimization of embedded vliw processors based on instruction clustering," in *Design Automation Conference, 2002. Proceedings. 39th*, pp. 886 – 891, 2002.

[38] Analog Devices Inc., "Dsp microcomputer adsp-2189." URL=http://www.analog.com/static/imported-files/data_sheets/ADSP-2189M.pdf, 2000.

[39] R. Luo, H. Luo, H. Yang, and Y. Xie, "An instruction-level analytical power model for designing the low power systems on a chip," in *ASIC, 2005. ASICON 2005. 6th International Conference On*, July 2005.

[40] G. A. Seber and A. J. Lee, *Linear regression analysis*, vol. 936. John Wiley & Sons, 2012.

[41] X. Yan, *Linear regression analysis: theory and computing*. World Scientific, 2009.

[42] W. Bircher, M. Valluri, J. Law, and L. John, "Runtime identification of microprocessor energy saving opportunities," in *Low Power Electronics and Design, 2005. ISLPED'05. Proceedings of the 2005 International Symposium on*, pp. 275–280, IEEE, 2005.

[43] Tensilica Inc., "Xtensa instruction set architecture (isa) reference manual." URL=http://0x04.net/~mwk/doc/xtensa.pdf, 2010.

[44] Y. Fei, S. Ravi, A. Raghunathan, and N. Jha, "A hybrid energy-estimation technique for extensible processors," *Computer-Aided Design of Integrated Circuits and Systems, IEEE Transactions on*, vol. 23, pp. 652 – 664, may 2004.

[45] S. Nikolaidis, N. Kavvadias, P. Neofotistos, K. Kosmatopoulos, T. Laopoulos, and L. Bisdounis, "Instrumentation set-up for instruction level power modeling," *Integrated Circuit Design. Power and Timing Modeling, Optimization and Simulation*, pp. 263–290, 2002.

[46] C. X. Huang, B. Zhang, A.-C. Deng, and B. Swirski, "The design and implementation of powermill," in *Proceedings of the 1995 International Symposium on Low Power Design*, ISLPED '95, (New York, NY, USA), pp. 105–110, ACM, 1995.

[47] M. E. A. Ibrahim, M. Rupp, and H. A. H. Fahmy, "A precise high-level power consumption model for embedded systems software," *EURASIP J. Embedded Syst.*, vol. 2011, pp. 1:1–1:14, Jan. 2011.

[48] J. Laurent, E. Senn, N. Julien, E. Martin, *et al.*, "High-level energy estimation for dsp systems," in *PATMOS*, 2001.

[49] E. Senn, N. Julien, J. Laurent, and E. Martin, "Power consumption estimation of a c program for data-intensive applications," in *Integrated Circuit Design. Power and Timing Modeling, Optimization and Simulation* (B. Hochet, A. Acosta, and M. Bellido, eds.), vol. 2451 of *Lecture Notes in Computer Science*, pp. 332–341, Springer Berlin Heidelberg, 2002.

[50] E. Senn, J. Laurent, N. Julien, and E. Martin, "Softexplorer: Estimating and optimizing the power and energy consumption of a c program for dsp applications," *EURASIP J. Appl. Signal Process.*, vol. 2005, pp. 2641–2654, Jan. 2005.

[51] N. Julien, J. Laurent, E. Senn, and E. Martin, "Power consumption modeling and characterization of the ti c6201," *Micro, IEEE*, vol. 23, no. 5, pp. 40–49, 2003.

[52] C. Brandolese, W. Fomacian, F. Salice, and D. Sciuto, "An instruction-level functionality-based energy estimation model for 32-bits microprocessors," in *Design Automation Conference, 2000. Proceedings 2000. 37th*, pp. 346 –350, 2000.

[53] C. Brandolese, F. Salice, W. Fornaciari, and D. Sciuto, "Static power modeling of 32-bit microprocessors," *Computer-Aided Design of Integrated Circuits and Systems, IEEE Transactions on*, vol. 21, pp. 1306 – 1316, nov 2002.

[54] D. Brooks, V. Tiwari, and M. Martonosi, "Wattch: a framework for architectural-level power analysis and optimizations," in *Computer Architecture, 2000. Proceedings of the 27th International Symposium on*, pp. 83 –94, june 2000.

[55] W. Ye, N. Vijaykrishnan, M. Kandemir, and M. J. Irwin, "The design and use of simplepower: a cycle-accurate energy estimation tool," in *Proceedings of the 37th Annual Design Automation Conference*, DAC '00, (New York, NY, USA), pp. 340–345, ACM, 2000.

[56] R. Y. Chen, M. J. Irwin, and R. S. Bajwa, "Architecture-level power estimation and design experiments," *ACM Trans. Des. Autom. Electron. Syst.*, vol. 6, pp. 50–66, Jan. 2001.

[57] T. Sato, M. Nagamatsu, and H. Tago, "Power and performance simulator: Esp and its application for 100 mips/w class risc design," in *Low Power Electronics, 1994. Digest of Technical Papers., IEEE Symposium*, pp. 46–47, 1994.

[58] S. Gurumurthi, A. Sivasubramaniam, M. Irwin, N. Vijaykrishnan, and M. Kandemir, "Using complete machine simulation for software power estimation: the softwatt approach," in *High-Performance Computer Architecture, 2002. Proceedings. Eighth International Symposium on*, pp. 141–150, 2002.

[59] J. Nunez-Yanez and G. Lore, "Enabling accurate modeling of power and energy consumption in an arm-based system-on-chip," *Microprocessors and Microsystems*, vol. 37, no. 3, pp. 319–332, 2013.

[60] A. Carroll and G. Heiser, "An analysis of power consumption in a smartphone.," in *USENIX annual technical conference*, pp. 1–14, 2010.

[61] I. Lee, H. Kim, P. Yang, S. Yoo, E.-Y. Chung, K.-M. Choi, J.-T. Kong, and S.-K. Eo, "Powervip: Soc power estimation framework at transaction level," in *Proceedings of the 2006 Asia and South Pacific Design Automation Conference*, ASP-DAC '06, (Piscataway, NJ, USA), pp. 551–558, IEEE Press, 2006.

[62] A. Shye, B. Scholbrock, and G. Memik, "Into the wild: Studying real user activity patterns to guide power optimizations for mobile architectures," in *Microarchitecture, 2009. MICRO-42. 42nd Annual IEEE/ACM International Symposium on*, pp. 168–178, Dec 2009.

[63] Y. Xiao, R. Bhaumik, Z. Yang, M. Siekkinen, P. Savolainen, and A. Yla-Jaaski, "A system-level model for runtime power estimation on mobile devices," in *Green Computing and Communications (GreenCom), 2010 IEEE/ACM Int'l Conference on Int'l Conference on Cyber, Physical and Social Computing (CPSCom)*, pp. 27–34, Dec 2010.

[64] Intel, "80960ja/jf/jd/js/jc/jt 3.3v embedded 32-bit microprocessor." URL=http://download.intel.com/design/i960/datashts/27315906.pdf, 1998.

[65] Intel, "80960ha/hd/ht 32-bit high-performance superscalar processor." URL=http://datasheet.octopart.com/UG80960HD6616SL2GN-Intel-datasheet-39477.pdf, 1998.

[66] J. Russell and M. Jacome, "Software power estimation and optimization for high performance, 32-bit embedded processors," in *Computer Design: VLSI in Computers and Processors, 1998. ICCD '98. Proceedings. International Conference on*, pp. 328–333, 1998.

[67] S. Sultan and S. Masud, "Rapid software power estimation of embedded pipelined processor through instruction level power model," in *Performance Evaluation of Computer Telecommunication Systems, 2009. SPECTS 2009. International Symposium on*, vol. 41, pp. 27 –34, July 2009.

[68] ARM, "Strongarm data sheet v2.0." URL=http://www.home.marutan.net/arcemdocs/sa110.pdf, 1996.

[69] A. Sinha, N. Ickes, and A. Chandrakasan, "Instruction level and operating system profiling for energy exposed software," *Very Large Scale Integration (VLSI) Systems, IEEE Transactions on*, vol. 11, pp. 1044 –1057, Dec. 2003.

[70] MCT EElektronikladen GbR, "Hc08 welcome kit user manual." URL=http://elmicro.com/en/kit08.html, 2001.

[71] Y. Shao and D. Brooks, "Energy characterization and instruction-level energy model of intel's xeon phi processor," in *Low Power Electronics and Design (ISLPED), 2013 IEEE International Symposium on*, pp. 389–394, Sept 2013.

[72] G. Coley, "Beaglebone rev a6 system reference manual." URL=http://beagleboard.org/static/beaglebone/latest/Docs/Hardware/BONE_SRM.pdf, 2012.

[73] I. Corporation, "Intel pentium 4 processor-manuals." URL=ftp://download.intel.com/design/Pentium4/papers/24943801.pdf, 2002.

[74] N. Kavvadias, P. Neofotistos, S. Nikolaidis, C. Kosmatopoulos, and T. Laopoulos, "Measurements analysis of the software-related power consumption in microprocessors," *Instrumentation and Measurement, IEEE Transactions on*, vol. 53, pp. 1106 – 1112, aug. 2004.

[75] ARM, "cortex-a8 technical reference manual." URL=http://infocenter.arm.com/help/topic/com.arm.doc.ddi0344d/DDI0344D_cortex_a8_r2p1_trm.pdf, 2006-2007.

[76] Intel, "History-intel-chips-timeline-poster." URL=http://www.intel.la/content/dam/www/public/us/en/documents/corporate-information/history-intel-chips-timeline-poster.pdf, 2012.

[77] N. H. Weste and D. M. Harris, *CMOS VLSI design: a circuits and systems perspective*. Pearson Education India, 2005.

[78] A. Butko, R. Garibotti, L. Ost, and G. Sassatelli, "Accuracy evaluation of gem5 simulator system," in *Reconfigurable Communication-centric Systems-on-Chip (ReCoSoC), 2012 7th International Workshop on*, pp. 1–7, 2012.

[79] D. Lampret, "OpenRISC 1200 ip core specification." URL=http://opencores.org/svnget,or1k?file=/trunk/or1200/doc/openrisc1200_spec.doc, Jan 2011.

[80] Texas Instruments, "Cmos power consumptino and c_{pd}calculation." URL=http://www.ti.com/lit/an/scaa035b/scaa035b.pdf, June 1997.

[81] M. Horowitz, T. Indermaur, and R. Gonzalez, "Low-power digital design," in *Low Power Electronics, 1994. Digest of Technical Papers., IEEE Symposium*, pp. 8–11, IEEE, 1994.

[82] G. Yip, "Expanding the synopsys primetime solution with power analysis." URL=http://www.synopsys.com/Tools/Implementation/SignOff/CapsuleModule/ptpx_wp.pdf, June 2006.

[83] M. Ros and P. Sutton, "A hamming distance based vliw/epic code compression technique," in *Proceedings of the 2004 International Conference on Compilers, Architecture, and Synthesis for Embedded Systems*, CASES '04, (New York, NY, USA), pp. 132–139, ACM, 2004.

[84] J. Lee and A. Smith, "Branch prediction strategies and branch target buffer design," *Computer*, vol. 17, no. 1, pp. 6–22, 1984.

[85] D. Jimenez and C. Lin, "Dynamic branch prediction with perceptrons," in *High-Performance Computer Architecture, 2001. HPCA. The Seventh International Symposium on*, pp. 197–206, 2001.

[86] D. Callahan, K. Kennedy, and A. Porterfield, "Software prefetching," *SIGPLAN Not.*, vol. 26, pp. 40–52, Apr. 1991.

[87] D. Callahan, K. Kennedy, and A. Porterfield, "Software prefetching," in *Proceedings of the fourth international conference on Architectural support for programming languages and operating systems*, ASPLOS IV, (New York, NY, USA), pp. 40–52, ACM, 1991.

[88] I.-C. Chen, C.-C. Lee, and T. Mudge, "Instruction prefetching using branch prediction information," in *Computer Design: VLSI in Computers and Processors, 1997. ICCD '97. Proceedings., 1997 IEEE International Conference on*, pp. 593–601, 1997.

[89] Q. Zhu, F. David, C. Devaraj, Z. Li, Y. Zhou, and P. Cao, "Reducing energy consumption of disk storage using power-aware cache management," in *Software, IEE Proceedings-*, pp. 118–118, 2004.

[90] M. Pedram, "Power minimization in ic design: principles and applications," *ACM Trans. Des. Autom. Electron. Syst.*, vol. 1, pp. 3–56, Jan. 1996.

[91] D.-J. Lee, M.-C. Kim, and I. L. Markov, "Low-power clock trees for cpus," in *Proceedings of the International Conference on Computer-Aided Design*, pp. 444–451, IEEE Press, 2010.

[92] T. Li and L. K. John, "Run-time modeling and estimation of operating system power consumption," *SIGMETRICS Perform. Eval. Rev.*, vol. 31, pp. 160–171, June 2003.

[93] A. Lungu, P. Bose, A. Buyuktosunoglu, and D. J. Sorin, "Dynamic power gating with quality guarantees," in *Proceedings of the 14th ACM/IEEE international symposium on Low power electronics and design*, ISLPED '09, (New York, NY, USA), pp. 377–382, ACM, 2009.

[94] Z. Hu, A. Buyuktosunoglu, V. Srinivasan, V. Zyuban, H. Jacobson, and P. Bose, "Microarchitectural techniques for power gating of execution units," in *Proceedings of the 2004 international symposium on Low power electronics and design*, ISLPED '04, (New York, NY, USA), pp. 32–37, ACM, 2004.

[95] J. Pouwelse, K. Langendoen, and H. Sips, "Dynamic voltage scaling on a low-power microprocessor," in *Proceedings of the 7th annual international conference on Mobile computing and networking*, MobiCom '01, (New York, NY, USA), pp. 251–259, ACM, 2001.

[96] T. Pering, T. Burd, and R. Brodersen, "The simulation and evaluation of dynamic voltage scaling algorithms," in *Proceedings of the 1998 international symposium on Low power electronics and design*, ISLPED '98, (New York, NY, USA), pp. 76–81, ACM, 1998.

[97] ARM, Holdings, "Arm1176jzf-s technical reference manual." URL=http://infocenter.arm.com/help/topic/com.arm.doc.ddi0301h/DDI0301H_arm1176jzfs_r0p7_trm.pdf, 2009.

[98] Samsung, "Samsung s3c6410 mobile pprocessor." URL=http://www.samsung.com/global/business/semiconductor/file/media/s3c6410_datasheet_200804-0.pdf, Apr 2008.

[99] ARM Holdings, "Arm1176jzf-s technical reference manual." URL=http://infocenter.arm.com/help/topic/com.arm.doc.ddi0301h/DDI0301H_arm1176jzfs_r0p7_trm.pdf, 2009.

[100] FriendlyARM, "mini6410 schematic v1016." URL=http://www.friendlyarm.net/downloads, 2010.

[101] A. Sloss, D. Symes, and C. Wright, *ARM System Developer's Guide: Designing and Optimizing System Software*. Morgan Kaufmann, 2004.

[102] J. L. Hennessy and D. A. Patterson, *Computer Architecture, Fourth Edition: A Quantitative Approach*. San Francisco, CA, USA: Morgan Kaufmann Publishers Inc., 2006.

[103] L. Gauthier and T. Ishihara, "Processor energy characterization for compiler-assisted software energy reduction," *JECE*, vol. 2012, pp. 8:8–8:8, Jan. 2012.

[104] S. Hao, D. Li, W. G. Halfond, and R. Govindan, "Estimating android applications' cpu energy usage via bytecode profiling," in *Green and Sustainable Software (GREENS), 2012 First International Workshop on*, pp. 1–7, IEEE, 2012.

[105] E. Grochowski and M. Annavaram, "Energy per instruction trends in intel microprocessors," *Technology@ Intel Magazine*, vol. 4, no. 3, pp. 1–8, 2006.

[106] Garnesh, "Apple's intrinsity acquisition: Winners and losers." URL=http://www.anandtech.com/show/3665/apples-intrinsity-acquisition-winners-and-losers, July 2012.

[107] K. Roberts-Hoffman and P. Hegde, "Arm cortex-a8 vs. intel atom: Architectural and benchmark comparisons," *Dallas: University of Texas at Dallas*, 2009.

[108] BeagleBone , "Beaglebone schematics." URL=http://beagleboard.org/static/beaglebone/latest/Docs/Hardware/BONE_SCH.pdf, 2012.

[109] ARM, "Keil embedded development tools." URL=http://infocenter.arm.com/help/index.jsp?topic=/com.arm.doc.kui0100a/armasm_cihgjhed.htm, 2007-2010.

[110] M. Guthaus, J. Ringenberg, D. Ernst, T. Austin, T. Mudge, and R. Brown, "Mibench: A free, commercially representative embedded benchmark suite," in *Workload Characterization, 2001. WWC-4. 2001 IEEE International Workshop on*, pp. 3–14, 2001.

[111] ARM Ltd., "The ARM Cortex-A9 Procesors," *ARM Ltd. White Paper*, 2009.

[112] ARM, "Cortex-a9 technical reference manual." URL=http://infocenter.arm.com/help/topic/com.arm.doc.ddi0388i/DDI0388I_cortex_a9_r4p1_trm.pdf, 2010.

[113] Texas instruments, "Omap4460 multimedia device silicon revision 1.x technical reference manual." URL=http://www.ti.com/lit/ug/swpu235aa/swpu235aa.pdf, 07 2013.

[114] Texas Instruments OMAP Family of Products, "Omap4460 multimedia device silicon revision 1.x technical reference manual." URL=http://www.ti.com/lit/ug/swpu235aa/swpu235aa.pdf, 2011.

[115] PandaBoard, "Pandaboard es rev b1 schematics." URL=http://pandaboard.org/sites/default/files/board_reference/pandaboard-es-b/panda-es-b-schematic.pdf, 2011.

[116] D. Ye, J. Ray, C. Harle, and D. Kaeli, "Performance characterization of spec cpu2006 integer benchmarks on x86-64 architecture," in *Workload Characterization, 2006 IEEE International Symposium on*, pp. 120–127, IEEE, 2006.

[117] J. L. Henning, "Performance counters and development of spec cpu2006," *SIGARCH Comput. Archit. News*, vol. 35, pp. 118–121, Mar. 2007.

[118] S. C. Woo, M. Ohara, E. Torrie, J. P. Singh, and A. Gupta, "The splash-2 programs: Characterization and methodological considerations," vol. 23, pp. 24–36, 1995.

[119] B. Wichmann, A. Canning, D. Clutterbuck, L. Winsborrow, N. Ward, and D. Marsh, "Industrial perspective on static analysis," *Software Engineering Journal*, vol. 10, pp. 69–75, Mar 1995.

[120] U. Lk:jat, S. Kerrison, A. Serrano, K. Georgiou, P. Lopez-Garcia, N. Grech, M. V. Hermenegildo, and K. Eder, "Energy consumption analysis of programs based on xmos isa-level models," vol. 8901 of *Lecture Noles in Computer Science*, pp. 72–90, Springer International Publishing, 2014.

[121] R. Jayaseelan, T. Mitra, and X. Li, "Estimating the worst-case energy consumption of embedded software," in *Real-Time and Embedded Technology and Applications Symposium, 2006. Proceedings of the 12th IEEE*, pp. 81–90, IEEE, 2006.