

# **An Instruction Level Energy Characterization of ARM Processors**

*Evangelos Vasilakis*

Computer Architecture and VLSI Systems (CARV) Laboratory,  
Institute of Computer Science (ICS), Foundation of Research and Technology  
Hellas (FORTH)

**Technical Report FORTH-ICS/TR-450, March  
2015**

Work performed as a Master Thesis at the Department of Computer Science,  
University of Crete, under the supervision of Prof. Manolis G.H Katevenis, with  
the financial support of FORTH-ICS under the GreenVM project



## Abstract

As mobile devices and data-centers expand to cover global needs for services and personal computing, power consumption of systems and devices has become the most prevalent concern for hardware designers and software developers. ARM processors already dominate the mobile world and are taking leaps into the server market due to their inherent energy efficiency. In this work we study the energy characteristics of modern ARM processors at the instruction level.

To characterize the energy consumption of ARM processors we measure the energy consumption of special purpose benchmarks. Our measurements are made using actual voltage/current sensors provided by the *Odroid-XU+E* development board which contains an ARM big.LITTLE processor consisting of two clusters of four Cortex-A7 and four Cortex-A15 cores.

Our characterization benchmarks are designed specifically to stress specific units of the datapath. With two different benchmarks for each instruction type, we study both the latency and the energy of instructions as well as the maximum throughput of the processor for that instruction.

Our findings for Cortex-A7 cores show that integer instructions cost from 50 to 80 pJ each, float/double instructions from 80 pJ to 350 pJ each, and more complex instructions like divisions cost from 150 pJ to 1200 pJ per instruction. Load and store instructions cost 150 pJ to 200 pJ each when hitting in the L1 cache whereas the cost increases up to 270 pJ when accessing the L2 cache. On the Cortex-A15, instructions cost three to five times more than on Cortex-A7 for the same clock frequency, even when the two cores show the same throughput for an instruction.

For benchmarks that fit mostly in the L1 cache, we observed that at a same clock frequency, their execution time is 20% to four times faster on Cortex-A15, while energy to completion is increased by 2 to 4 times, relative to Cortex-A7. When comparing Cortex-A7 at the lowest frequency of 500 MHz to Cortex-A15 at the highest frequency of 1.5 GHz, we see that the execution time is 4 to 10 times faster on Cortex-A15, while energy to completion is increased by 5 to 9 times relative to Cortex-A7.

Through these measurements, we developed a thorough characterization of the ARM instruction set with energy and latency metrics for every instruction type. We validated the correctness of our characterization by developing an instruction level energy model and testing it on a variety of real programs. Our evaluation shows average mispredictions of 8.5% for Cortex-A7 and 14% for Cortex-A15.

Furthermore, we utilize our characterization and energy model to quantify the energy characteristics of heterogeneous multiprocessing, like ARM *big.LITTLE*, and show how this can help optimal workload placement in such systems. We highlight the different factors that contribute to the energy expenditure of such systems and show how these differ from one processor to the other.



## Περίληψη

Καθώς οι φορητές συσκευές και τα data-centers επεκτείνονται για να καλύψουν τις παγκόσμιες ανάγκες για ατομικές υπολογιστικές υπηρεσίες, η ενεργειακή κατανάλωση τους έχει γίνει ένα από τα πιο σημαντικά θέματα για τους σχεδιαστές υπολογιστών και τους προγραμματιστές. Οι επεξεργαστές ARM ήδη κυριαρχούν στην αγορά φορητών συσκευών και επεκτείνονται και στην αγορά εξυπηρετητών λόγω της έμφυτης τους ενεργειακής αποδοτικότητας. Σε αυτή τη εργασία μελετάμε τα ενεργειακά χαρακτηριστικά μοντέρνων επεξεργαστών ARM σε επίπεδο εντολών.

Για να μελετήσουμε την ενεργειακή κατανάλωση των επεξεργαστών ARM μετράμε την ενεργειακή κατανάλωση προγραμμάτων ειδικού σκοπού. Οι μετρήσεις μας πραγματοποιούνται με χρήση πραγματικών αισθητήρων τάσης/ρεύματος που παρέχονται από την πλατφόρμα ανάπτυξης *Odroid-XU+E* η οποία περιέχει ένα επεξεργαστή ARM big,LITTLE ο οποίος αποτελείται από τέσσερις πυρήνες Cortex-A7 και τέσσερις Cortex-A15.

Τα προγράμματα που χρησιμοποιήσαμε για την μελέτη μας είναι σχεδιασμένα ειδικά για να πιέσουν συγκεκριμένες μονάδες του datapath των επεξεργαστών. Με δυο διαφορετικά προγράμματα για κάθε τύπο εντολής, μελετάμε την καθυστέρηση και την ενεργειακή κατανάλωση των εντολών καθώς και την μέγιστη απόδοση του επεξεργαστή για την κάθε μία.

Οι μετρήσεις μας για τους Cortex-A7 δείχνουν ότι integer εντολές κοστίζουν από 50 pJ έως 80 pJ η καθεμία, εντολές float/double κοστίζουν από 80 pJ έως 350 pJ η καθεμία, και πιο πολύπλοκες εντολές όπως διαιρέσεις κοστίζουν από 150 pJ έως 1200 pJ ανά εντολή. Οι εντολές load και store κοστίζουν 150 pJ έως 200 pJ η καθεμία όταν η πρόσβαση γίνεται στο πρώτο επίπεδο κρυφής μνήμης, ενώ το κόστος αυξάνεται μέχρι τα 270 pJ όταν η πρόσβαση γίνεται στο δεύτερο επίπεδο κρυφής μνήμης. Στον Cortex-A15 οι εντολές κοστίζουν τρεις με πέντε φορές περισσότερο από ότι στον Cortex-A7 για την ίδια συχνότητα ρολογιού, ακόμα και όταν οι δύο επεξεργαστές επιτυγχάνουν την ίδια απόδοση για μια εντολή.

Για προγράμματα που χωρούν κυρίως στο πρώτο επίπεδο κρυφής μνήμης, παρατηρούμε ότι στην ίδια συχνότητα ρολογιού ο χρόνος εκτέλεσής τους είναι 20% έως τέσσερις φορές γρηγορότερος στον Cortex-A15, ενώ η ενεργειακή κατανάλωση μέχρι την ολοκλήρωση αυξάνεται 2 με 4 φορές, σε σχέση με τον Cortex-A7. Όταν συγκρίνουμε τον Cortex-A7 στην χαμηλότερη συχνότητα των 500 MHz με τον Cortex-A15 στην υψηλότερη συχνότητα των 1500 MHz, βλέπουμε ότι ο χρόνος εκτέλεσης είναι 4 με 10 φορές γρηγορότερος στον Cortex-A15, ενώ η ενεργειακή κατανάλωση μέχρι την ολοκλήρωση αυξάνεται 5 με 9 φορές σε σχέση με τον Cortex-A7.

Μέσα από τις μετρήσεις μας αναπτύξαμε έναν πλήρη χαρακτηρισμό για το σετ εντολών ARM με μετρικές για την ενεργειακή κατανάλωση και την καθυστέρηση κάθε εντολής. Επαληθεύσαμε τον χαρακτηρισμό μας μέσω της ανάπτυξης ενός ενεργειακού μοντέλου επιπέδου εντολών και την δοκιμή του σε ένα πλήθος πραγματικών προγραμμάτων. Οι δοκιμές μας δείχνουν μέσο σφάλμα πρόβλεψης 8.5% για τους επεξεργαστές Cortex-A7 και 14.5% για τους Cortex-A15.

Επιπλέον, χρησιμοποιούμε τον χαρακτηρισμό και το ενεργειακό μας μοντέλο για να ποσοτικοποιήσουμε τα ενεργειακά χαρακτηριστικά συστημάτων ετερογενούς πολυεπεξεργασίας όπως το ARM big.LITTLE και δείχνουμε πώς αυτό μπορεί να βοηθήσει την αποτελεσματικότερη κατανομή επεξεργαστικών πόρων σε τέτοια συστήματα. Τονίζουμε τους διαφορετικούς παράγοντες που συνεισφέρουν στην ενεργειακή κατανάλωση τέτοιων συστημάτων και δείχνουμε πώς αυτό μπορεί να διαφέρει ανάμεσα στους δύο επεξεργαστές.



# Contents

<b>1</b>	<b>Introduction</b>	<b>1</b>
1.1	Contributions . . . . .	3
<b>2</b>	<b>Motivation - Related Work</b>	<b>4</b>
2.1	Related Work . . . . .	4
2.2	Motivation . . . . .	8
<b>3</b>	<b>Background</b>	<b>10</b>
3.1	Processor Energy Consumption . . . . .	10
3.2	Processor Performance and Energy Efficiency Metrics . . . . .	11
<b>4</b>	<b>Methodology</b>	<b>13</b>
4.1	Special Purpose Benchmark Design . . . . .	13
<b>5</b>	<b>Experimental Setup</b>	<b>15</b>
5.1	ARM big.LITTLE . . . . .	15
5.2	ARM Cortex-A7 and Cortex-A15 cores . . . . .	16
5.3	Power Sensors . . . . .	17
5.4	Migration Policies and Frequencies . . . . .	18
<b>6</b>	<b>ARM Instruction Set</b>	<b>19</b>
6.1	ARM Instruction Categories . . . . .	22
6.1.1	Branch Instructions . . . . .	22
6.1.2	Integer Arithmetic and Logic Instructions . . . . .	22
6.1.3	Floating Point Arithmetic Instructions . . . . .	23
6.1.4	Register Movement Instructions . . . . .	23
6.1.5	Compare and Test Instructions . . . . .	24
6.1.6	Load and Store Instructions . . . . .	25
6.2	Other Instructions and Assembler Mnemonics . . . . .	26
<b>7</b>	<b>Measurement Results - Energy per Instruction</b>	<b>27</b>
7.1	Results Summary . . . . .	28
7.2	Integer Arithmetic and Logic Instructions . . . . .	29
7.3	Float Arithmetic Instructions . . . . .	34

7.4	Double Arithmetic Instructions . . . . .	36
7.5	Integer Move Instructions . . . . .	38
7.6	Float Move Instructions . . . . .	40
7.7	Double Move Instructions . . . . .	42
7.8	Integer Compare and Test Instructions . . . . .	44
7.9	Float Compare Instructions . . . . .	46
7.10	Double Compare Instructions . . . . .	47
7.11	Integer Load and Store Instructions . . . . .	48
7.12	Float and Double Load and Store Instructions . . . . .	50
7.13	Energy per Cycle . . . . .	51
<b>8</b>	<b>ARM versus x86</b>	<b>53</b>
<b>9</b>	<b>Evaluation</b>	<b>55</b>
9.1	Evaluation Benchmarks . . . . .	55
9.2	Energy Model . . . . .	63
9.3	Evaluation Results . . . . .	68
9.3.1	Discussion . . . . .	74
<b>10</b>	<b>big.LITTLE Comparison</b>	<b>77</b>
<b>11</b>	<b>Conclusions and Future Work</b>	<b>82</b>
<b>A</b>	<b>Using Architectural Counters to Evaluate the Cost of Instructions in x86 Architectures</b>	<b>84</b>
A.1	Introduction . . . . .	84
A.2	Methodology . . . . .	84
A.2.1	Benchmark Specification . . . . .	85
A.2.2	Experimental Setup . . . . .	86
A.3	Benchmark Description and Results . . . . .	86
A.4	Conclusion . . . . .	98

# Chapter 1

## Introduction

Energy consumption has always been a major concern for computer architects. At first heat dissipation and therefore maximum power was the limiting factor. High temperatures and overall heat generation is the reason that high power chips require expensive packaging and massive heatsinks with active cooling. Even if heat dissipation and power demands can be satisfied, overall energy consumption has become one of the most important design problems of the last decade. Its importance is mainly attributed to two different reasons:

First, as data-centers and super-computers are becoming more and more prevalent, energy consumption contributes greatly to the total cost of ownership (TCO) and to their environmental impact both for running the actual servers and for cooling them. Studies suggest that servers consume approximately 1.5% of the global power consumption [1]. In addition, cooling and power costs may soon rise to more than the cost of the actual servers [2]. The role of processor energy consumption is important as it can be up to 30% of a server's power demand [3].

Second, The ubiquity of mobile devices such as smart phones laptops and tablets, has made battery life a big marketing point. Mobile devices have increased in numbers globally from 6.9 in 2013 to 7.4 billion in 2014 [4]. Furthermore, as more and more processing power is required from devices with limited battery capacity and no practical heat dissipation mechanisms, energy efficiency and peak power have become critical design concerns.

Energy consumption has been at the forefront of optimization efforts for the biggest part of the last decade. In the early 2000's performance scaling by increasing the frequency hit the power wall not only because of design constraints and heat dissipation issues, but also due to the sheer power demand of complex processors. Computer architect's efforts to push the limits of Instruction Level Parallelism demanded out-of-order processors with huge speculation mechanisms that were for the most part underutilized due to the serial nature of code and instruction dependencies. These factors have led to a plateau in single thread performance that cannot be overcome without huge losses in energy efficiency and chip area.

There are a number of techniques for reducing energy consumption in proces-

sors. The most significant was the advent of multi-core processors. Multi-cores have been a great success, not only because they have pushed the performance limits further by exploiting parallelism, but also because they can induce huge energy benefits. Multi cores offer better performance per Watt because they utilize smaller, more energy efficient cores that are inherently more energy efficient, and additionally, with more cores fine-grain power state control can be applied more easily.

Furthermore, there have been a plethora of techniques for energy optimizations at the circuit and architecture level, these include DVFS [5] loop caches [6, 7], and energy optimizing data encoding schemes [8, 9]. Even lower level techniques include clock and power gating, low threshold voltage transistor technology and more.

In this work we characterize the energy consumption of two modern ARM processors by designing special purpose benchmarks that stress specific data-path units every time. These measurements are done in a wide range of frequencies that reveal how energy consumption scales with frequency. From our measurements of these benchmarks we produce a static energy estimation model for each instruction type based on the Energy per Instruction (EPI) and latency that we observe. We then proceed with evaluating our model with real benchmarks completely separate from the ones used to develop the energy model. Our model has an average error of 8.5% for Cortex-A7 cores and 14% for Cortex-A15 cores. Our energy model can be used at compile time to optimize code for energy consumption and also for characterizing existing workload energy-wise given a run-time and an instruction breakdown. In addition to the above, we use some earlier work from the authors the energy cost of instructions at X86 processors to compare the ARM and X86 architectures energy wise at the instruction level.

The rest of this thesis is structured as follows:

In Chapter 2 we present some related work and the motivation for this work In Chapter 3 we offer some background on processor energy consumption and performance. In Chapter 4 and 5 we describe our methodology and experimental setup. In Chapter 6 we give an overview of the ARM instruction set. Chapter 7 presents the results of our characterization. Chapter 8 Compares the energy of X86 and ARM processors based on our characterization from chapter 7 and some earlier work on X86 instruction characterization. Chapter 9 presents our energy model and the evaluation results. Chapter 10 offers some insight on the use of our characterization and energy model for optimal compute resource allocation, and chapter 11 concludes this work with some remarks and future work plans. Our earlier work on X86 Intel processors can be found in appendix A

## 1.1 Contributions

The contributions of this work are:

- A thorough characterization of the ARM instruction set for the energy and performance characteristics of two different ARM processors at the instruction level using real energy measurements and characterization benchmarks designed specifically for that purpose.
- An accurate instruction level energy model based on the instruction level characterization.
- An evaluation of our energy model using a variety of real world benchmarks like whetstone, linpack, matmul, and fibonacci. Our model achieves average energy misestimations of 8.5% for Cortex-A7 and 14% for Cortex-A15 cores.
- A significant insight on the energy efficiency and performance trade-offs for heterogeneous multiprocessing architectures like ARM big.LITTLE.
- A quantitative comparison between ARM and Intel X86 architectures based on some earlier work on X86 architectures.

## Chapter 2

# Motivation - Related Work

### 2.1 Related Work

There has been great interest in the power efficiency of processors and systems in the last 20 years. There are many ways one could use to study the energy consumption of processors and trade-offs between performance and energy consumption. One way towards that is a series of simulators and energy models have been proposed to help limit the design-space and point out design inefficiencies early in the design process.

At the Cache and Memory level, the most well-known tools used are Dramsim2 and Cacti which can be used for timing and energy analysis of DRAMS and caches respectively [10, 11]. Orion2 is a model used to estimate the energy of Networks on Chip (NOCs) that are the de-facto interconnects used for chip multiprocessors [12]. Wattch is a processor energy simulation tool that can provide architects with pre-silicon energy estimation [13]. For a wider system approach McPat can be used to estimate the Energy, Delay and Area metrics for a design with quantitative properties and activity factors as inputs for the simulation [14]. Gem5 is a simulator that is widely used and although it does not support power estimation, it has proven very useful in research, including this work, where it was used to obtain the instruction traces for the evaluation benchmarks [15].

The goal of most, if not all, works regarding processor energy consumption is to correlate the energy of the processor with some other measurable quantity, since actual measurements are not always possible or desired. Once a correlation of energy with some other metric has been established, the results can be used for software or hardware design optimization.

The metric with which energy will be correlated can vary depending on approach, it can be either the runtime, the IPC/CPI metric, the instruction mix, runtime events like branch mis-predictions or cache misses, parallelism or active threads, and many more.

Depending on the metric used and its temporal availability, the emerging model

can be used either statically by the hardware designer or software/compiler developer to ensure a good performance/energy trade-off, and dynamically at runtime, for techniques like auto-tuning or power capping.

To develop each model, some measurement of energy consumption of the processor is needed to form a correlation with some other metric(s) that will be proxies for energy consumption. The way of measuring the energy varies in the literature, from direct processor measurements to system wide measurements to architectural simulation with tools like *wattch* and gate level or register transfer level hardware simulations.

Bellow we present the most prevalent works from the literature covering all options for measurement method, correlating metric and energy model types (static or dynamic).

In [16] and [17] the authors have introduced an method including empirical measurements and linear regression for deriving a detailed instruction level energy model for a 3-stage pipelined ARM7TDMI 32 bit processor. Their findings include differences in energy consumption among instructions based on opcodes, fetch address, register number and hamming distances between instructions, instruction addresses, and their operands. For their experiments the use a fine grain energy sensor and statistical analysis. Their method requires a cycle accurate power measuring technique which is not feasible for todays clock speeds. Additionally, they have not included the effects of stalls in the data-path due to inter-instruction dependencies and the same is true for load and store instructions, however they do mention these limitations in their conclusions as future work. A similar work to the above is [18] where the authors use gate level netlist with annotated capacitance information to evaluate their energy model for the ARM7TDMI processor through gate level simulation.

In [19] an instruction-level power model for a single core, in-order RISC processor architecture is presented. The authors do not analyze each instruction individually, but study the average power and running time instead. Their results find that the power in a processor is nearly constant, no matter what instructions are executed, but the I/O port power is related to the behavior of the program. This however does not apply to the processors in our study and the reason is that in [19] they implemented an OpenRISC processor in which nearly all of the instructions need 5 pipeline stages and no matter what the instruction is, the ALU always performs all the different operations and only chooses a specific one as an output.

Another work where the authors do not consider each arithmetic and logic instruction individually is [20]. Their analysis is done on an ARM1176JZF-S processor. The authors show that the power is related to both the distribution of instruction types and the operations per clock cycle (OPC) of the program. They also prove that energy per operation (EPO) decreases with increasing operations per clock cycle, which is a point we also confirm through this work. Their results distinguish the energy per instruction for load, store and arithmetic and logic instructions, they also distinguish between the operands types of instructions like in our work, but find no significant differences. Their model can have errors in power

prediction ranging from -7% to +8%.

In [21], Wang and Ranganathan develop an instruction-level prediction mechanism to estimate the energy consumption of a given program under different numbers of cores in a GPU. They build a mathematical model of energy consumption where the independent variable is the number of active stream multiprocessors in the GPU which takes the profile of PTX instructions as input. With the predicted energy-optimal number of active cores from their model, there can be energy saving from 7.31% to 11.76% on average, with a worst case of performance lost 4.92% for the benchmarks they studied.

Another paper for GPU energy characterization is [22]. In that work, the authors have created a execution time prediction model and an energy consumption prediction model that take instruction-level and thread-level parallelism into consideration. The energy model works with the time prediction from the time prediction model and complemented with empirical data for each specific GPU device.

in [23] the authors have characterized the energy consumption of data transfers and arithmetic operations in X86-64 micro-architectures from Intel and AMD. Their approach is to isolate the energy consumption characteristics of certain basic operations like data transfers and arithmetic operations and they also investigate how expensive data transfers from different cache levels or main memory are.

Another energy model for x86-64 is [24] where the authors construct a model by identifying how energy per instruction scales with the number of cores, the number of active threads per core, and instruction types in the Intel Xeon Phi processor. They use a set of specialized micro-benchmarks exercising different categories of instructions with varying memory behavior, number of active cores, and number of active threads per core to characterize the Energy per Instruction of the core. The energy model utilized the energy per instruction results along with performance counter statistics and achieves an accuracy between 1% and 5% for real world benchmarks.

[25] presents a methodology to solve the problem of run-time power optimization by designing a processor based on the SimpleScalar/PISA instruction set architecture that estimates its own power/energy consumption. Estimation is performed by the addition of small counters that tally events which consume power. This methodology results in an average power error of 2% and energy estimation error of 1.5%.

In [26] Jordan et al. propose a rapid method to estimate the energy consumption of candidate architectures for VLIW ASIP processors. The proposed method avoids the time-consuming simulation of the candidate prototypes, without any loss of accuracy in the predicted energy consumption. In this work they find that we can accurately predict the energy consumption of proposed architectures while avoiding simulation of the complete system.

In [27], an instruction-level energy model is proposed for the data-path of very long instruction word (VLIW) pipelined processor that can be used to provide accurate power consumption information during either an instruction-level simulation

or power-oriented scheduling at compile time. The analytical model takes into account several software-level parameters (such as instruction ordering, pipeline stall probability, and instruction cache miss probability) as well as micro-architecture level ones (such as pipeline stage power consumption per instruction) providing an efficient pipeline-aware instruction-level power estimation whose accuracy is very close to those given by register transfer or gate-level simulations. They have demonstrated an average error in accuracy of 4.8% of the instruction-level estimation engine with respect to the gate-level engine simulation.

Another power estimation model based on hardware performance counters is presented in [28]. The authors use linear regression and rely on already available, high-level benchmarks for training instead of self-written or hand-tuned micro-kernels. That way, they develop an energy consumption model for the IBM POWER7 processor with errors less than 5% across various multi-threading usage scenarios.

In [29] Martonosi and Cotreras have used performance counters to model the energy consumption of an Intel Xscale processor with quite good accuracy, the average estimated power consumption is within 4% of the measured average CPU power consumption.

Martonosi and Joseph also examine the use of hardware performance counters as proxies for power meters in [30]. They simulate an Alpha 21264 core with *Wattch* and also use real power measurements with an Intel Pentium Pro based system to develop and evaluate their model.

Isi and Martonosi employ a similar methodology with performance counters in [31], this time they provide power breakdowns for 22 of the major subunits of the Intel Pentium 4 processor over minutes of SPEC2000 and desktop workload execution.

In [32] the authors categorize the AMD Phenom performance counters into four buckets: FP Units, Memory, Stalls, and Instructions Retired and develop microbenchmarks specifically to stress those four counters and explore the space of their cross product. They achieve median errors in their modeling of less than 8%. Just like in our study, their evaluation benchmarks are separate from the ones they used to derive the energy model.

In [33] the authors use and extend the Embedded StrongARM Energy Simulator (EMSIM) to dynamically calculate the source-code level program energy consumption between breakpoints set by GDB. For the processor energy model they use the StrongArm SA-1100 instruction-level energy model.

In [34] the authors present a general methodology to implement a processor energy model based on instruction-level characterization of a SPARC-based Leon3 processor. The model is developed by simulating back-annotated gate-level netlist and has two levels of accuracy: a coarse-grain estimation based on characterizing each single instruction and a fine-grain estimation accounting for the impact of instructions interdependency on energy and based on characterizing pairs of instructions together.

Natarajan et al in [35] have calculated the effect of mis-speculation and over-provisioning in an Alpha 21264 core simulated by *Wattch*. They have found that

flushed instructions account for approximately 6% of total energy, while over-provisioning imposes a tax of 17% on average.

Blem et al analyze measurements on the ARM Cortex-A8 and Cortex-A9 and Intel Atom and Sandybridge i7 microprocessors over workloads spanning mobile, desktop, and server computing to investigate the role of ISA in modern microprocessors performance and energy efficiency in[36]. They find that ARM and x86 processors are simply engineering design points optimized for different levels of performance, and there is nothing fundamentally more energy efficient in one ISA class or the other.

## 2.2 Motivation

From the related work described above we can see a plethora of energy models, characterization and measurement methods and use scenarios for the findings of each work. Although our methodology is similar to some of the works above and it combines some techniques seen in these, it differs from these in a few fundamental ways:

- **Measurement granularity:** Our method does not require too fine grain measurements like in [16, 17, 18] or gate level simulation like in [19] in order to get the energy per cycle or energy per instruction.
- **Correlating metric:** Our energy model is static, this means that the correlating metrics are not extracted at runtime like cache misses or branch mis-predictions like in [29, 32, 33]. Our model uses instruction breakdown and run cycles that can be known at compile time based on the latency of each instruction that we also extrapolate through our measurements. This makes our model best suited for compile time optimizations.
- **Width and depth of characterization:** In our approach we characterize most of the ARM instruction set, showing the (not so) subtle differences between instructions. Additionally, we characterize each instruction not only for energy but also for latency, this can tell us not only the cost of each instruction but also the cost of each instruction relative to its latency which is one of the most important factors of our energy model. Furthermore, by examining two different processors with the same instruction set at a wide range of frequencies gives us a better understanding of how energy consumption scales with frequency, as well as, what is the relation between different implementations of the same instruction set.
- **Target architecture:** Our work focuses on the latest generation of ARM processors which are significantly more complex than those studied in [16, 17, 18, 19, 20, 25].
- **Usability:** Our model can be exploited for optimizations at compile time and also for the energy estimation of existing workloads based on run cycles and instruction breakdown. These metrics are available both at compile time

and via runtime profiling and do not require any in-depth understanding of the processor at hand.

For the reasons stated above we believe our work augments and complements the current effort to understand the energy consumption of processors in a novel and complete way.

Additionally, this work sheds some light on the idea of heterogeneous multi-processing [37] as an energy saving technique, as well as the relation of energy consumption with operating frequency for two different processor types and a wide range of frequencies.

## Chapter 3

# Background

Understanding the energy consumption of processors has been an interesting research subject for a long time. The same is true for research on architectural and circuit-level optimizations to reduce energy consumption and improve energy efficiency. In this chapter we give a brief introduction on processor energy consumption and introduce a number of metrics that have been used for quantifying energy consumption and performance.

### 3.1 Processor Energy Consumption

Processors are complex circuits made of transistors. A transistor can have two states, either ON or OFF, meaning that current can and cannot run through it respectively. Logic gates with one or more inputs and outputs are composed by connecting transistors together, the way one transistor's output is connected to the input of one or more other transistors signifies the logic function that logic gate implements. Each transistor is characterized by its inherent capacitance ( $C$ ), when the transistor changes state, that capacitance is either charged or discharged according to the state transition. There are two ways in which transistors consume energy.

The first is static energy that is consumed regardless of activity due to leakage currents. Static energy depends on the supply voltage and the characteristics of the transistors such as gate length and the materials that it is composed of. This means that static energy consumption is mostly an inherent characteristic of the circuit.

The second type of transistor energy consumption is dynamic energy. Dynamic energy is consumed when a transistor changes state meaning that the transistor capacitance is either charged or discharged. Dynamic energy is also dependent on the transistor characteristics that determine the capacitance but also on the state change frequency and the voltage. The equation that describes the energy expenditure  $E$  of a transistor operating on voltage  $V$  at frequency  $f$  and has capacitance equal to  $C$  with activity  $a$  is:

$$E = \frac{1}{2} \times C \times V^2 \times f \times a$$

From this equation we see that dynamic energy is linear to capacitance and frequency while quadratic in respect to voltage. The activity factor describes the percentage of state changes for a transistor, this is due to the fact that when the state does not change between cycles there is no dynamic energy consumption. Furthermore, at higher frequencies the voltage has to be higher to ensure correct operation, that is because in order for the transistor to achieve one of its two stable states, the capacitance has to have enough time to charge or discharge, because of this, if the frequency is increased so must the voltage.

The equation above is used to describe the dynamic energy in circuits with more than one transistor such as processors. However, the capacitance of an entire chip is not always known and the activity factor cannot be determined easily without painfully detailed simulations at the transistor level for every possible state of the circuit. Due to these limitations, it is not easy to utilize this equation directly with processors. Nonetheless, its usefulness mainly consists of the relation -linear or quadratic- that energy consumption has with frequency, voltage, and activity factor.

## 3.2 Processor Performance and Energy Efficiency Metrics

There are a number of metrics to characterize the performance and energy efficiency of processors. The most trivial energy/power metric is Thermal Design Power (*TDP*). *TDP* is a characteristic of the processor chip provided by the manufacturer for heat dissipation issues, it is a quantity measured in Watts and, to put it simply, it is the maximum power and heat production for which there must be adequate cooling. This characteristic of processors is pretty useless for any other than its intended purpose because the processor chip will not always reach that level of power drain and most of the time it will work well below that limit even at full utilization.

To study the performance of a processor a series of metrics has been proposed. The simplest are metrics that quantify the raw computing power of a chip, these are metrics like *Gflops* or *Gops* and can give an upper limit to the computing capacity of a processor. For more meaningful measurements of a processor performance and comparative measurements with others, the rates of a range of standardized benchmarks are used, these benchmarks utilize common microkernels from mathematic and scientific problems and give a rating of the processor depending on the time it took to complete them [38, 39, 40].

The most interesting category of metrics is that of correlating metrics. These correlate performance and energy consumption in order to give a more holistic view of the efficiency of a processor. Some examples of such metrics are Energy

Delay Product ( $EDP$ ) and some of its derivatives like  $ED^2P$ .  $EDP$  is simply the product of the energy consumed and the time it took to perform a computation, this metric is valuable in that it evens out differences between architectures with vastly different performance and energy characteristics and when no common point of comparison can be found.

$ED^2P$  is the same as  $EDP$  with emphasis on the time it took to run a computation. One can choose between variations of such metrics based on the purpose of the processor under study. Some simpler and more general correlating metrics, utilized mostly in large scale systems like data centers and supercomputers, are  $Gflops/Watt$  or  $Jobs/Watt$  but these metrics usually concern the power consumption of an entire system and not only the processors, nonetheless, they are worth mentioning for a wider insight.

In these last few paragraphs the main performance, energy and energy efficiency metrics were summarized, the range of these metrics and different use-scenarios were highlighted to showcase the fact that energy efficiency is mostly a relative concept that depends on many different factors and metrics.

## Chapter 4

# Methodology

Processors consume energy when executing instructions but also when stalling, whether stalling is caused by dependencies between instructions or while waiting for the memory system to respond to a load instruction. Determining the exact contribution of each component of a processor to the total energy consumption is not feasible for two reasons.

First, the low level architecture of most processors, including those in our study, is not disclosed apart from a wide architecture view, so it is not possible to know the exact constitution of a processor and thus a circuit level approach is not feasible.

Second, even if the low level architecture of the processor is known, the sheer complexity of the system is prohibiting. To have a low level component based energy model of a processor would require, apart from full knowledge of the architecture, painstakingly long simulations with circuit analysis tools like Spice [41].

Additionally, these simulations would have to account for all possible interactions between instructions, instruction orderings, and their arguments. Such a methodology would provide a very detailed model, however, such a model would be mostly unusable to a compiler developer or a programmer trying to achieve energy efficiency at the instruction lever due to its size and complexity.

For the reasons stated above we approach the problem from a higher level. We view the processor as a closed system and try to determine its energy characteristics based on observations and measurements of benchmarks design specifically to target specific instructions or instruction types. This way, we build an energy model that is usable for energy optimizations without trading off too much accuracy.

### 4.1 Special Purpose Benchmark Design

In order to isolate the net effect of each instruction type on the energy consumption of the processor, we have to isolate each instruction. To do that, we have designed a benchmark for each instruction, we run our benchmarks for a sustained amount of time to achieve statistically important measurements and mitigate the effects of

initializations and operating system overheads.

Each benchmark is a highly unrolled loop that executes one thousand instructions of the same type in every iteration. This level of unrolling is done to diminish the effect of branches. The instructions are written directly in assembly to avoid any compiler optimizations and the dependencies between them are controlled.

Due to the fact that all our processors can issue more than one instruction per cycle (two for Cortex-A7 and three for Cortex-A15) and also to study the energy and performance of instructions with different datapath utilization, we have designed two kinds of benchmarks for each instruction.

The first benchmark type executes instructions that have no dependencies between them, this means that every instruction has no Read-After-Write (RAW) dependencies with the previous instruction in program order, this allows us to see the maximum throughput of the processor for every instruction type. An example can be seen in listing 4.1.

The second type of benchmark intentionally introduces RAW hazards between every instruction and the one before it in program order. This allows us to study the latency of each instruction, that is, how many cycles a single instruction must occupy a functional unit of the data-path until it has completed and its result can be used by a subsequent instruction.

This type of measurement is very important to our energy model development as it can show how much of the total instruction energy (EPI) can be attributed to each instruction and what part of that energy is part of the datapath cost regardless of the instruction being executed. This type of benchmark is not possible for every instruction as not all instructions have a result that can be used by a subsequent instruction of the same type. For example *mov* instructions with immediate operand as described in 6.1.4. An example of code with RAW dependencies is shown in listing 4.2.

```
1 add    r1, r2, r3
   add    r4, r5, r6
3 add    r7, r8, r9
```

**Listing 4.1:** Instructions with no RAW hazards

```
1 add    r1, r2, r3
   add    r4, r1, r6
3 add    r5, r4, r7
```

**Listing 4.2:** Instructions with RAW hazards

For our study we have created more than 100 benchmarks to target all the configurations described above.

## Chapter 5

# Experimental Setup

Our work on ARM processors was carried out on a odroid XU+E board [42]. This board includes a Samsung Exynos 5410 (or Exynos 5 octa) System on Chip (SoC), a PowerVR SGX544MP3 GPU and 2 GB of LPDDR3 for main memory attached as a Package on Package (PoP) through a DFI protocol along with various peripherals [43, 44, 45].

The SoC includes 8 ARM processor cores, four Cortex-A7 and four Cortex-A15 on a heterogeneous multiprocessing configuration [46, 47]. Although there are 8 cores in total, only 4 can operate at a time and there is a way to quickly migrate running processes from one cluster to another mostly for power saving. This configuration is known as single ISA heterogeneous processing in the literature and developed by ARM under the name *big.LITTLE* [37, 48]. The processor is manufactured at a 28 nm HKMG process by Samsung [49].

### 5.1 ARM big.LITTLE

*ARM big.LITTLE* is the commercial name for ARM's heterogeneous multiprocessing architecture based on the idea of single ISA heterogeneous multiprocessing for energy efficiency first proposed in [37]. The main idea behind this is that a system can have two different core types with the same instruction set and the capacity to transfer state and migrate processes at runtime from one core type to the other.

The different core types are optimized for energy efficiency and performance respectively, when the cores are underutilized the low power cores work while the high performance ones are powered down. When there is need for more performance, the high performance cores are powered on and the running tasks are migrated there until there is no longer need for high performance, i.e the cores are underutilized, so the running processes are once again handed over to the low power cores.



**Figure 5.1:** Odroid XU+E Board

## 5.2 ARM Cortex-A7 and Cortex-A15 cores

In ARM's big.LITTLE implementation of single ISA heterogeneous multiprocessing, the big (high performance) are Cortex-A15 cores while the little (low power) cores are Cortex-A7.

Cortex-A7 cores are dual issue in-order cores with 32 KB of L1 Data, 32 KB of L1 Instruction Cache and 512 KB of shared L2 cache. Their pipeline is 8 to 10 stages long and consists of one load/store, one multiply, one floating point, and two integer units as seen in 5.2.

Cortex-A15 cores are triple issue out-of-order cores with a data-path length between 15 and 24 stages. The L1 caches remain the same in size with the Cortex A7 at 32 KB for both data and instructions, however the also shared L2 cache is significantly bigger at 2 MB. The Cortex-A15 datapath has a separate load and store unit, two integer units, two floating point units as well as a separate branch unit and a single multiply unit as seen in 5.3.

Both core types have identical architecture register sets, these are 16 32-bit general purpose registers and 16 64-bit floating point registers that can also be seen as 32 32-bit floating point registers.

In terms of frequency scaling, the *little* Cortex-A7 cores can operate from 500 MHz to 1.2 GHz and the *big* Cortex-A15 cores from 800 MHz to 1.5 GHz. The architectural differences, different datapath lengths and frequency ranges reveal that the little cores are designed to trade off performance for energy efficiency while the big cores for the opposite.

The two different clusters of big and little cores are connected with an ARM CCI-400 cache coherent interconnect [50], this makes migration of processes from one cluster to the other easy and fast, ARM reports a migration time of a few microseconds. Although it is possible to utilize some of the little cores and some of the big cores or all of them at once as stated in [51] under the term *Global Task Scheduling*, for this study we have not considered this scenario as our interests lie

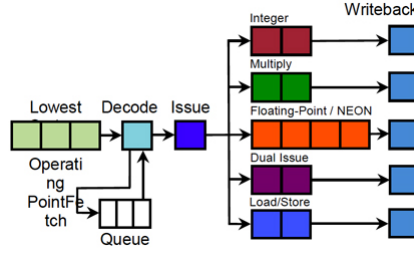


Figure 5.2: ARM Cortex-A7 Pipeline

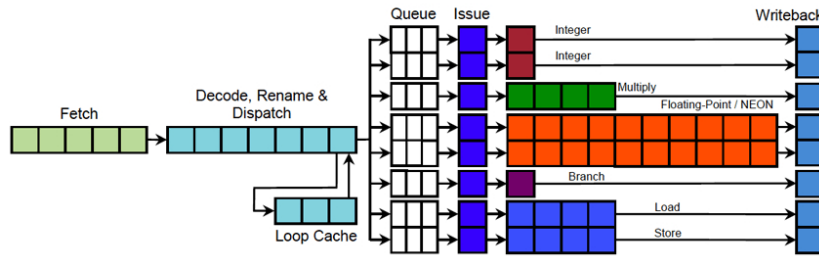


Figure 5.3: ARM Cortex-A15 Pipeline

in the energy characteristics of cores and not the entire system.

Both Cortex-A7 and Cortex-A15 cores implement the ARMv7a Instruction set and have floating point hardware units as well as hardware division but do not implement the SIMD extensions [52, 53].

### 5.3 Power Sensors

The odroid XU+E board that we used is equipped with four different power/current sensors. The sensors are ina231 chips and one is used for each cluster of *big* and *little* cores, one is used for the on package DRAM and one for the GPU. These sensors are connected to the processor via an I2C bus and provide registers for configuration and reading their values.

The ina231 sensors measure two types of voltage, bus and shunt voltage in an interleaved manner. The bus voltage indicates the voltage of the load and the shunt shows the current drawn when converted accordingly either in the sensor chip itself or through the driver in the OS kernel. It is also possible for the ina231 chips to calculate the voltage-current product in the chip but the default driver chooses to do so in kernel space to avoid I2C transactions.

The ina231 uses eight 16-bit registers, all accessible through i2c interface. The measured bus-voltage and shunt-voltage reside inside the two read-only ina231 registers. The calculated current and power similarly reside inside the other two read-only registers. The rest of the 4 registers are read/write and they are used for configuration purposes.

The configuration registers can be used to set the conversion intervals for converting bus and shunt voltages into voltage and current and to set the number of samples that will be averaged and stored into the read-only registers. The available conversion times for either bus-voltage or shunt voltage are: 140us, 204us, 332us, 588us, 1100us, 2116us, 4156us, and 8244us. The available configuration number of samples are: 1, 4, 16, 64, 128, 256, 512, and 1024 samples.

The ina231 chip follows the following routine to update its registers:

1. Makes a shunt voltage measurement.
2. Makes a bus voltage measurement.
3. Updates its measurement registers by the average of the bus-voltage when the set number of samples has been taken.

For example, if bus-voltage conversion time is set to 1100us, shunt-voltage conversion time is set to 140us and number of averaging samples is set to 4, each pair of shunt-bus voltage value lasts for 1240us and each register update lasts for  $4 \times 1240\text{us} = 4960\text{us}$  or 4.96 ms. Thus, each measurement is updated every 4.96 ms.

The measurement registers are read from the driver and mapped onto a set of files available for reading in user-space. For our study we have chosen, after careful examination and experimentation, 140us for the bus and shunt voltage conversion times and to average every 2048 measurements. That configuration gives our measurements a granularity of 280ms that we have found to be detailed enough without imposing big overheads for the system software.

## 5.4 Migration Policies and Frequencies

There are a number of settings regarding big.LITTLE migration, active core clusters and frequencies available in our platform. The migration policy determines when and how computing can migrate from the Cortex A7 to the Cortex A15 cores and reversely, this can be set to either *disabled*, *only A7*, *only A15* and *either A7 or A15* which is the dynamic setting that allows migration from one cluster to the other depending on core utilization.

Frequency can be set for both cores clusters within a preset range from 500 MHz to 1.2 GHz for the Cortex-A7 *little* cores and from 800 MHz to 1.5 GHz for the Cortex-A15 *big* cores, the step for both core types being 100 MHz leaving a total of eight different frequency settings for the *little* cores and also eight for the *big* cores.

All the above settings are available at user-space through read/write system files that can also be read to get the current values. Available in the system are also temperature sensors for each core, these can be used to get the current temperature of each core separately. We use these sensors to make sure that the core temperatures in our experiments behave in a similar way. That way we do not account for the effect of temperature in energy consumption as shown in [54].

## Chapter 6

# ARM Instruction Set

In this section we introduce the ARM instruction set and its general philosophy, we then present a categorization of the instructions based on functionality as well as operands and type.

The processors we used for this study implement the ARMv7 instruction set [52]. ARMv7 contains two instruction sets, the ARM and THUMB instruction sets. These are almost identical in functionality and differ in the way the instructions are encoded.

In the THUMB instruction set, the instructions can be encoded in either 16 or 32 bits. This has the advantage of smaller executable size for systems with little available memory, but also presents some drawbacks. The most important drawback is that most 16 bit instructions can only access 8 of the 16 general purpose registers available in the processor. Additionally, some operations require more than one 16-bit instruction and so are more efficiently implemented with a single 32-bit instruction.

These limitations of the THUMB instruction set have shifted our focus away from it. So we study only the ARM instruction set which offers full functionality and uniformity, we achieved this by setting the right compiler flags in our experiments.

The ARM instruction set is a RISC type instruction set, all instructions are encoded in 32 bits and can perform operations between registers, as well as load and store operation between registers and memory.

However, operations between memory and registers like the x86 CISC instruction set [55] are not supported. There are some major features that characterize the ARM instruction set and differentiate it from the standard RISC instruction set as described in the literature [56, 57].

The most prevalent difference is the conditional execution of instructions. Most ARM and THUMB instructions can be conditionally executed by setting a flag in their assembly mnemonic. The instruction will be fetched normally and go through some stages of the pipeline but the operation will not be performed, or more accurately, the instruction will not have any effects on the programmers

cond.	Mnemonic extension	Meaning (integer)	Meaning (floating-point)	Condition flags
0	EQ	Equal	Equal	Z==1
1	NE	Not equal	Not equal, or unordered	Z==0
10	CS	Carry set	Greater than, equal, or unordered	C==1
11	CC	Carry clear	Less than	C==0
100	MI	Minus	negative	Less than
101	PL	Plus, positive or zero	Greater than, equal, or unordered	N==0
110	VS	Overflow	Unordered	V==1
111	VC	No overflow	Not unordered	V==0
1000	HI	Unsigned higher	Greater than or unordered	C==1 and Z==0
1001	LS	Unsigned lower or same	Less than or equal	C==0 or Z==1
1010	GE	Signed greater than or equal	Greater than, or equal	N==V
1011	LT	Signed less than	Less than, or unordered	N != V
1100	GT	Signed greater than	Greater than	Z==0 and N==V
1101	LE	Signed less than or equal	Less than, equal, or unordered	Z==1 or N != V
1110	None ( AL )	Always (un- conditional)	Always (unconditional)	Any

**Table 6.1:** Conditional execution codes

model, memory or co-processor. Effectively, if a conditional flag is set and the corresponding bit in the Current Program Status Register (*CPSR*) is not, it is as if the instruction does not exist. The flags can be set by an instruction if the *s* flag is set in the instruction mnemonic.

The purpose of this feature is to reduce the number of branches for small *if-else* clauses, and thus speed up computation and improve code density. However, there are significant arguments against the support for conditional instructions. Each instruction has to provide 4 bits of its encoding for the condition flags, which is a valuable resource in a tight 32 or 16 bit encoding for ARM and THUMB instruction

---

Mnemonic	Description
LSL	Logical shift left
LSR	Logical shift right
ASR	Arithmetic shift right
ROR	Rotate Right
RRX	Rotate right one with extend

**Table 6.2:** Shift operations

sets respectively.

Arguments have been made that the extra four bits made available by eliminating conditional execution can be used to double the number of general purpose registers accessible by instructions from 16 to 32 in [58], and a hybrid solution for the same goal is proposed in [59]. Furthermore, ARM has removed the support for conditional execution in its 64 bit ARMv8 instruction set [60]. The reasons for this is the substantial performance of branch predictors which has made the implementation cost of conditional execution disproportionate to its benefits. The suffixes for conditional execution are seen in table 6.1.

Another feature of the ARM architecture and instruction set is the presence of a barrel shifter before the ALU. This barrel shifter allows one argument of an instruction to be optionally shifted by a constant value or by a value provided by a register. This feature makes some operations like multiplications with powers of 2 easier and faster. There are four possible ways a value can be shifted, these are shown in table 6.2 along with a short description.

ARM instructions can have a range of different arguments, these arguments can be either one, two, three or even a set of registers and an immediate value. The way immediate values are encoded is not straightforward in the ARM instruction set. There are a total of 12 bits available for encoding immediate values in 32-bit ARM instructions, which have a limited range of  $2^{12} = 4096$  values.

The way this limited range is used to describe immediate values is that the 8 bits are used to describe an 8 bit integer and the remaining 4 bits describe a rotation of that value into a 32 bit word. The 16 different ways to rotate the 8 bit value allow a left rotation of any even number from 0 to 30. This means that an immediate can be any 8 bit value starting at any even bit within a 32 bit word. This way of dealing with immediate values has some limitations, but given the limited 12 bits space for immediate values in the instruction encoding, this method can encode a wide range of numbers. When the number that must be used cannot be encoded with that method, the value will have to be either loaded from memory, or constructed using arithmetic instructions.

Mnemonic	Arguments	Description
b	immediate	Branch to target address
bl, blx	immediate	Function call with or without changing the instruction set
blx	register	Function call and change instruction set
bx	register	Branch to target address

**Table 6.3:** Branch instructions

## 6.1 ARM Instruction Categories

The main instruction categories are presented bellow along with a short description of their functionality.

### 6.1.1 Branch Instructions

There are various branch instructions which can be executed conditionally based on a flag set by previous instruction execution,. There are dedicated branch-and-link instructions to call functions and branch instructions can also be used to change to, and from, ARM and THUMB instruction sets. Branch instructions can branch to an address specified either by a register or by an offset from the current program counter value given by an immediate value. The different branch instructions along with their argument types, a short description, and their assembler mnemonic are presented in table 6.3.

### 6.1.2 Integer Arithmetic and Logic Instructions

The ARM instruction set offers a range of arithmetic and logic instructions for integers. The most common arithmetic operations are *addition*, *subtraction*, *multiplication*, and *division* and the most common logic instructions are logic *and*, *or*, and *xor*.

These instructions have three operands which can be either three registers or two registers and an immediate value, with the exception of division and multiplication which can only have three registers as operands.

The arithmetic instructions perform an arithmetic operation on two operands and store the result in the third register operand. When all three operands of an arithmetic instruction are registers one of the two source registers can be shifted by an immediate value or by a value given by a fourth register, this option however is not available for multiplication and division instructions. A summary of these instructions can be seen in table 6.4.

Mnemonic	Arguments	Description
add	Registers or immediate	Addition
sub	Registers or immediate	Subtraction
rsb	Registers or immediate	Reverse subtraction
mul	Registers	Multiplication
div	Registers	Division
and	Registers or immediate	Logical AND
orr	Registers or immediate	Logical OR
eor	Registers or immediate	Logical XOR

**Table 6.4:** Integer arithmetic and logic instructions

### 6.1.3 Floating Point Arithmetic Instructions

Similar to the corresponding integer instructions, there are arithmetic instructions for floating point values with 32 or 64 bits precision. These instructions do not accept immediate arguments and operate on the floating point register banks of the processor. In our case-study there are 32 32-bit floating point registers named *s0* to *s31* which can be aliased to 16 double precision 64-bit registers named *d0* to *d15*.

These instructions perform an arithmetic operation on the values of two operands and store the result to the third register operand. The most common instructions of this type are, addition, subtraction, multiplication and division. A summary of these instructions can be seen in table 6.5.

Mnemonic	Arguments	Description
fadd	Registers	Addition
fsub	Registers	Subtraction
fmul	Registers	Multiplication
fdiv	Registers	Division

**Table 6.5:** Floating point arithmetic and logic instructions

### 6.1.4 Register Movement Instructions

There are instructions that can move data between registers and likewise move an immediate value to a register for both integer and floating point registers. These instructions have two operands, they operate by copying the value of the second operand to the first, the first operand must always be a register and the second can be either a register or an immediate value encoded in the instruction.

The data from the second operand can be copied to the first operand either verbatim or bitwise inverted, depending on the instruction. A summary of these instructions for both integer and floating point operands can be seen in table 6.6

Mnemonic	Arguments	Description
mov	Registers or immediate	Move
mvn	Registers or immediate	Move bitwise inverse
fcpy	Registers	Move
fneg	Registers	Move bitwise inverse

**Table 6.6:** Move instructions

### 6.1.5 Compare and Test Instructions

There are four main comparison instructions, two compare and two test instructions. These instructions take two operands, one of which is always a register and the other can either be an immediate value or a register. The second operand can be optionally be shifted by either a constant value encoded in the instruction or a value provided by a register.

The compare and test instructions are very similar to the arithmetic and logical instructions in that they too perform an arithmetic or logic operation on the operands. They differ only in the fact that they always update the status flags in the *CPSR* register and in that the result of the operation is discarded afterwards. The main integer compare and test instructions are seen in table 6.7

In addition to compare and test instructions for integers which are widely used for loops, there are also compare instructions for floating point values. These instructions can only compare two floating point registers which can be either 32 or 64 bits wide or a floating point register with zero. Contrary to their integer counterparts, the second operand cannot be an immediate value and cannot be shifted. These instructions are *fcmpes* and *fcmped* for comparing two registers of 32 and 64 bits respectively and *fcmpzs* and *fcmpzd* for comparing a 32 or 64 bit floating point register with zero.

Mnemonic	Arguments	Description
cmp	Registers or immediate	Subtracts the two operands and updates the flags
cmn	Registers or immediate	Adds the two operands and updates the flags
teq	Registers or immediate	Logical AND between the operands and updates the flags
tst	Registers or immediate	Logical OR between the operands and updates the flags

**Table 6.7:** Compare and test instructions

Mnemonic	Type	Description
ldr	Load	Loads a 32 bit word into a general purpose register
str	Store	Stores a 32 bit word from a general purpose register
ldmfd	Load multiple	Loads multiple general purpose registers, decreases base register after
stmfd	Store multiple	Stores from multiple general purpose registers, decreases base register after
flds	Load float	Loads a 32 bit float to a floating point register
fldd	Load double	Loads a 64 bit double to a double register
fsts	Store float	Stores a 32 bit float from a floating point register
fstd	Store double	Stores a 64 bit double to a double register
fldmfd	Load float multiple	Loads multiple 32 bit floating point registers
fldmfd	Load double multiple	Loads multiple 64 bit double registers
fstmss	Store float multiple	Stores from multiple 32 bit floating point registers
fstmss	Store double multiple	Stores from multiple 64 bit double registers

**Table 6.8:** Load and store instructions

### 6.1.6 Load and Store Instructions

There is a great variety of load and store instructions available in the ARM instruction set. All operations to and from the memory system must be performed through these instructions as it is in the RISC philosophy. Load and store instructions can be grouped in categories based on the size of the data loaded or stored and the addressing mode.

Regarding the length of the data to be loaded or stored, there are instructions that can load one, two, four or eight bytes into a single register. There are also load and store instructions that can load data into or store from multiple registers. Regarding the addressing modes, there is a wide variety as well, with the most common and widely used being the register addressing mode, this uses a register value as a base address and an optional immediate offset. One variation of these instructions can use the program counter register with an offset.

Furthermore, there are options for load and store instructions that can alter the base register after the memory operation has been completed, the base register

can either be incremented or decremented by the number of words loaded or stored and the change in the base register can be done before or after the memory access. The most common load and store instructions that were studied in this work are presented in table 6.8

## 6.2 Other Instructions and Assembler Mnemonics

In this work we studied the most characteristic and important instructions as revealed by intuition and more importantly by the instruction breakdowns of various real life benchmarks. While there are more instructions than those presented above, our work has focused on the most prevalent in real benchmarks. The same applies for the variations of these instructions. In order to make our study more robust and compact we chose to leave aside instructions that are rarely used and either group them into instruction categories that are more common or into a separate category of *other instructions*. Our runtime instruction breakdown shows these uncategorized instructions to be less than 7% of the total of executed instructions. This compromise is accepted in order to improve the robustness of our models as well as usefulness and intuitive understanding of the results.

There are also some instruction flavors that were not suited for characterizations with our methodology.

**Branches** are not suited as the effects of branches cannot be studied on their own but only in a wider context of other instructions, this is not compatible with our characterization methodology.

**Load and store multiple** instructions do not allow for immediate offsets, this means that we cannot study them the same way we did for load and store single instructions. For the purposes of this study, these instructions were omitted from characterization and grouped with single load and store instructions at our evaluation.

**Register shifted arithmetic** instructions were also absent from our characterization as preliminary measurements showed little to no difference in energy and latency between them and not-shifted-operand arithmetic instructions. Furthermore, excluding these instructions from our characterization and grouping them with non-shifted variants has drastically reduced the already wide variations of integer arithmetic and logic instructions, this has made our study more robust without sacrificing too much accuracy.

**Assembler Mnemonics:** ARM has recently introduced new assembler mnemonics for its instructions. Although the ARM architecture manual follows these new mnemonics, in this work we present the previous version so as to be compatible with the disassemblers, simulators and other various tools we used as these have not yet adopted the new standard. In most cases the mnemonics remain the same and when they differ the difference is easy to spot.

## Chapter 7

# Measurement Results - Energy per Instruction

In this chapter We show the results of our measurements for both core types (Cortex-A7 and Cortex-A15) across all 16 frequency configurations as well as our remarks and insights from the results.

### **Metrics and Categorization:**

Through measuring the runtime and energy of each benchmark that we designed for each instruction as described in section 4.1 we can extrapolate the cost of each instruction in terms of total energy, latency, and power. These being presented in terms of Energy Per Instruction (EPI), Cycles Per Instruction (CPI), and Energy Per Cycle (EPC) respectively.

We have categorized the ARM instruction set in a logical way that differentiates instructions based on their functionality and what part of the processor data-path they utilize. Furthermore we differentiate internally in each category based on operands type and instruction flavor for each case. Our instruction categorization consists of four categories that are:

- Arithmetic and Logic
- Data Movement
- Compare and test
- Load and Store

In the following sections we present our findings for each instruction type. First we present the latency and CPI of each instruction. These two differ in that latency is the number of cycles it takes a RAW dependency to be satisfied whereas CPI is the maximum throughput of the processor for each instruction type when there are no dependencies. To determine these two values we used two different types of benchmarks as described above in section 4.1. For each instruction type we present the Energy Per Instruction (EPI) and Cycles per Instruction (CPI) both for when RAW dependencies are present and not.

After each instruction type analysis, we present some insights and general conclusions regarding the cost of each instruction.

## 7.1 Results Summary

Before proceeding with the details of all instruction variations and our measurements across all frequency ranges, we present a short outline of our results by grouping the myriads of instructions in categories based on their minimum and maximum energy consumption. The energy consumption is the lowest when there are no RAW dependencies between instructions and the highest when there are.

The groups we have created for this purpose are:

- **Simple Integer:** Simple integer instructions are integer arithmetic and logic instructions besides multiplications and divisions and also all register movement, and compare and test instructions for integers.
- **Simple float/double:** These are all float and double additions and subtractions along with register movement and compare.
- **Multiplication:** Multiplications for integer, float and double operands.
- **Division:** Divisions for integer, float and double operands.
- **Load:** Loads for integer, float and double operands for different cache level access.
- **Store:** Stores for integer, float and double operands for different cache level access.

The minimum and maximum energy per instruction for all the groups described above are presented in table 7.1.

Instruction	Cortex-A7		Cortex-A15	
	min EPI	max EPI	min EPI	max EPI
Simple Integer	50	80	200	450
Simple Float/Double	90	200	250	1500
Multiplication	80	340	360	1730
Division	150	1200	1270	1960
Load (L1 hit)	150	195	450	450
Store (L1 hit)	185	195	680	750
Store (L1 miss)	200		700	
Load (L1 miss)	270		1000	

**Table 7.1:** Minimum (w/o RAW) and maximum (w/ RAW) Energy per Instruction (pJ) at 1GHz

## 7.2 Integer Arithmetic and Logic Instructions

Integer logic and arithmetic instructions are described in section 6.1.2. We have categorized these instructions based on operands. The first category is instructions that have three registers as operands and the second has two registers and an immediate value. *Add*, *sub*, *and*, *orr*, *eor* and *rsb* instructions can have 3 registers or 2 registers and an immediate as operand while *mul* and *div* instructions can only accept 3 registers.

**Latency:** All instructions have a latency of one cycle, except for *mul* and *div* that have 3 and 5 cycles of latency respectively. These are the same for both core types.

**CPI:** *Add*, *sub*, *and*, *eor* and *orr* instructions with an immediate operand achieve a CPI of 0.5 at both cores. *Rsb* instructions with an immediate operand have a CPI of 1 at Cortex-A7 cores and 0.5 at the Cortex-A15 cores.

With all operands being registers, *add*, *sub*, *orr*, *eor* and *rsb* instructions have different CPIs at different cores, that is 1 for Cortex-A7 and 0.5 for Cortex-A15. *Mul* instructions have a CPI of 1 at the Cortex-A15 cores and 1.2 at Cortex-A7. The differences between the two cores in terms of throughput (CPI) can be explained by the Cortex-A7 having less ports at their register file.

A Counterintuitive result is that, *div* instructions have a better CPI of 3 at Cortex-A7 Cores than 5 at Cortex-A15. The CPI and latencies described above can be seen in table 7.2 for instructions with three register operands, and table for instructions with two register and an immediate operand.

Instruction	Cortex-A7		Cortex-A15	
	Latency	CPI	Latency	CPI
add	1	1	1	0.5
and	1	1	1	0.5
eor	1	1	1	0.5
mul	3	1.2	3	1
orr	1	1	1	0.5
rsb	1	1	1	0.5
sub	1	1	1	0.5
div	5	3	5	5

**Table 7.2:** Integer logic and arithmetic instructions with register operands: Latency and CPI

**Energy:** Table 7.4 and 7.5 show the energy per instruction for arithmetic and logic instructions with 3 register operands when there are RAW dependencies between the instructions. It is obvious that not all instructions have the same cost even when they have the same latency. For example *add* instructions have a smaller energy demand than *add* instructions across both core types and all frequencies. Additionally we can see that the energy is not analogous to the CPI

Instruction	Cortex-A7		Cortex-A15	
	Latency	CPI	Latency	CPI
add	1	0.5	1	0.5
and	1	0.5	1	0.5
eor	1	0.5	1	0.5
orr	1	0.5	1	0.5
rsb	1	1	1	0.5
sub	1	0.5	1	0.5

**Table 7.3:** Integer logic and arithmetic instructions with register and immediate operands: Latency and CPI

of each instruction. For example, although *div* instructions take five cycles to complete and *add* instructions only one, their energy is only three times greater.

The same results for EPI are shown in tables 7.6 and 7.7 for instructions with immediate operands when there are RAW dependencies. Again in this case we see the same general relationship between different instruction, that is, although they have the same CPI and operands they differ in the energy consumption. Furthermore, Instructions with an immediate operand steadily consume less energy per instruction than their three register operands counterparts.

Tables 7.8, 7.9, 7.10 and 7.11 show the results for Energy Per Instruction when there are no dependencies between instructions and thus, they are allowed to reach their maximum CPI. From the results it is obvious that when instructions do not have RAW dependencies to limit their CPI, they cost less per instruction when the throughput is greater (i.e CPI is smaller). For example, with Cortex-A15 at 800 MHz, add instructions cost 375 pJ per instruction when executed one per cycle and only 226 pJ per instruction when two instructions are executed per cycles. This however means that the Energy per Cycle of the processor is bigger when

**Cortex-A7 Energy Per Instruction (pJ)**

Freq. MHz Instr.	500	600	700	800	900	1000	1100	1200
add	63	62	61	64	72	82	94	105
and	54	53	52	54	61	69	79	89
eor	55	55	54	56	63	72	81	92
mul	116	114	112	116	128	146	166	189
orr	55	55	54	56	63	72	81	92
rsb	63	62	62	65	72	83	93	105
sub	64	63	62	65	73	83	94	105
div	178	174	170	177	195	221	251	286

**Table 7.4:** Integer logic and arithmetic instructions with 3 register operands with RAW dependencies

**Cortex-A15 Energy Per Instruction (pJ)**

<b>Freq. MHz</b> <b>Instr.</b>	800	900	1000	1100	1200	1300	1400	1500
add	386	396	432	467	499	545	600	666
and	348	354	386	415	444	482	529	585
eor	351	360	394	425	453	496	546	598
mul	764	804	846	919	975	1062	1161	1276
orr	360	369	400	436	466	509	557	612
rsb	386	392	432	467	496	546	597	667
sub	386	395	429	467	497	548	602	667
div	1148	1170	1272	1380	1468	1603	1738	1920

**Table 7.5:** Integer logic and arithmetic instructions with 3 register operands with RAW dependencies**Cortex-A7 Energy Per Instruction (pJ)**

<b>Freq. MHz</b> <b>Instr.</b>	500	600	700	800	900	1000	1100	1200
add	61	60	59	62	69	79	90	100
and	58	57	56	59	65	75	84	95
eor	60	59	58	61	68	78	89	99
orr	59	58	57	59	66	76	86	97
rsb	66	65	64	67	75	86	96	109
sub	61	60	60	62	69	80	90	101

**Table 7.6:** Integer logic and arithmetic instructions with an immediate operand with RAW dependencies

more instructions are executed per cycle. Additionally, the cost of executing 1 instruction per cycle is more than half of that of executing two instructions per cycle.

**Cortex-A7 Energy Per Instruction (pJ)**

<b>Freq. MHz</b> <b>Instr.</b>	500	600	700	800	900	1000	1100	1200
add	43	42	42	45	49	56	63	71
and	37	36	36	38	42	48	54	61
eor	42	42	42	44	49	55	62	70
orr	38	37	37	39	44	50	55	63
rsb	63	61	61	64	71	82	92	103
sub	43	42	43	45	49	57	62	71

**Table 7.10:** Integer logic and arithmetic instructions with an immediate operand without RAW dependencies

**Cortex-A15 Energy Per Instruction (pJ)**

<b>Freq. MHz</b> <b>Instr.</b>	800	900	1000	1100	1200	1300	1400	1500
add	375	384	419	455	485	532	583	646
and	354	363	397	435	459	506	547	603
eor	372	382	415	455	481	530	576	642
orr	363	372	406	441	472	514	565	619
rsb	383	393	430	466	497	546	596	663
sub	374	386	421	478	486	532	580	646

**Table 7.7:** Integer logic and arithmetic instructions with an immediate operand with RAW dependencies**Cortex-A7 Energy Per Instruction (pJ)**

<b>Freq. MHz</b> <b>Instr.</b>	500	600	700	800	900	1000	1100	1200
add	62	61	61	64	71	81	92	103
and	54	53	52	55	61	70	79	90
eor	55	54	53	55	62	71	80	91
mul	61	60	59	61	68	78	89	100
orr	55	54	54	56	63	71	82	92
rsb	59	58	58	60	68	77	87	99
sub	59	58	58	60	68	77	88	98
div	121	119	117	121	134	152	174	197

**Table 7.8:** Integer logic and arithmetic instructions with 3 register operands without RAW dependencies**Cortex-A15 Energy Per Instruction (pJ)**

<b>Freq. MHz</b> <b>Instr.</b>	800	900	1000	1100	1200	1300	1400	1500
add	226	232	255	277	295	323	353	390
and	191	197	214	233	249	269	294	324
eor	194	200	217	236	251	271	297	329
mul	321	329	357	391	412	449	490	539
orr	200	206	225	247	259	282	309	341
rsb	212	219	239	260	278	300	328	364
sub	211	217	236	260	275	300	327	362
div	1149	1169	1270	1384	1469	1604	1738	1914

**Table 7.9:** Integer logic and arithmetic instructions with 3 register operands without RAW dependencies

Cortex-A15 Energy Per Instruction (pJ)								
Freq. MHz Instr.	800	900	1000	1100	1200	1300	1400	1500
add	227	232	252	280	295	321	353	394
and	201	207	225	233	258	284	308	343
eor	226	232	253	277	293	319	352	390
orr	209	214	233	256	270	295	325	359
rsb	231	240	259	288	301	331	359	402
sub	227	233	253	278	296	321	351	395

**Table 7.11:** Integer logic and arithmetic instructions with an immediate operand without RAW dependencies

The general insight from these measurements is that at higher throughputs, processors are more energy efficient. Furthermore, instructions that have an immediate operand are less energy hungry than their all register operands counterparts. Additionally, logic instructions are cheaper than arithmetic instructions and multiplications and divisions are the most expensive of all.

When comparing the energy consumption of the two core types we see that Cortex-A15 cores have a greater EPI than Cortex-A7. Even when the two cores operate at the same frequency and achieve the same CPI for the same instruction the *big* cores consume three to five times more energy per instruction than the *little* ones. For example, at 1200 MHz *add* instructions with CPI equal to 1 consume 105 pJ at Cortex-A7 and 499 pJ at Cortex-A15 cores. At the same time *and* instructions are only three times more expensive at 1000MHz when the CPI is equal to 0.5 at both cores. The general remark is that at Cortex-A15 cores resolving dependencies between instructions is relatively more expensive energy-wise than Cortex-A7 cores.

As far as scaling of energy consumption with frequency is concerned, Cortex-A7 cores seem to have a flat energy consumption for frequencies up to 800 MHz and increase linearly from there to their maximum frequency of 1200 MHz. On Cortex-A15 cores, on the other hand, EPI scales linearly across their entire frequency range from 800 MHz to 1500 MHz.

### 7.3 Float Arithmetic Instructions

Floating point arithmetic instructions are described in detail in section 6.1.3. There are four main instructions for *addition*, *subtraction*, *multiplication* and *division*. These instructions only operate on floating point 32-bit register operands and do not accept immediate values.

**CPI and Latency:** Table 7.12 shows the latency and CPI of these instructions. It is interesting to see that all of these instructions have a latency of four or more cycles, with divisions having a latency of 15 cycles at Cortex-A7 and 7 cycles at Cortex-A15 cores.

At the same time, *additions*, *subtractions* and *multiplications* have a greater latency at the *big* Cortex-A15 cores than the *little* Cortex-A7.

In terms of CPI, or instruction throughput when there are no RAW dependencies between instructions, it is worth noticing that Cortex-A7 cores achieve a CPI of 1 for *additions*, *subtractions* and *multiplications* while at Cortex-A7 cores, these instructions have a CPI slightly greater than 1. Furthermore, *divisions* are 3 cycles faster at Cortex-A7 core when there are no dependencies between them. At Cortex-A15, divisions are faster by two cycles when there are no dependencies.

**Energy:** Tables 7.13, 7.14, 7.15, and 7.16 show the energy consumption of float arithmetic instructions with and without RAW dependencies. When there are dependencies float arithmetic instructions cost two to three times more than their integer counterparts at Cortex-A7 processors, The same is not true for Cortex-A15 cores where *multiplications* cost twice as much when the operands are floats, *divisions* cost about 50% more for floats, and *additions*, *subtractions* cost four times more. In general the same relationship for the cost between these instructions holds at floating point arithmetic instructions, with *additions* and *subtractions* being cheaper than *multiplications*, and *divisions* being the most expensive operations of all. When there are no RAW dependencies, the energy per instruction cost drops substantially, from two to four times depending on instruction type and core type. For example, at Cortex-A7 at 1000 MHz *additions* cost 199 pJ when the CPI is 4 cycles and half as much at 93 pJ when the CPI is 1 cycle. However at Cortex-A15 at the same frequency *additions* can cost more than three times less from 1471 pJ to 458 pJ with a CPI of 5 and 1 respectively.

Comparing between core types we find that at full throughput (no dependen-

Instruction	Cortex-A7		Cortex-A15	
	Latency	CPI	Latency	CPI
fadds	4	1.15	5	1
fdivs	18	15	7	5
fmuls	4	1.15	6	1
fsubs	4	1.15	5	1

**Table 7.12:** Float Arithmetic instructions: Latency and CPI

cies between instructions) and at the same corresponding frequencies, the same instructions cost from three to five times more energy at the *big* Cortex-A15 cores.

**Cortex-A7 Energy Per Instruction (pJ)**

<b>Freq. MHz</b> <b>Instr.</b>	500	600	700	800	900	1000	1100	1200
fadds	157	153	151	157	174	199	227	258
fdivs	566	546	534	556	616	702	794	903
fmuls	161	157	154	161	178	203	233	265
fsubs	158	154	152	158	175	200	227	259

**Table 7.13:** Floating point arithmetic instructions with RAW dependencies**Cortex-A15 Energy Per Instruction (pJ)**

<b>Freq. MHz</b> <b>Instr.</b>	800	900	1000	1100	1200	1300	1400	1500
fadds	1318	1347	1471	1606	1704	1842	2003	2212
fdivs	1744	1781	1944	2117	2255	2446	2656	2917
fmuls	1536	1574	1714	1869	1988	2152	2345	2575
fsubs	1318	1347	1474	1605	1701	1842	2012	2208

**Table 7.14:** Floating point arithmetic instructions with RAW dependencies**Cortex-A7 Energy Per Instruction (pJ)**

<b>Freq. MHz</b> <b>Instr.</b>	500	600	700	800	900	1000	1100	1200
fadds	71	69	68	72	81	93	105	118
fdivs	476	462	452	470	522	593	670	763
fmuls	71	70	69	72	81	93	106	118
fsubs	71	70	69	72	82	93	106	118

**Table 7.15:** Floating point arithmetic instructions without RAW dependencies**Cortex-A15 Energy Per Instruction (pJ)**

<b>Freq. MHz</b> <b>Instr.</b>	800	900	1000	1100	1200	1300	1400	1500
fadds	379	391	424	458	496	533	586	642
fdivs	1317	1344	1468	1599	1699	1837	2009	2207
fmuls	382	389	429	468	495	545	579	647
fsubs	380	386	427	467	492	535	585	646

**Table 7.16:** Floating point arithmetic instructions without RAW dependencies

## 7.4 Double Arithmetic Instructions

Double arithmetic instructions are described in detail in section 6.1.3. There are four main instructions for *addition*, *subtraction*, *multiplication* and *division*. These instructions only operate on floating point 64-bit register operands and do not accept immediate values.

**CPI and Latency:** Table 7.17 shows the latency and CPI of these instructions. At Cortex-A15 cores the latency of these instructions remains the same with their 32 bit precision counterparts (table 7.12) while the CPI is slightly worse for all instructions except divisions.

For Cortex-A7, *additions* and *subtractions* retain the same latency of four cycles as the 32 bit version, while multiplication and divisions almost double at 7 from 4 and 32 from 18 cycles respectively. All CPI characteristics also get worse at Cortex-A7 cores with *additions* and *subtractions* having a CPI of 1.28 and multiplications at 4 from 1.15, divisions get a CPI of 29.5 cycles from 15 at 32-bit precision.

Instruction	Cortex-A7		Cortex-A15	
	Latency	CPI	Latency	CPI
fadd	4	1.28	5	1.08
fdiv	32	29.5	7	5
fmul	7	4	6	1.15
fsub	4	1.28	5	1.08

**Table 7.17:** Double Arithmetic instructions: Latency and CPI

**Energy:** Tables 7.18, 7.19, 7.20, and 7.21 show the energy consumption of double arithmetic instructions with and without RAW dependencies. At cortex-A7 cores, *additions* and *subtractions* consume almost the same, if not slightly more energy, than their 32 bits flavors, while *multiplications* and *divisions* consume approximately two times more whether there are RAW dependencies or not. At Cortex-A15 cores all instructions consume slightly more energy than the corresponding instructions with 32 bit operands.

In general, double instructions are more expensive and slower than float ones, this is most noticeable at the *little* Cortex-A7 cores. With regards to scaling with frequency, the same patterns are observed as with all other instructions for both types of cores.

**Cortex-A7 Energy Per Instruction (pJ)**

<b>Freq. MHz</b> <b>Instr.</b>	500	600	700	800	900	1000	1100	1200
fadd	157	153	150	156	173	197	224	256
fdiv	963	931	908	943	1040	1190	1347	1531
fmul	269	263	257	268	296	339	386	440
fsub	157	153	150	157	173	198	226	257

**Table 7.18:** Double arithmetic instructions with RAW dependencies**Cortex-A15 Energy Per Instruction (pJ)**

<b>Freq. MHz</b> <b>Instr.</b>	800	900	1000	1100	1200	1300	1400	1500
fadd	1338	1365	1483	1617	1713	1863	2044	2238
fdiv	1767	1801	1958	2142	2274	2464	2688	2942
fmul	1555	1588	1730	1888	2004	2170	2378	2601
fsub	1334	1362	1486	1620	1722	1859	2044	2232

**Table 7.19:** Double arithmetic instructions with RAW dependencies**Cortex-A7 Energy Per Instruction (pJ)**

<b>Freq. MHz</b> <b>Instr.</b>	500	600	700	800	900	1000	1100	1200
fadd	72	70	69	72	81	93	106	120
fdiv	874	846	826	858	949	1083	1226	1393
fmul	182	177	174	186	202	231	264	300
fsub	72	70	70	73	82	93	106	120

**Table 7.20:** Double arithmetic instructions without RAW dependencies**Cortex-A15 Energy Per Instruction (pJ)**

<b>Freq. MHz</b> <b>Instr.</b>	800	900	1000	1100	1200	1300	1400	1500
fadd	406	416	458	501	529	572	622	689
fdiv	1331	1363	1485	1614	1721	1864	2039	2235
fmul	427	439	479	522	559	610	664	730
fsub	407	417	457	497	532	569	622	695

**Table 7.21:** Double arithmetic instructions without RAW dependencies

## 7.5 Integer Move Instructions

Integer move instructions are described in detail in section 6.1.4. There are two flavors of move instructions *mov* and *mvn* that copy the second argument to the first either verbatim or bitwise inverted, the second argument can be either a register or an immediate.

**CPI and Latency:** All move instructions have a latency of 1 cycle, regardless of flavor and operands type. Also, all instructions achieve a CPI of 0.5 at both core types except *mvn* instructions with both register operands that have a CPI of 1 for the *little* Cortex-A7 cores.

**Energy:** Tables 7.22, 7.23, 7.24 and 7.25 show the energy consumption of move instructions for Cortex-A7 and Cortex-A15 cores respectively. Generally *mvn* instructions are more costly than *mov* instructions, even when they achieve the same CPI. Surprisingly, instructions with an immediate operand when compared with the same instructions with two register operands consume less energy at Cortex-A7 cores and more at Cortex-A15.

In general these instructions consume roughly the same and slightly less energy per instruction as integer arithmetic and logic instructions of the same CPI and latency.

**Cortex-A7 Energy Per Instruction (pJ)**

<b>Freq. MHz</b> <b>Instr.</b>	500	600	700	800	900	1000	1100	1200
mov	55	54	53	55	62	71	81	91
mvn	65	63	63	66	74	84	95	107

**Table 7.22:** Integer move instructions with RAW dependencies

**Cortex-A15 Energy Per Instruction (pJ)**

<b>Freq. MHz</b> <b>Instr.</b>	800	900	1000	1100	1200	1300	1400	1500
mov	333	342	371	402	427	471	511	553
mvn	356	365	397	432	459	501	546	599

**Table 7.23:** Integer move instructions with RAW dependencies

**Cortex-A7 Energy Per Instruction (pJ)**

<b>Freq. MHz</b> <b>Instr.</b>	500	600	700	800	900	1000	1100	1200
mov	35	34	34	35	40	45	50	57
mvn	60	58	58	61	68	78	88	99
mov (imm)	38	37	37	39	44	49	55	63
mvn (imm)	38	38	37	39	44	50	56	64

**Table 7.24:** Integer move instructions without RAW dependencies**Cortex-A15 Energy Per Instruction (pJ)**

<b>Freq. MHz</b> <b>Instr.</b>	800	900	1000	1100	1200	1300	1400	1500
mov	179	184	200	221	237	257	274	295
mvn	211	216	234	253	277	301	319	351
mov (imm)	196	203	221	240	257	275	301	325
mvn (imm)	201	204	221	246	261	281	303	336

**Table 7.25:** Integer move instructions without RAW dependencies

## 7.6 Float Move Instructions

Float move instructions are described in detail in section 6.1.4. There are two flavors of move instructions *fcyps* and *fnegs* that copy the second register operand to the first either verbatim or bitwise inverted.

**CPI and Latency:** At Cortex-A7 cores these instructions have a latency of four cycles and can achieve a CPI of 1 cycle per instruction. At Cortex-A15 cores they have a latency of three cycles and achieve a CPI of 0.5, that is two instructions per cycle as seen in table 7.26

Instruction	Cortex-A7		Cortex-A15	
	Latency	CPI	Latency	CPI
fcyps	4	1	3	0.5
fnegs	4	1	3	0.5

**Table 7.26:** Float move instructions: Latency and CPI

**Energy:** Tables 7.27, 7.28, 7.29 and 7.30 show the energy consumption of move instructions for Cortex-A7 and Cortex-A15 cores respectively. Generally *fnegs* instructions are more costly than *fcyps* instructions, even when they achieve the same CPI. The energy footprint of float move instructions is in par with the addition and subtraction arithmetic float instructions for Cortex-A7 cores. At Cortex-A15 cores, these instructions consume approximately 2/3 of the energy of arithmetic float instructions.

**Cortex-A7 Energy Per Instruction (pJ)**

Freq. MHz Instr.	500	600	700	800	900	1000	1100	1200
fcyps	155	154	151	157	174	198	227	258
fnegs	157	154	152	159	175	199	228	260

**Table 7.27:** Float move instructions with RAW dependencies

**Cortex-A15 Energy Per Instruction (pJ)**

Freq. MHz Instr.	800	900	1000	1100	1200	1300	1400	1500
fcyps	859	869	950	1032	1098	1192	1308	1431
fnegs	850	867	952	1038	1100	1192	1308	1432

**Table 7.28:** Float move instructions with RAW dependencies

**Cortex-A7 Energy Per Instruction (pJ)**

<b>Freq. MHz</b> <b>Instr.</b>	500	600	700	800	900	1000	1100	1200
fcyps	70	69	69	72	81	93	105	118
fnegs	71	70	69	73	82	93	106	118

**Table 7.29:** Float move instructions without RAW dependencies**Cortex-A15 Energy Per Instruction (pJ)**

<b>Freq. MHz</b> <b>Instr.</b>	800	900	1000	1100	1200	1300	1400	1500
fcyps	219	222	242	267	286	311	339	373
fnegs	219	227	248	267	285	313	341	378

**Table 7.30:** Float move instructions without RAW dependencies

## 7.7 Double Move Instructions

Double move instructions are described in detail in section 6.1.4. There are two flavors of move instructions *fcpyd* and *fnegd* that copy the second register operand to the first either verbatim or bitwise inverted.

**CPI and Latency:** At Cortex-A7 cores these instructions have a latency of four cycles and achieve a CPI of 1.18 cycles per instruction.

At Cortex-A15 cores they have a latency of three cycles and achieve a CPI of 0.5, that is two instructions per cycle as seen in table 7.31. When compared with float move instructions they have the same latency in all cases. However, the max CPI they can achieve at Cortex-A7 cores is 1.18 versus 1 for float moves.

Instruction	Cortex-A7		Cortex-A15	
	Latency	CPI	Latency	CPI
fcpyd	4	1.18	3	0.5
fnegd	4	1.18	3	0.5

**Table 7.31:** Double move instructions: Latency and CPI

**Energy:** Tables 7.32, 7.33, 7.34 and 7.35 show the energy consumption of move instructions for Cortex-A7 and Cortex-A15 cores respectively. Generally *fnegd* instructions are more costly than *fcpyd* instructions, even when they achieve the same CPI. The energy footprint of double move instructions is in par with the addition and subtraction arithmetic double instructions for Cortex-A7 cores. At Cortex-A15 cores, these instructions consume approximately 2/3 of the energy of arithmetic double instructions. When compared with float move instructions, double move instructions generally consume more energy at the Cortex-A15 cores, while, at Cortex-A7 cores, they consume slightly less energy when running without RAW dependencies and slightly more when running with RAW dependencies.

Cortex-A7 Energy Per Instruction (pJ)								
Freq. MHz Instr.	500	600	700	800	900	1000	1100	1200
fcpyd	161	159	156	163	180	206	235	268
fnegd	163	160	157	164	181	206	236	270

**Table 7.32:** Double move instructions with RAW dependencies

**Cortex-A15 Energy Per Instruction (pJ)**

<b>Freq. MHz</b> <b>Instr.</b>	800	900	1000	1100	1200	1300	1400	1500
fcpyd	888	900	983	1068	1135	1236	1352	1485
fnegd	881	899	984	1075	1135	1235	1356	1484

**Table 7.33:** Double move instructions with RAW dependencies**Cortex-A7 Energy Per Instruction (pJ)**

<b>Freq. MHz</b> <b>Instr.</b>	500	600	700	800	900	1000	1100	1200
fcpyd	67	65	65	68	76	86	98	111
fnegd	67	66	65	68	76	87	99	111

**Table 7.34:** Double move instructions without RAW dependencies**Cortex-A15 Energy Per Instruction (pJ)**

<b>Freq. MHz</b> <b>Instr.</b>	800	900	1000	1100	1200	1300	1400	1500
fcpyd	222	230	252	277	290	315	346	385
fnegd	223	230	250	276	292	318	349	392

**Table 7.35:** Double move instructions without RAW dependencies

## 7.8 Integer Compare and Test Instructions

Integer compare and test instructions are described in detail in section 6.1.5. There are two compare and two test instructions, *cmp*, *cmn*, *tst* and *teq*. These instructions can compare either two registers or a register and an immediate and update the status flags according to the result.

**CPI and Latency:** All integer compare and test instructions have a latency of 1 cycle for all core types. When both operands are registers these instructions achieve a CPI of 1 at Cortex-A7 cores and 0.5 at Cortex-A15 cores. When comparing a register and an immediate, *cmp* and *cmn* achieve a CPI of 0.5 at all core types. *Tst* and *teq* can execute two at each cycle at Cortex-A7 and only one at each cycle at Cortex-A15.

**Energy:** Tables 7.36, 7.37, 7.38 and 7.39 show the energy per instruction of these instructions for both core types and both types of operands. In general, the energy of compare and test instructions is comparable and a little lower than the energy of integer arithmetic instructions with the same latency.

**Cortex-A7 Energy Per Instruction (pJ)**

<b>Freq. MHz</b> <b>Instr.</b>	500	600	700	800	900	1000	1100	1200
cmn	58	57	57	59	66	76	86	97
cmp	59	58	58	60	67	78	88	98
teq	57	56	56	59	65	75	85	96
tst	57	56	55	58	65	74	84	95

**Table 7.36:** Integer Compare and Test instructions with two register operands

**Cortex-A15 Energy Per Instruction (pJ)**

<b>Freq. MHz</b> <b>Instr.</b>	800	900	1000	1100	1200	1300	1400	1500
cmn	183	188	204	221	234	261	284	309
cmp	184	189	206	223	236	261	282	311
teq	185	188	205	222	235	259	283	306
tst	182	186	202	222	234	257	283	304

**Table 7.37:** Integer Compare and Test instructions with two register operands

**Cortex-A7 Energy Per Instruction (pJ)**

<b>Freq. MHz</b> <b>Instr.</b>	500	600	700	800	900	1000	1100	1200
cmn	40	40	40	42	47	52	60	67
cmp	40	40	40	42	47	53	59	68
teq	39	40	39	41	46	53	58	67
tst	37	37	37	39	43	49	55	62

**Table 7.38:** Integer Compare and Test instructions with a register and an immediate operand**Cortex-A15 Energy Per Instruction (pJ)**

<b>Freq. MHz</b> <b>Instr.</b>	800	900	1000	1100	1200	1300	1400	1500
cmn	191	195	213	231	241	267	293	319
cmp	191	193	211	231	242	268	292	321
teq	343	346	379	410	435	472	522	570
tst	338	345	376	407	433	475	520	570

**Table 7.39:** Integer Compare and Test instructions with a register and an immediate operand

## 7.9 Float Compare Instructions

Float compare instructions are described in detail in section 6.1.5. There are two float compare instructions, *fcmpzs* and *fcmps*, the first compares a float register with zero and the second compares two float registers, they both update the status flags according to the result.

**CPI and Latency:** All float compare instructions have a latency of 1 cycle for all core types and a maximum throughput of 1 instruction per cycle at both core types.

**Energy:** Tables 7.40 and 7.41 show the energy per instruction of these instructions for both instruction types and both core types. In general, the energy of float compare instructions is comparable and a little lower than the energy of float arithmetic instructions with the same CPI.

**Cortex-A7 Energy Per Instruction (pJ)**

<b>Freq. MHz</b> <b>Instr.</b>	500	600	700	800	900	1000	1100	1200
fcmpzs	63	63	62	66	73	84	95	107
fcmps	72	71	71	75	83	95	108	120

**Table 7.40:** Float Compare instructions

**Cortex-A15 Energy Per Instruction (pJ)**

<b>Freq. MHz</b> <b>Instr.</b>	800	900	1000	1100	1200	1300	1400	1500
fcmpzs	326	330	360	394	414	454	497	545
fcmps	347	353	388	424	445	488	533	592

**Table 7.41:** Float Compare instructions

## 7.10 Double Compare Instructions

Double compare instructions are described in detail in section 6.1.5. There are two double compare instructions, *fcmpzd* and *fcmpd*, the first compares a double register with zero and the second compares two double registers, they both update the status flags according to the result.

**CPI and Latency:** All double compare instructions have a latency of 1 cycle for all core types and a maximum throughput of 1 instruction per cycle at both core types.

**Energy:** Tables 7.42 and 7.43 show the energy per instruction of these instructions for both instruction types and both core types. In general, the energy of double compare instructions is comparable and a little lower than the energy of double arithmetic instructions with the same CPI.

**Cortex-A7 Energy Per Instruction (pJ)**

<b>Freq. MHz</b> <b>Instr.</b>	500	600	700	800	900	1000	1100	1200
fcmpzd	67	67	66	70	78	89	101	114
fcmped	77	76	77	80	90	103	115	130

**Table 7.42:** Double Compare instructions

**Cortex-A15 Energy Per Instruction (pJ)**

<b>Freq. MHz</b> <b>Instr.</b>	800	900	1000	1100	1200	1300	1400	1500
fcmpzd	337	343	374	407	429	468	511	565
fcmped	362	369	405	439	462	510	556	613

**Table 7.43:** Double Compare instructions

## 7.11 Integer Load and Store Instructions

Integer load and store instructions are described in detail in section 6.1.6.

**CPI and Latency:** Load instructions take a single cycle to complete when they hit in the L1 cache. Store instructions show latencies closer to two cycles. Specifically, in our measurements we saw 1.8 cycles per instruction at Cortex-A17 and two cycles per instruction at Cortex-A15. However for both loads and stores, when the access pattern varies so does the achieved CPI and processor frequency.

**Energy:** Load and store instructions can have a varying energy footprint depending on the memory footprint. Even when achieving the same CPI, load and store instructions consume more energy the bigger the part of the caches they access. Tables 7.44, 7.45, 7.46 and 7.47 show the energy consumption per instruction of load and store instructions according to memory footprint.

It is obvious from the results that stores are typically 50% more energy consuming than loads across all configurations. It can also be seen that energy per instruction increases as the footprint of the memory does.

**Cortex-A7 Energy Per Instruction (pJ)**

<b>Freq. MHz</b> <b>Mem</b>	500	600	700	800	900	1000	1100	1200
4K	114	114	112	118	132	149	167	189
8K	114	112	112	118	131	150	166	190
16K	110	111	111	115	128	145	162	186
32K	109	108	109	112	125	142	161	180
64K	109	108	108	113	125	142	160	179
128K	109	109	110	115	128	145	164	184
256K	115	116	116	123	139	157	178	201
512K	126	128	129	137	154	177	201	230
1024K	145	146	147	157	178	206	237	273

**Table 7.44:** Integer Load instructions

**Cortex-A15 Energy Per Instruction (pJ)**

<b>Freq. MHz</b> <b>Mem</b>	800	900	1000	1100	1200	1300	1400	1500
4K	410	421	463	500	526	582	636	702
8K	409	418	459	500	530	580	631	704
16K	407	419	459	496	527	578	630	702
32K	408	420	458	497	525	576	634	704
64K	409	421	461	498	527	573	634	703
128K	415	429	469	507	539	591	647	710
256K	428	441	482	521	552	606	667	739
512K	468	483	531	574	610	673	748	825
1024K	608	636	702	774	833	930	1041	1184

**Table 7.45:** Integer Load instructions**Cortex-A7 Energy Per Instruction (pJ)**

<b>Freq. MHz</b> <b>Mem</b>	500	600	700	800	900	1000	1100	1200
4K	149	146	147	155	172	194	220	248
8K	147	147	147	154	172	195	220	248
16K	148	146	146	154	171	194	219	247
32K	147	146	146	154	171	192	219	246
64K	148	147	147	155	172	195	220	246
128K	151	149	150	158	176	199	223	252
256K	155	157	157	166	185	210	238	268
512K	168	170	171	181	203	232	263	299
1024K	190	191	192	205	232	267	307	350

**Table 7.46:** Integer Store instructions**Cortex-A15 Energy Per Instruction (pJ)**

<b>Freq. MHz</b> <b>Mem</b>	800	900	1000	1100	1200	1300	1400	1500
4K	671	687	756	820	869	940	1034	1134
8K	686	694	760	826	869	948	1039	1137
16K	675	694	760	829	870	948	1039	1144
32K	680	697	761	826	871	953	1040	1150
64K	679	698	765	832	875	957	1047	1149
128K	686	705	771	838	883	962	1055	1164
256K	698	718	781	846	896	980	1071	1187
512K	730	751	823	890	940	1032	1127	1258
1024K	886	922	1025	1138	1238	1385	1560	1785

**Table 7.47:** Integer Store instructions

## 7.12 Float and Double Load and Store Instructions

Float load and store instructions are described in detail in section 6.1.6.

**CPI and Latency:** Float load instructions take a single cycle to complete when they hit in the L1 cache at both Cortex-A7 and Cortex-A15 cores. Double load instructions achieve 10% less instructions per cycle at Cortex-A7 cores with a CPI of 1.1 versus 1 for Cortex-A15 cores with a CPI of 1. Store instructions achieve a CPI from 1.6 to 1.8 depending on core type and precision. Float stores achieve a CPI of 1.75 at both core types while double stores 1.6 at Cortex-A7 and 1.8 at Cortex-A15.

However, when the access pattern varies so does the achieved CPI and processor frequency.

**Energy:** Since load and store instruction energy depends mostly on memory footprint, we have studied float and double load and store instructions only for the smallest memory footprint.

The energy of these instructions scales with memory footprint the same as integer variants. Nonetheless, it is interesting to see if there is any difference in energy due to the different register files they access for reading or writing.

The results of these measurements are shown in tables 7.48, 7.49, 7.50 and 7.50.

We observe that the relationship between loads and stores still holds with stores being more energy consuming than loads, additionally, float load and store instructions cost slightly less than double loads and stores.

**Cortex-A7 Energy Per Instruction (pJ)**

<b>Freq. MHz</b> <b>Instr.</b>	500	600	700	800	900	1000	1100	1200
flds	117	114	113	120	133	150	169	187
fldd	145	144	141	147	160	196	207	240

**Table 7.48:** Float and Double Load instructions

**Cortex-A15 Energy Per Instruction (pJ)**

<b>Freq. MHz</b> <b>Instr.</b>	800	900	1000	1100	1200	1300	1400	1500
flds	418	415	452	504	540	593	633	709
fldd	427	429	448	494	562	590	646	735

**Table 7.49:** Float and Double Load instructions

**Cortex-A7 Energy Per Instruction (pJ)**

<b>Freq. MHz</b> <b>Instr.</b>	500	600	700	800	900	1000	1100	1200
fsts	143	140	139	146	165	186	209	234
fstd	153	152	155	156	173	195	219	250

**Table 7.50:** Float and Double Store instructions**Cortex-A15 Energy Per Instruction (pJ)**

<b>Freq. MHz</b> <b>Instr.</b>	800	900	1000	1100	1200	1300	1400	1500
fsts	615	631	683	746	794	865	942	1030
fstd	661	677	734	791	832	922	1005	1099

**Table 7.51:** Float and Double Store instructions

## 7.13 Energy per Cycle

The previous sections described our measurements for each instruction type CPI and latency. From these two metrics it is easy to calculate the energy per cycle (EPC) of a processor for each type of instructions. Although different instructions consume different amounts of energy, they have differences in execution time as well.

To make clear that different instructions have a different impact on processor energy consumption, we have plotted the minimum and maximum EPC that we observed from our measurements of each instruction.

Tables 7.52, 7.53 and figures 7.1 and 7.2 show how energy per cycle can vary for each processor at each frequency setting.

It is visible that at Cortex-A7 cores maximum EPC can be up to five greater than minimum EPC, for Cortex-A15, maximum EPC can be as much as three times the minimum EPC.

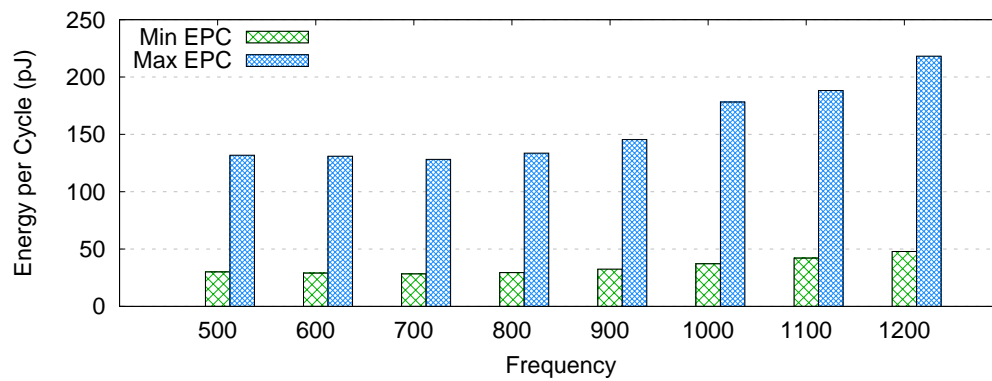
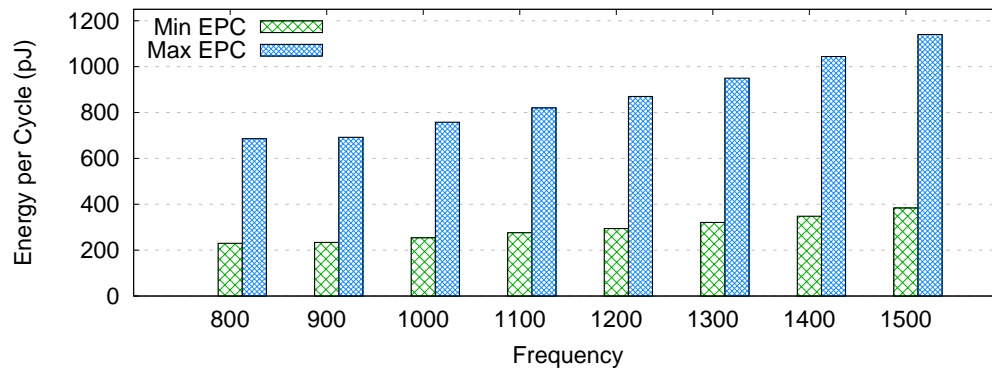
**Cortex-A7 Min and Max Energy Per Cycle (pJ)**

<b>Freq. MHz</b> <b>Mem</b>	500	600	700	800	900	1000	1100	1200
Min	30	29	28	29	33	37	42	48
Max	132	131	128	134	145	178	188	218

**Table 7.52:** Cortex-A7: Minimum and Maximum Energy per Cycle

**Cortex-A15 Min and Max Energy Per Cycle (pJ)**

<b>Freq. MHz</b> <b>Mem</b>	800	900	1000	1100	1200	1300	1400	1500
Min	230	234	254	276	294	321	348	384
Max	686	692	758	820	870	950	1044	1140

**Table 7.53:** Cortex-A15: Minimum and Maximum Energy per Cycle**Figure 7.1:** Cortex-A7: Minimum and Maximum Energy per Cycle**Figure 7.2:** Cortex-A15: Minimum and Maximum Energy per Cycle

## Chapter 8

# ARM versus x86

In appendix A we present some earlier work on the energy consumption of Intel x86 processors. The methodology we used in the two different architectures is very similar except for the fact that at our X86 experimental platform we relied on processor counters to measure the energy of instructions while for ARM processors we used actual energy readings from dedicated sensors.

While the two processor architectures differ greatly in terms of architecture, instructions set, frequency range and fabrication technology we can have some rough quantitative comparison between the two. The ARM processors in our study were fabricated at 28nm while the Intel processor at 32 nm, The ARM processors can reach a top frequency of 1.5 GHz while Intel can reach 3.3 GHz. The ARM processors we study are 32-bit wide while the Intel processors have 64-bit datapaths.

**Instruction set differences:** ARM processors implement the ARM instruction set while Intel processors the X86 instructions set. The main difference between these two is that all arithmetic operations in ARM must be done between registers while for X86 a memory location can also be an operand. The second great difference is the format of arithmetic instructions, while at the ARM instruction set these instructions have a three register format with one destination register and two source registers, at X86 they have only two and the destination register is the same as one of the source registers.

In this section we will compare similar instructions for both architectures at their maximum frequency.

**Integer arithmetic instructions:** Simple arithmetic instructions can be executed one at a cycle when there are dependencies at both cores. When there are no dependencies the best CPI of ARM is 0.5 while Intel's is better at 0.33. The energy cost of simple arithmetic instructions is two times higher at Intel at around 1100 pJ per instruction against 600 for Cortex-A15.

Multiplications achieve similar CPIs at both ARM and Intel processors. The energy cost however differs greatly with ARM multiplications costing 1300 and 550 pJ when there are dependencies and when there are not, while the same energy

cost at our Intel case study is 7600 pJ and 2800 pJ, almost five times the energy for an instruction.

Divisions have greater CPIs at the X86 processor, close to 30 cycles per instruction when there are dependencies while the CPI at Cortex-A15 is just 5 cycles. Energy is also much greater than ARM at 100000 pJ compared to just 2000 pJ at Cortex-A15, that is almost 50 times more energy.

**Float/double arithmetic instructions:** While the floating point operations at Intel were executed by a vector unit 128 bits wide, it is possible to see the difference for float and double since we studied all variations of instructions, including using only one 32 bit lane or a single 64 bit lane.

Performance wise, at ARM float and double instructions can achieve CPIs of 5-6 cycles per instructions when there are dependencies, while the X86 processor had a CPI of 3-5 for the same case. When there are no dependencies both processors achieved similar CPIs a little over 1.

Comparing the Energy per instruction for both processors, we find that intel has roughly the same energy for both float and double at 7900 pJ to 12750 pJ when there are dependencies and 3200 pJ to 4500 pJ when there are no dependencies between instructions. ARM cores consume 1/3 of x86 when there are dependencies and 1/5 of x86 when there are not.

**Load/store instructions:** At the minimum memory footprint at the x86 core, loads consume 2000 pJ and stores 3000 pJ, while at ARM the same instructions consume 700 pJ and 1100 pJ respectively. This means that L1 cache loads and stores cost three times as much at Intel than in ARM. At the maximum memory footprint of 2 MB loads and stores at the x86 core cost 20 nJ and 30 nJ respectively while at Cortex-A15 they cost 1800 pJ and 3000 pJ, this is 10 times less. This means that when accessing higher levels of the cache at Intel processors costs much more than at ARM.

**Conclusions:** In this short comparison we see great energy and performance differences between Intel and ARM processors, Intel processors seem to consistently consume more energy per instruction than ARM. However keep in mind the following limitations when considering this comparison.

- The ARM processors are fabricated at 28 nm while Intel's at 32 nm.
- ARM processors are 32 bit wide while Intel offer 64 bit datapaths.
- The instructions are from two different instruction sets not compatible with each-other.
- Our ARM measurements we done with actual sensors while the Intel measurements are provided by architectural registers inside the processor itself, making them less reliable and accurate.
- The Intel processor operates at 3.3 GHz while the ARM processor we compared it with can only reach 1.5 GHz.

## Chapter 9

# Evaluation

In this chapter we describe the evaluation of our instruction characterization. To evaluate our measurements and characterization, we have derived a simple energy model and tested it on a set of real benchmarks separate from the ones used to develop our characterization.

In the following sections we describe our evaluation benchmarks and the criteria behind our choices. Afterwards, we describe the energy model mathematical formula and the relevant coefficients for its variables, and finally, we show the results of our evaluation for both core types and all frequencies.

### 9.1 Evaluation Benchmarks

The evaluation benchmarks include some trigonometric functions from the C math library, 2D matrix multiplication, some array sorting algorithms, a fibonacci function, a nqueens solver program, and some whetstone and linpack benchmarks.

We have run all these benchmarks on all cores, with each core running an identical thread. We have measured the energy and execution time of these benchmarks on the same platform and the same methodology as the benchmarks we used to derive our energy model. To get the instruction breakdown of the evaluation benchmarks we have used the *gem5* simulator [15].

We have selected these benchmarks with three basic criteria:

- **Diversity:** The benchmarks must be diverse to cover all basic instruction categories with different mixtures. Additionally, the benchmarks must have diverse power requirements. As seen in figures 9.1 and 9.2, the energy per cycle of these benchmarks varies around 25% for Cortex-A7 and 45% for Cortex-A15.
- **Runtime behavior:** The benchmarks must present the same time and energy characteristics across runs, randomness is avoided in all benchmarks.
- **Gem5 friendly:** The benchmarks must avoid system calls and must be compiled with the *-static* compiler flag.

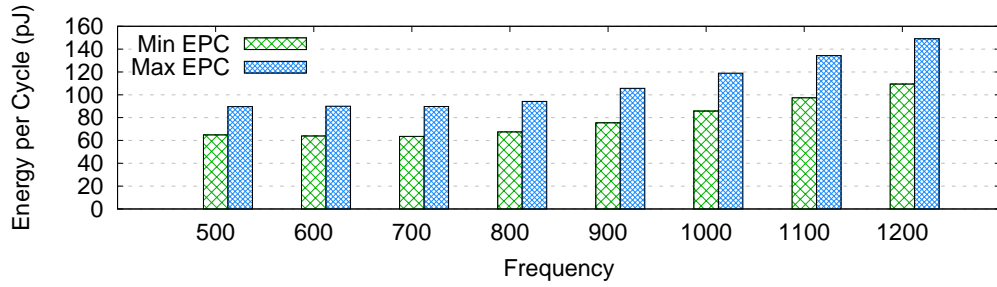


Figure 9.1: Cortex-A7: Benchmark Min and Max EPC

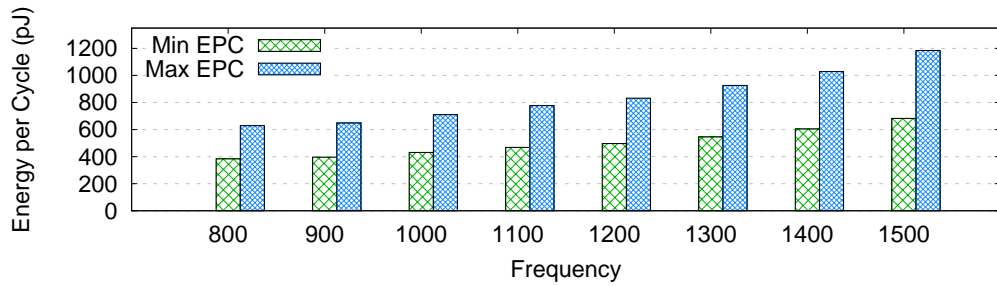


Figure 9.2: Cortex-A15: Benchmark Min and Max EPC

### Mathematical functions:

We have used *acos*, *atan*, *log*, *log10* and *sqrt* as mathematical functions from the standard C math library. These benchmarks vary in terms of energy consumption as well as in instruction mix.

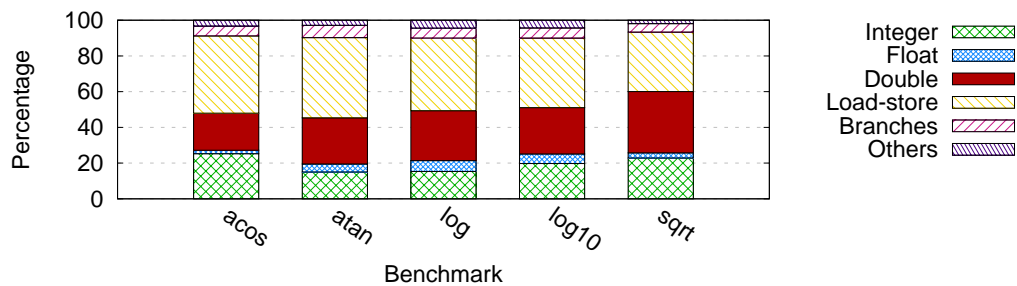


Figure 9.3: Executed Instructions Breakdown for math benchmarks %

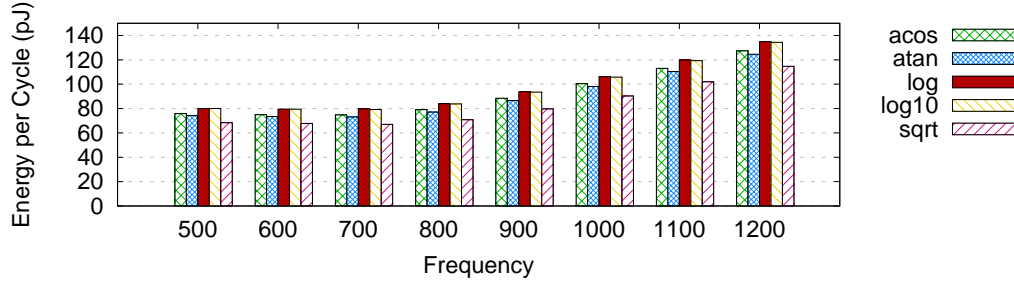


Figure 9.4: Cortex-A7: Measured EPC for math benchmarks

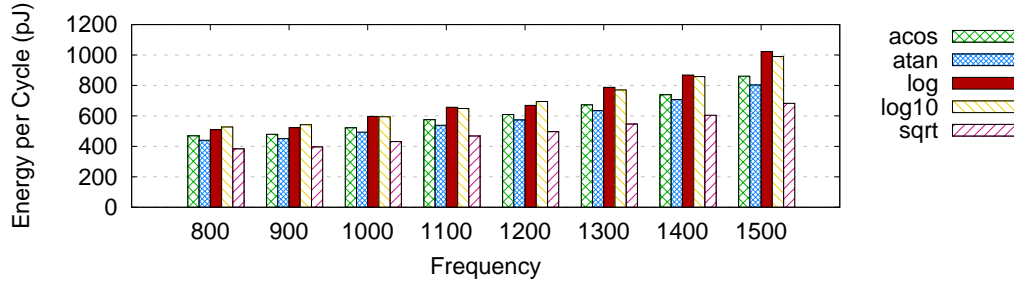


Figure 9.5: Cortex-A15: Measured EPC for math benchmarks

We can see from figure 9.3 that the instruction breakdown of these benchmarks differs in all groups of instructions.

As far as energy is concerned, from figures 9.4 and 9.5 we can see that *acos* steadily consumes more energy per cycle (EPC) than *atan* in both core types and all frequencies. *Log* and *log10* consume almost the same energy per cycle across all configurations, while *log* and *log10* consume noticeably more than *sqrt*.

### Sorts:

We have examined the Bubblesort and Selectionsort algorithms on arrays of integers, doubles, and floats. The different instruction breakdowns are visible in figure 9.6. The energy differences are visible at figures 9.7 and 9.8 where double precision sorts require more energy than float sorts and these in turn require more energy than integer sorts. Generally, insertion sort requires less energy per cycle than bubblesort for the same precision.

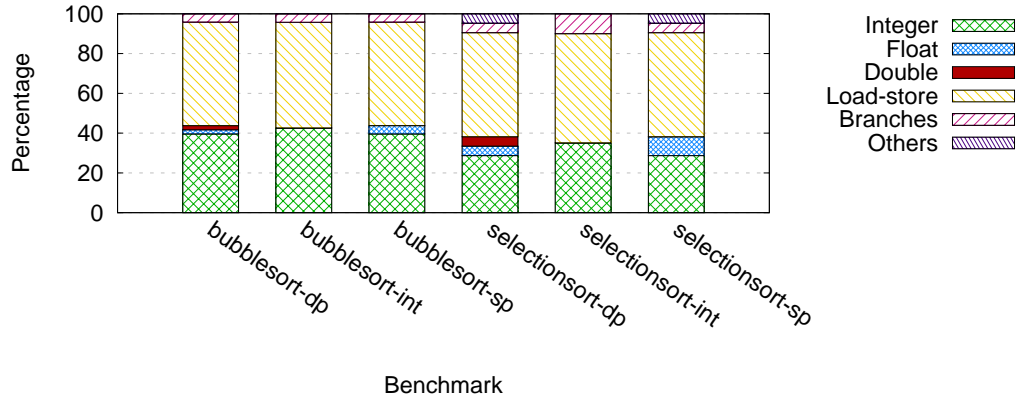


Figure 9.6: Executed Instructions Breakdown for sort benchmarks %

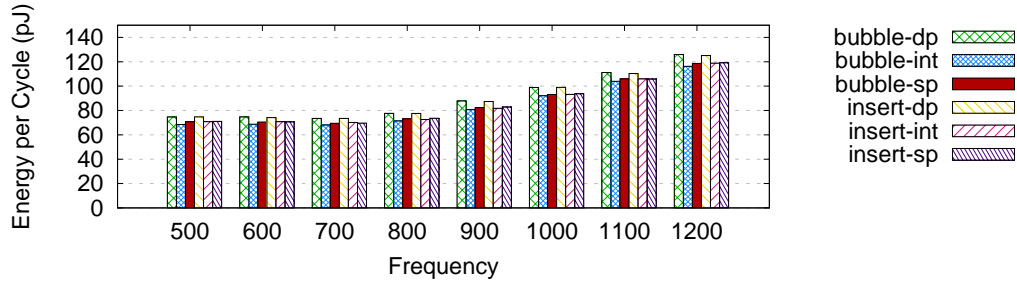


Figure 9.7: Cortex-A7: Measured EPC for sort benchmarks

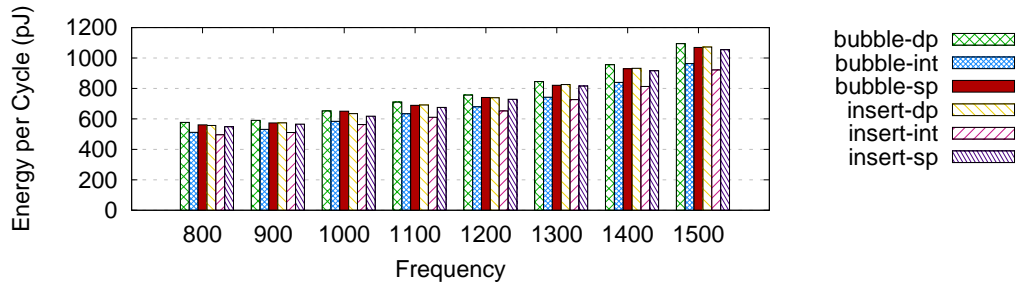


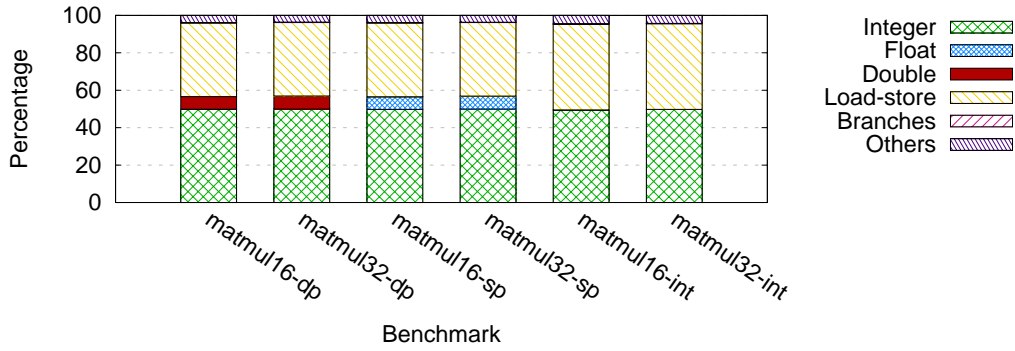
Figure 9.8: Cortex-A15: Measured EPC for sort benchmarks

### Matmul:

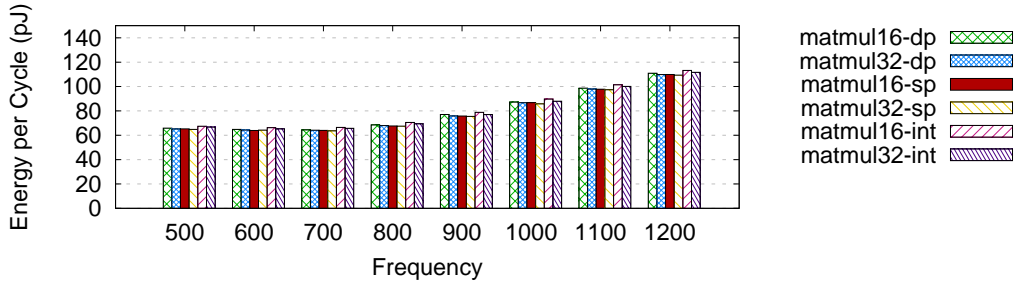
We have examined a simple matrix multiplication algorithm on arrays of doubles, floats and integers, and we have examined matrix sizes of *16 by 16* and *32 by 32*. These give a total of six different cases for matmul benchmarks.

The different instruction breakdowns are visible in figure 9.9. It is obvious that in the double benchmarks float instructions are replaced by double instructions and as the array size get bigger the percentage of branches lowers.

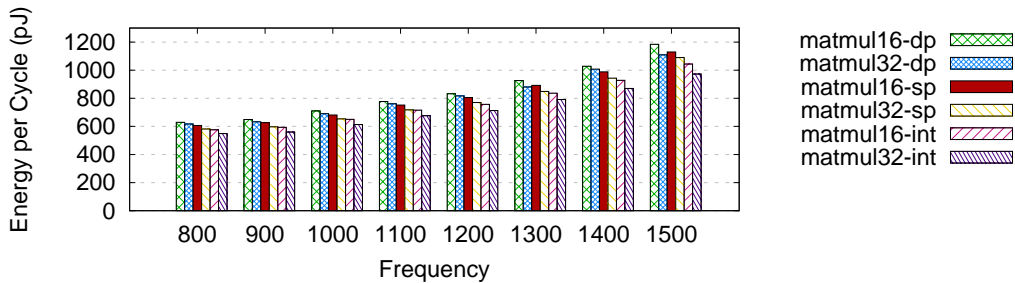
The energy differences are visible at figures 9.10 and 9.11. The general trend is that as the array gets bigger, the energy per cycle demand lowers, this is more visible at Cortex-A15 and less in Cortex-A7. Additionally, the type of the array also seems to have some influence on the energy demand of the benchmark with integer arrays being cheaper than float and double at Cortex-A15 cores. At Cortex-A7 cores however this is not the case.



**Figure 9.9:** Executed Instructions Breakdown for matmul benchmarks %



**Figure 9.10:** Cortex-A7: Measured EPC for matmul benchmarks



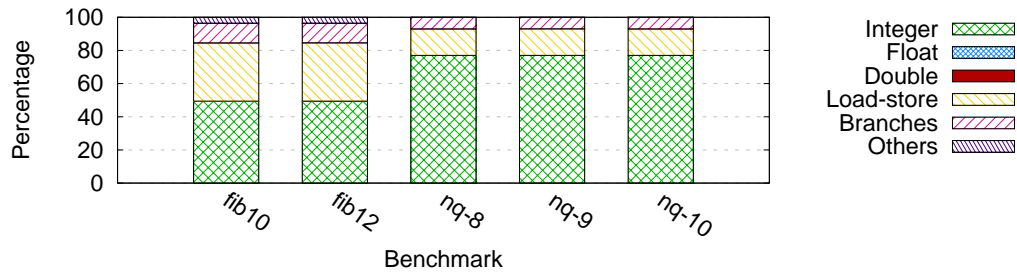
**Figure 9.11:** Cortex-A15: Measured EPC for matmul benchmarks

### Fibonacci, Nqueens:

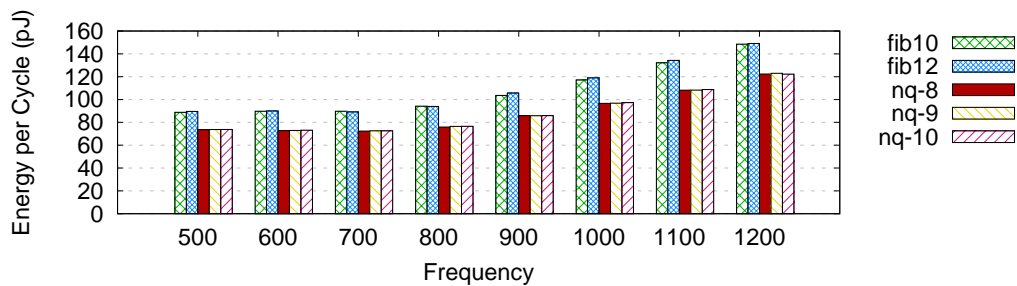
For integer instruction dominated benchmarks, we have tried a fibonacci function and a nqueens that we group together in our description for brevity. For the Nqueens benchmarks we have three different board sizes of 8, 9, and 10. For the fibonacci we have arguments of 10 and 12.

The different instruction breakdowns are visible in figure 9.12. It is visible that all these benchmarks are dominated by integer and load/store instructions. Furthermore, the fibonacci benchmark appears to use a lot more load/store and branch instructions, this is easlily attributed to the recursive function calls of the benchmark.

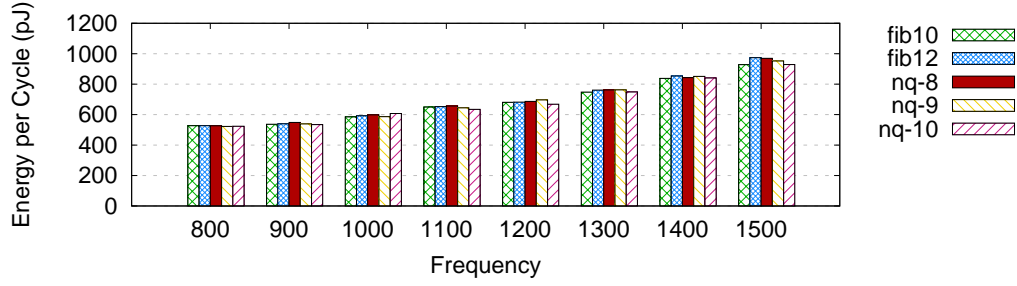
All variations of nqueens seem to have the same instruction breakdown. The energy differences are visible at figures 9.13 and 9.14. An interesting observation is that fibonacci consumes more energy per cycle than the nqueens benchmarks at Cortex-A7 cores while, at Cortex-A15 the have roughly the same energy per cycle. All Nqueens variations consume roughly the same energy on both core types.



**Figure 9.12:** Executed Instructions Breakdown for Fibonacci and Nqueens benchmarks



**Figure 9.13:** Cortex-A7: Measured EPC for Fibonacci and Nqueens benchmarks

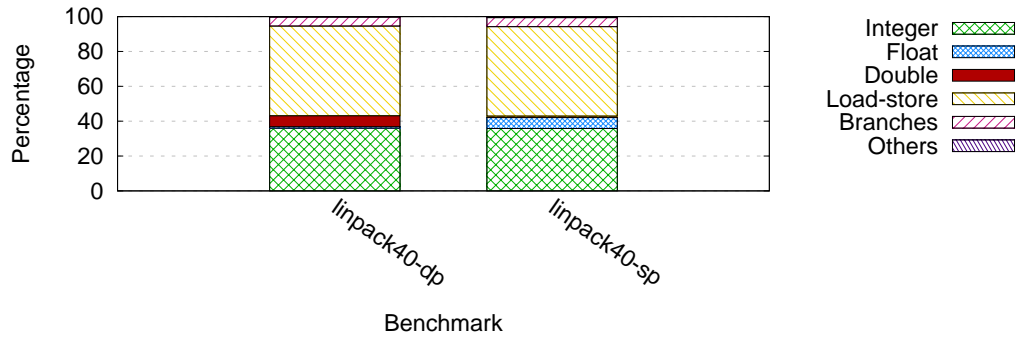


**Figure 9.14:** Cortex-A15: Measured EPC for Fibonacci and Nqueens benchmarks

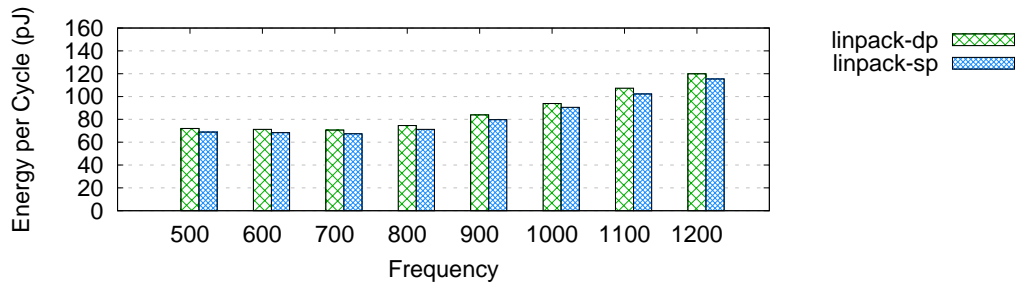
### Linpack:

In our evaluation we also examined Linpack [61] benchmarks for float and double precision. The different instruction breakdowns are visible in figure 9.15.

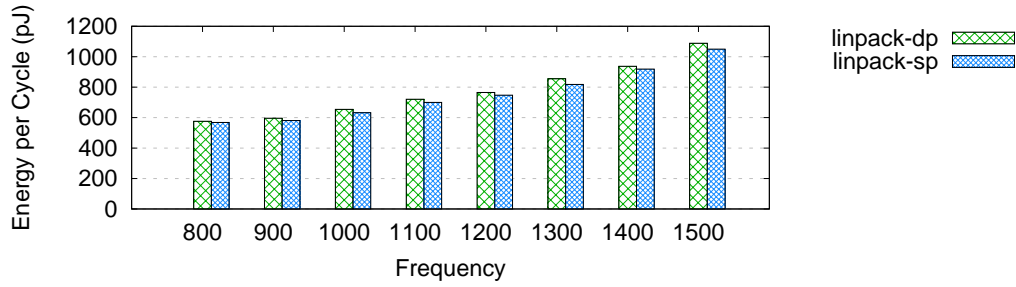
The energy differences are visible at figures 9.16 and 9.17. Double precision benchmarks seem to consume slightly more energy per cycle than float benchmarks.



**Figure 9.15:** Executed Instructions Breakdown for Linpack benchmarks %



**Figure 9.16:** Cortex-A7: Measured EPC for Linpack benchmarks

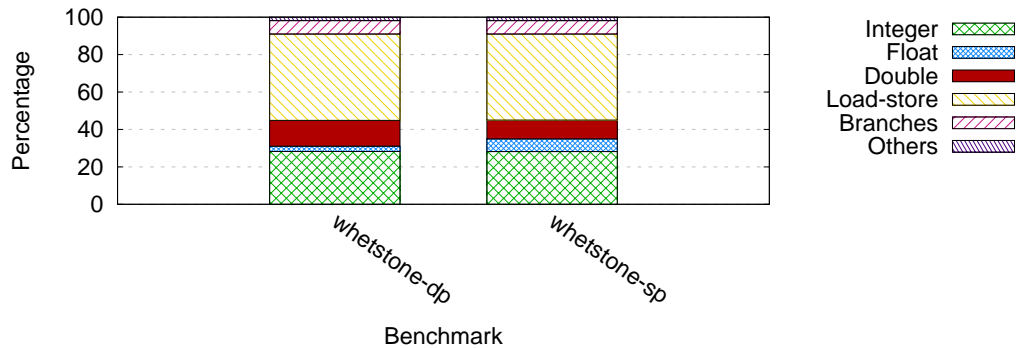


**Figure 9.17:** Cortex-A15: Measured EPC for Linpack benchmarks

### Whetstone:

Finally, for our evaluation we used the whetstone benchmarks with float and double precision. The different instruction breakdowns are visible in figure 9.18. Although there is a clear difference in the executed instructions between the two precision variants with double precision benchmarks using more double instructions and single precision benchmarks using more float instructions, these two are not exact opposites, this can be either attributed to the code of the benchmarks itself, or, probably to inefficiencies in our trace generation with the *gem5* simulator.

The energy differences are visible at figures 9.19 and 9.20. While double precision whetstone consumes more energy per cycle at Cortex-A7 cores, the same is not true for Cortex-A15. The difference, though subtle has to be noted.



**Figure 9.18:** Executed Instructions Breakdown for Whetstone benchmarks %

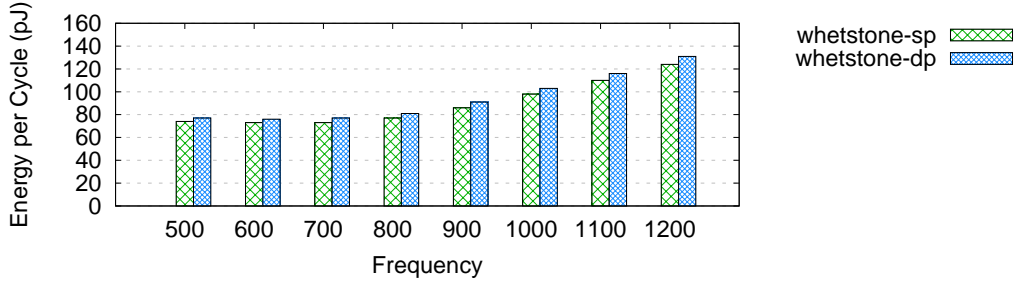


Figure 9.19: Cortex-A7: Measured EPC for Whetstone benchmarks

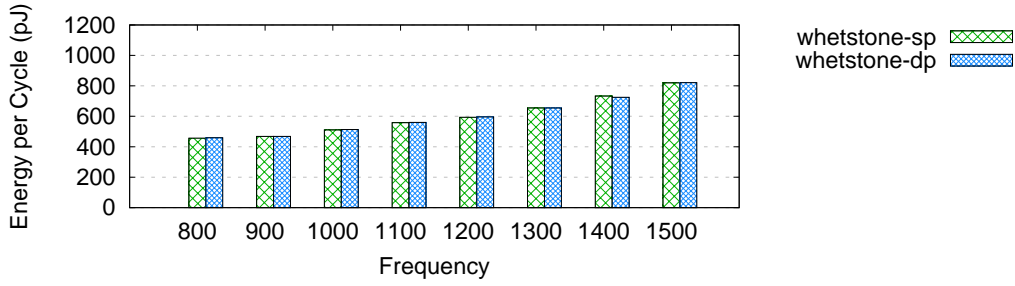


Figure 9.20: Cortex-A15: Measured EPC for Whetstone benchmarks

## 9.2 Energy Model

The simple model we have derived to test our instruction characterization relies on two factors. The first is an energy cost per cycle and the second is an additional energy cost for every instruction executed. Thus, our energy model has the form:

$$\text{Energy} = \text{Cycles} \times \text{EPC}_{\min} + \sum_{\text{Instructions}} \mathbf{I}_{\text{Count}} \times \mathbf{I}_{\text{Energy}}$$

Where

- **Energy:** The total energy of the program
- **Cycles:** The runtime of the program in cycles
- **EPC<sub>min</sub>:** The minimum EPC for that processor and frequency
- **I<sub>Count</sub>:** The number of instructions executed for each type
- **I<sub>Energy</sub>:** The energy contribution of each instruction type

To put it simply, we include a base energy cost that is equal to the minimum EPC times the number of cycles a program executes and then we add an extra cost for every instruction executed.

To calculate the energy contribution of each instruction we have simply subtracted the  $EPC_{min}$  times the Instruction latency from the total energy of the instruction:

$$I_{Energy} = I_{EPI} - EPC_{min} \times I_{Latency}$$

Where

- $I_{Latency}$ : The latency of each instruction as measured in chapter 7
- $I_{EPI}$ : The Energy of each instruction as measured in chapter 7
- $EPC_{min}$  The minimum EPC for that processor and frequency
- $I_{Energy}$ : The energy contribution of each instruction type

**Example:** Lets say that we want to calculate the energy contribution for add instructions with register operands for Cortex-A7 cores at 1000MHz. From table 7.52 we see that the minimum EPC for that core and frequency is 37 pJ. From tables 7.2 and 7.4 we see that these instructions have a latency of 1 cycle and an EPI of 82 pJ. so:  $I_{Energy} = I_{EPI} - EPC_{min} \times I_{Latency} = 82 - 1 \times 37 = 45pJ$ .

With the same process we calculated all the instruction energy contributions for all cores and all frequencies, these are shown in tables 9.1 to 9.4

### Energy contribution analysis

From the results of our extensive characterization in chapter 7 we see that different instructions have different energy and latency characteristics. The bottom line is that energy of each instruction is not always directly proportional to the number of cycles it needs, and since our energy model takes into account the energy per instruction as well as the latency, an instruction with higher EPI will not always have an energy contribution higher than that of an instruction with lower EPI as the instruction latency also plays an important role.

The gist of this approach is that the energy of an instruction with high latency will be amortized into the many cycles it takes to execute, and since instruction with high latencies have lower energy per cycle metrics it is expected that these will also have lower contributions in energy.

Still, instructions with the same latencies will have energy model contributions proportionate to their EPIs, but as we stated before, this is not the case for all instructions. For example integer additions have a bigger EPI and a correspondingly bigger model energy contribution than logical *and* instructions. At the same time, float additions have a lower EPI and lower energy contribution than float multiplications at Cortex-A7 cores, while at Cortex-A15 float multiplications have a lower energy contribution than additions although their EPI is higher. This is due to the fact that float additions and multiplications have the same latency at Cortex-A7 while at Cortex-A15 multiplications are slower.

**Cortex-A7 instruction energy contribution (pJ)**

<b>Freq. MHz</b> <b>Instr.</b>	500	600	700	800	900	1000	1100	1200
add	33	33	33	35	39	45	52	57
and	24	24	24	25	28	32	37	41
eor	25	26	26	27	30	35	39	44
mul	26	27	28	29	29	35	40	45
orr	25	26	26	27	30	35	39	44
rsb	33	33	34	36	39	46	51	57
sub	34	34	34	36	40	46	52	57
div	28	29	30	32	30	36	41	46
add (imm)	31	31	31	33	36	42	48	52
and (imm)	28	28	28	30	32	38	42	47
eor (imm)	30	30	30	32	35	41	47	51
orr (imm)	29	29	29	30	33	39	44	49
rsb (imm)	36	36	36	38	42	49	54	61
sub (imm)	31	31	32	33	36	43	48	53
fadds	37	37	39	41	42	51	59	66
fdivs	26	24	30	34	22	36	38	39
fmuls	41	41	42	45	46	55	65	73
fsubs	38	38	40	42	43	52	59	67
fadddd	37	37	38	40	41	49	56	64
fdivd	3	3	12	15	0	6	3	0
fmuld	59	60	61	65	65	80	92	104
fsubd	37	37	38	41	41	50	58	65
mov	25	25	25	26	29	34	39	43
mvn	35	34	35	37	41	47	53	59
mov (imm)	23	22.5	23	24.5	27.5	30.5	34	39
mvn (imm)	23	23.5	23	24.5	27.5	31.5	35	40
fcyps	35	38	39	41	42	50	59	66
fnegs	37	38	40	43	43	51	60	68
fcpyd	41	43	44	47	48	58	67	76
fnegd	43	44	45	48	49	58	68	78
cmn	28	28	29	30	33	39	44	49
cmp	29	29	30	31	34	41	46	50
teq	27	27	28	30	32	38	43	48
tst	27	27	27	29	32	37	42	47
cmn (imm)	25	25.5	26	27.5	30.5	33.5	39	43
cmp (imm)	25	25.5	26	27.5	30.5	34.5	38	44
teq (imm)	24	25.5	25	26.5	29.5	34.5	37	43
tst (imm)	22	22.5	23	24.5	26.5	30.5	34	38

**Table 9.1:** Cortex-A7: Instruction energy contribution

<b>Freq. MHz</b> <b>Instr.</b>	500	600	700	800	900	1000	1100	1200
fcmpzs	33	34	34	37	40	47	53	59
fcmps	42	42	43	46	50	58	66	72
fcmpzd	37	38	38	41	45	52	59	66
fcmped	47	47	49	51	57	66	73	82
ldr	84	85	84	89	99	112	125	141
str	95	93.8	96.6	102.8	112.6	127.4	144.4	161.6
flds	87	85	85	91	100	113	127	139
fsts	90.5	89.2	90	95.2	107.2	121.2	135.5	150
fdd	112	112.1	110.2	115.1	123.7	155.3	160.8	187.2
fstd	105	105.6	110.2	109.6	120.2	135.8	151.8	173.2

**Table 9.2:** Cortex-A7: Instruction energy contribution**Cortex-A15 instruction energy contribution (pJ)**

<b>Freq. MHz</b> <b>Instr.</b>	800	900	1000	1100	1200	1300	1400	1500
add	156	162	178	191	205	224	252	282
and	118	120	132	139	150	161	181	201
eor	121	126	140	149	159	175	198	214
mul	74	102	84	91	93	99	117	124
orr	130	135	146	160	172	188	209	228
rsb	156	158	178	191	202	225	249	283
sub	156	161	175	191	203	227	254	283
div	0	0	2	0	0	0	0	0
add (imm)	145	150	165	179	191	211	235	262
and (imm)	124	129	143	159	165	185	199	219
eor (imm)	142	148	161	179	187	209	228	258
orr (imm)	133	138	152	165	178	193	217	235
rsb (imm)	153	159	176	190	203	225	248	279
sub (imm)	144	152	167	202	192	211	232	262

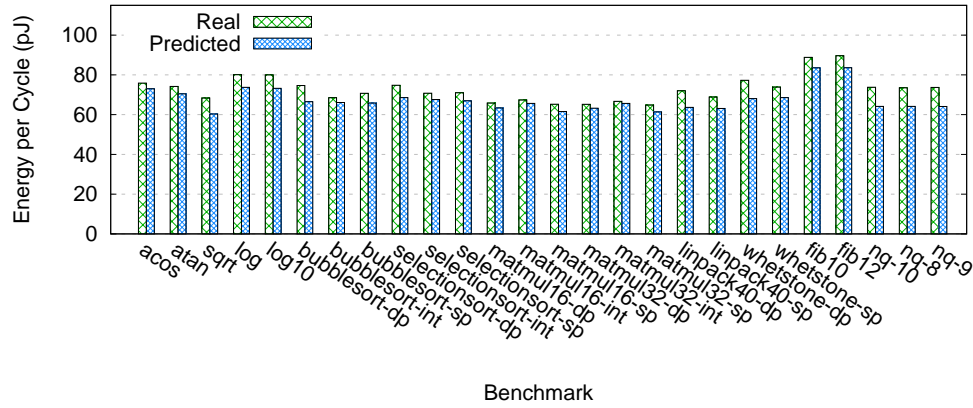
**Table 9.3:** Cortex-A15: Instruction energy contribution

<b>Freq. MHz</b> <b>Instr.</b>	800	900	1000	1100	1200	1300	1400	1500
fadds	168	177	201	226	234	237	263	292
fdivs	134	143	166	185	197	199	220	229
fmuls	156	170	190	213	224	226	257	271
fsubs	168	177	204	225	231	237	272	288
fadddd	188	195	213	237	243	258	304	318
fdivdd	157	163	180	210	216	217	252	254
fmuldd	175	184	206	232	240	244	290	297
fsubdd	184	192	216	240	252	254	304	312
mov	103	108	117	126	133	150	163	169
mvn	126	131	143	156	165	180	198	215
mov (imm)	81	86	94	102	110	114.5	127	133
mvn (imm)	86	87	94	108	114	120.5	129	144
fcyps	169	167	188	204	216	229	264	279
fnegs	160	165	190	210	218	229	264	280
fcpyd	198	198	221	240	253	273	308	333
fnegd	191	197	222	247	253	272	312	332
cmn	68	71	77	83	87	100.5	110	117
cmp	69	72	79	85	89	100.5	108	119
teq	70	71	78	84	88	98.5	109	114
tst	67	69	75	84	87	96.5	109	112
cmn (imm)	76	78	86	93	94	106.5	119	127
cmp (imm)	76	76	84	93	95	107.5	118	129
teq (imm)	228	229	252	272	288	311.5	348	378
tst (imm)	223	228	249	269	286	314.5	346	378
fcmpzs	96	96	106	118	120	133	149	161
fcmps	117	119	134	148	151	167	185	208
fcmpzd	107	109	120	131	135	147	163	181
fcmped	132	135	151	163	168	189	208	229
ldr	180	187	209	224	232	261	288	318
str	211	219	248	268	281	298	338	366
flds	188	181	198	228	246	272	285	325
fsts	212.5	221.5	238.5	263	279.5	303.2	333	358
f added	197	195	194	218	268	269	298	351
fstd	247	255.8	276.8	294.2	302.8	344.2	378.6	407.8

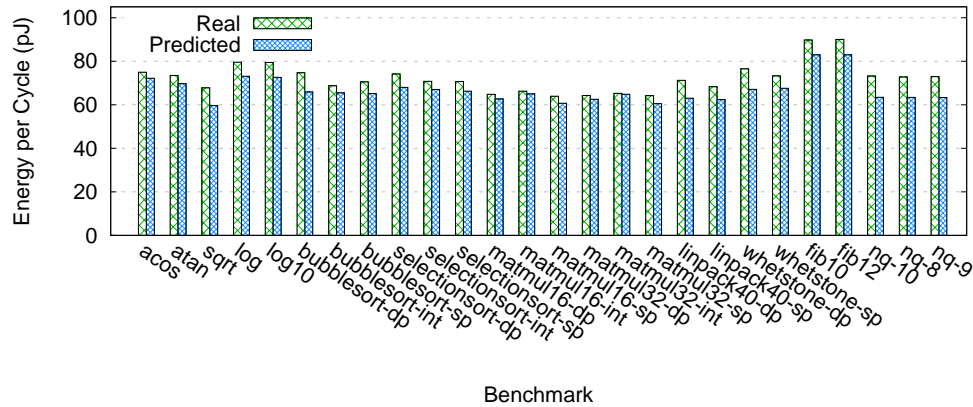
**Table 9.4:** Cortex-A15: Instruction energy contribution

### 9.3 Evaluation Results

In this section we present the predictions of our model alongside the actual energy consumption of the evaluation benchmarks. To normalize the results we do not present the total energy consumption but the EPC metric for both the real and predicted energy values, the results are shown for all frequency settings at both core types at figures 9.21 to 9.36.



**Figure 9.21:** Cortex-A7 at 500MHz: Real VS Model Prediction EPC



**Figure 9.22:** Cortex-A7 at 600MHz: Real VS Model Prediction EPC

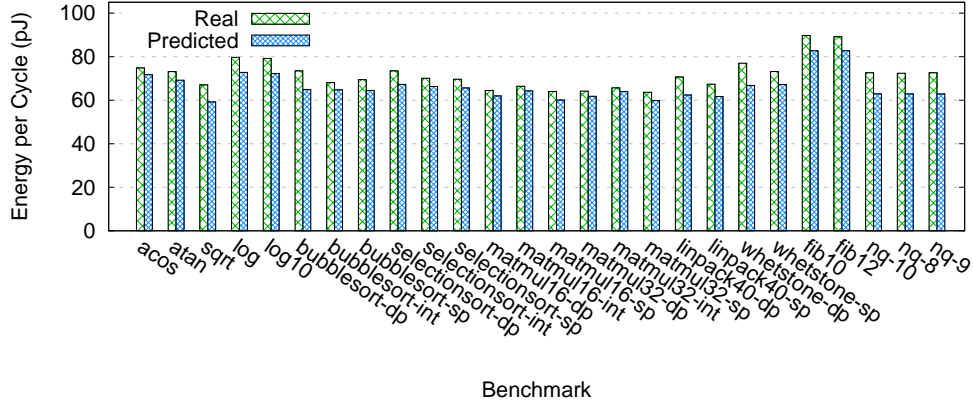


Figure 9.23: Cortex-A7 at 700MHz: Real VS Model Prediction EPC

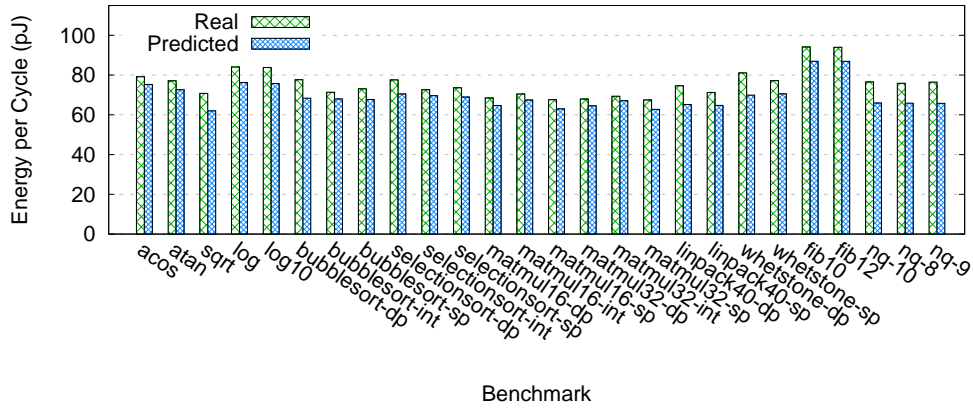


Figure 9.24: Cortex-A7 at 800MHz: Real VS Model Prediction EPC

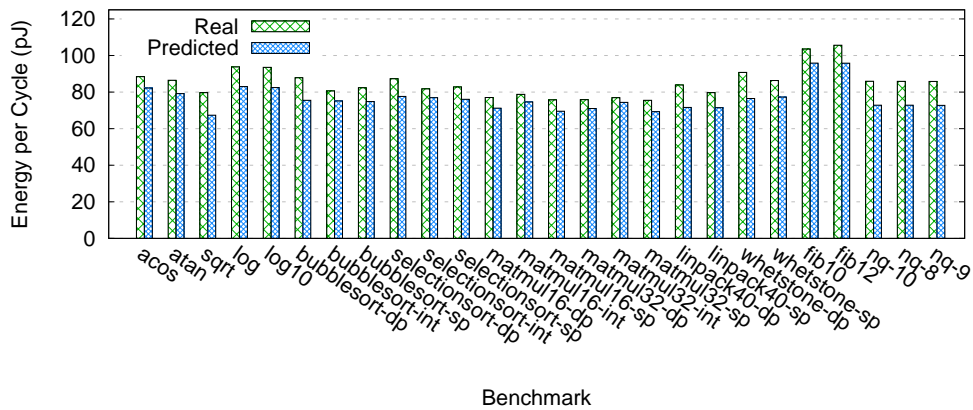


Figure 9.25: Cortex-A7 at 900MHz: Real VS Model Prediction EPC

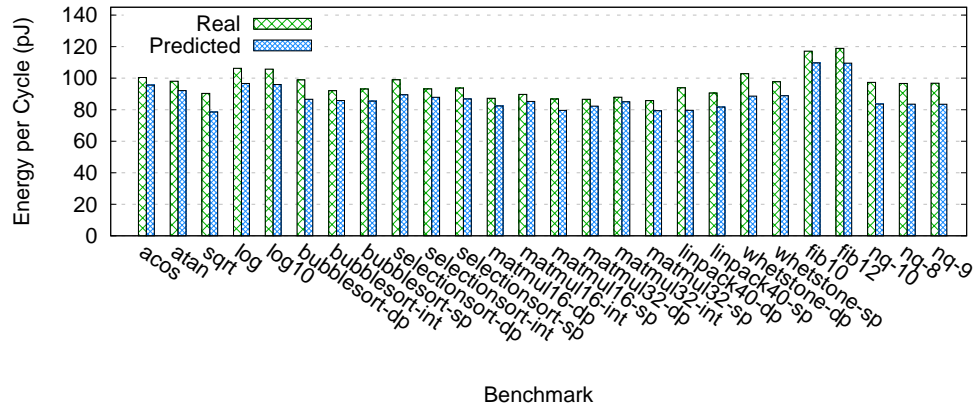


Figure 9.26: Cortex-A7 at 1000MHz: Real VS Model Prediction EPC

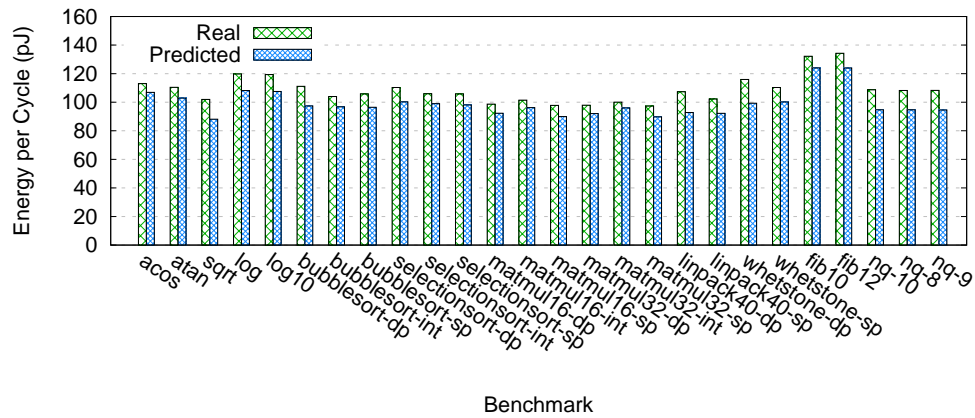


Figure 9.27: Cortex-A7 at 1100MHz: Real VS Model Prediction EPC

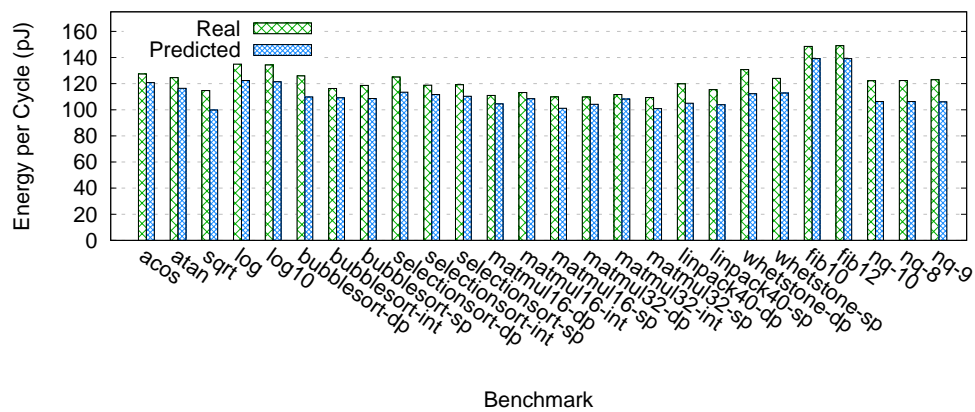


Figure 9.28: Cortex-A7 at 1200MHz: Real VS Model Prediction EPC

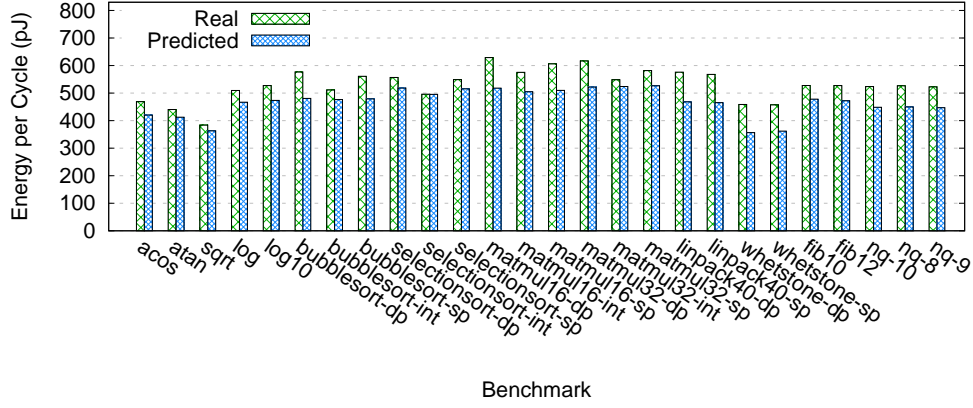


Figure 9.29: Cortex-A15 at 800MHz: Real VS Model Prediction EPC

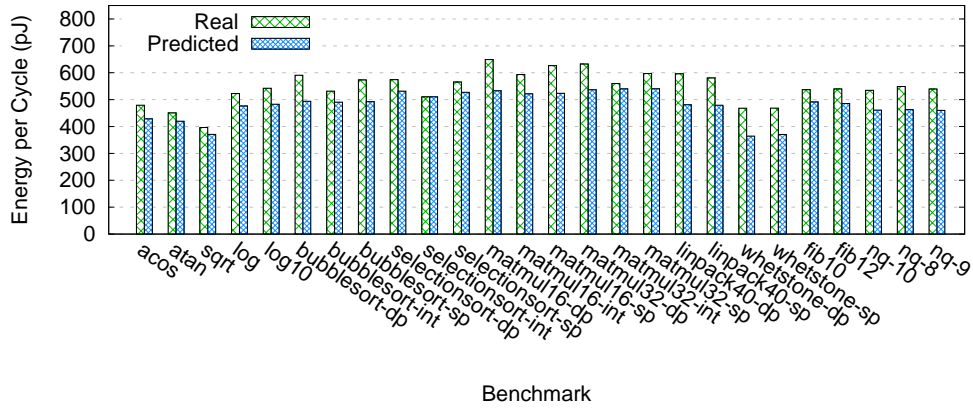


Figure 9.30: Cortex-A15 at 900MHz: Real VS Model Prediction EPC

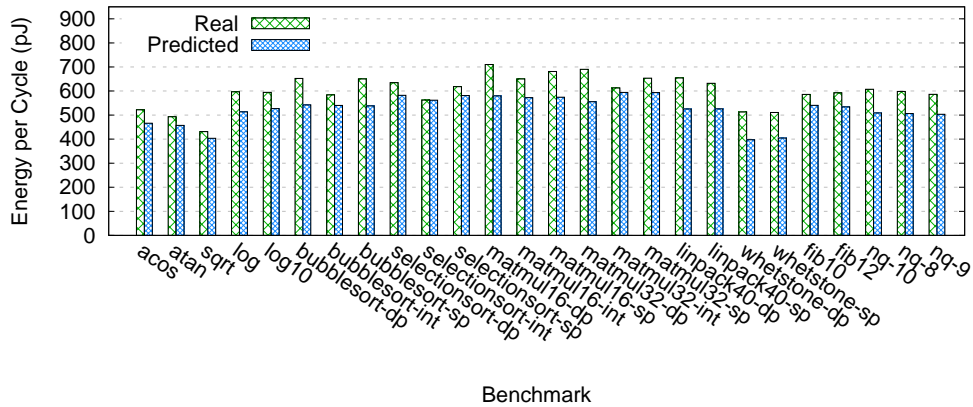
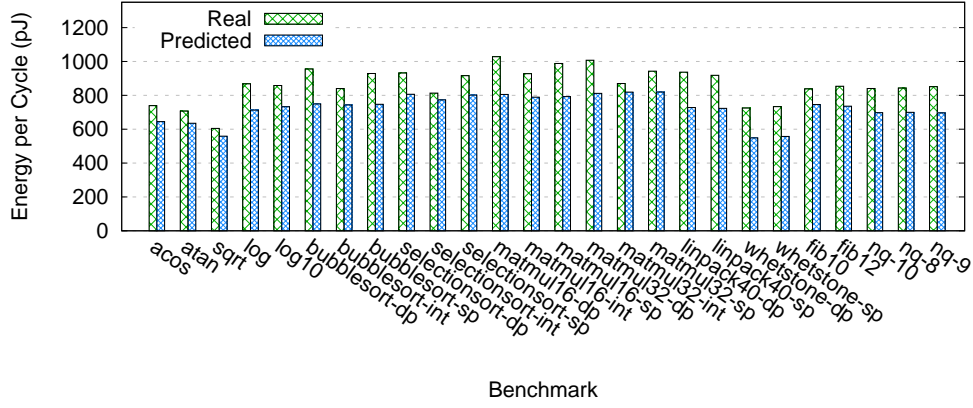
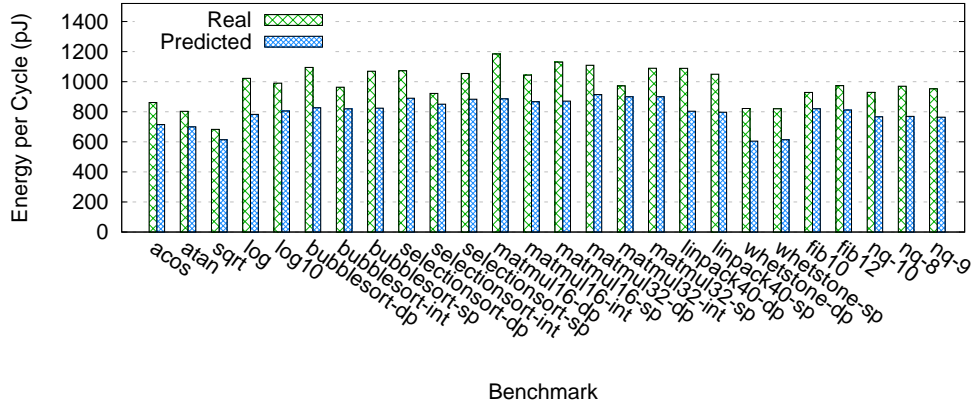


Figure 9.31: Cortex-A15 at 1000MHz: Real VS Model Prediction EPC





**Figure 9.35:** Cortex-A15 at 1400MHz: Real VS Model Prediction EPC



**Figure 9.36:** Cortex-A15 at 1500MHz: Real VS Model Prediction EPC

Our model shows an average deviation from the actual energy consumption in the order of 8.5% for Cortex-A7 cores and 14% for Cortex-A15 cores. The minimum, maximum and average error for Cortex-A7 cores are shown in figure 9.37, and for Cortex-A15 in figure 9.38.

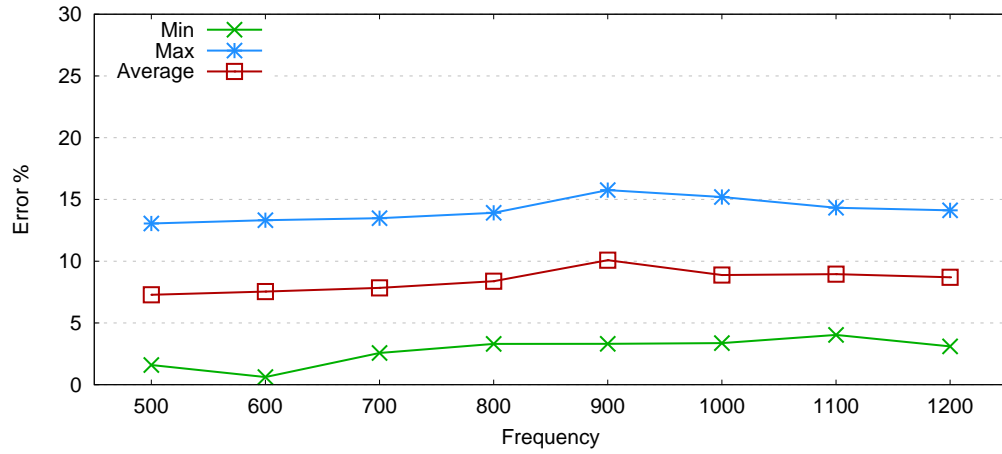


Figure 9.37: Cortex-A15: Model Prediction Error %

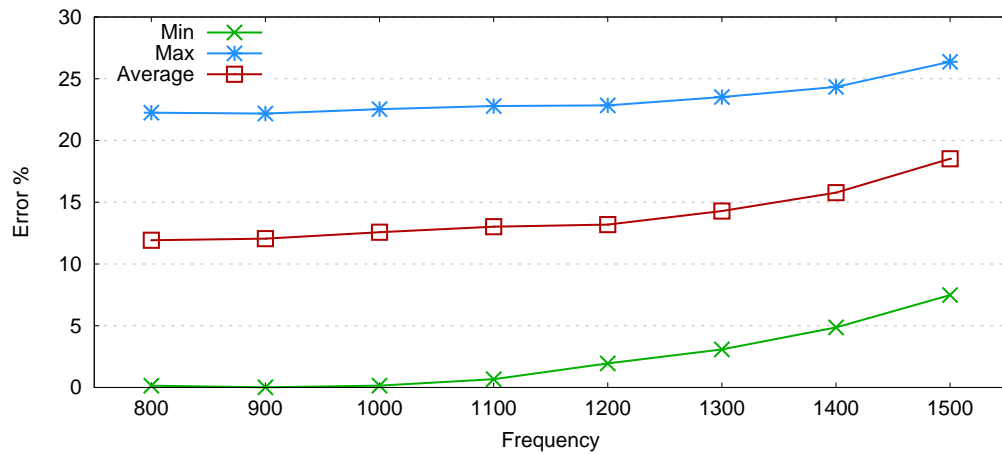
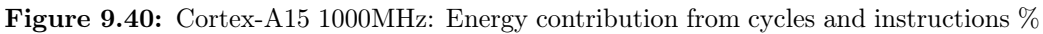
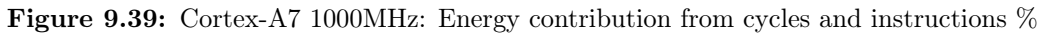


Figure 9.38: Cortex-A15: Model Prediction Error %

### 9.3.1 Discussion

#### Energy Contributions

Our model is based on a simple approach that separates the energy consumption in two basic factors. The first is the number of cycles a sequence of instructions takes to run, and the second is the energy contribution of every instruction executed within that number of cycles. Although our model can have a percentage of misprediction, it is important that we analyze the contribution of each of these two factors (cycles and instructions) to see if any one of these dominates the energy consumption and also to examine if the error rate is correlated with any one of them.



So, in conclusion, energy contribution from instructions is of less significance at the *little* than at the *big* cores. At the same time the importance of instructions contribution to energy is more pronounced at the *little* cores in relation to the *big* ones.

Technical Report FORTH-ICS/TR-450 75

predicted energy along with the error for both core types at the same frequency of 1000 MHz. These are characteristic of the results across the entire frequency range. We can see from the figures that the error percentage is not correlated with either of the factors of our model. Bellow we explain some reasons for the mis-predictions of our model.

## Errors

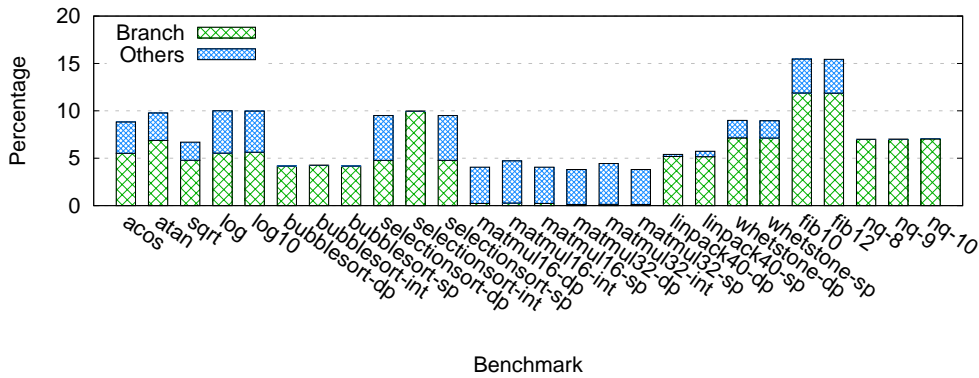
The energy model can mis-predict the energy results for a few reasons, the most important ones are:

### Branch and other instructions:

As we explain in section 6.2 branches are absent from our characterization simply because there is no straightforward way to model their energy consumption as with the other instructions we characterized. Furthermore, although our study is extensive it is not complete as to the myriads of instruction variations there are in the ARM instruction set and some instructions would have to be left uncategorized. For branches and other instructions we simply used the average instruction cost of all other instructions in the energy model. The percentage of branches and other instructions can vary from 4% to 15%, their percentage for each benchmark can be shown in figure 9.41.

**Instruction flavors:** as we describe in section 6.2, although we characterize efficiently most of the basic instruction categories in the ARM instruction set (except branches), some instruction flavors like shifted arithmetic and multiple load/store instructions had to be grouped with others for the sake of simplicity.

**Datapath control circuits complexity:** The processor datapath does not consist only of ALU and other functional units, a great and more complex part of the datapath is the control circuitry. Our synthetic benchmarks do not stress these circuits enough since the only execute one type of instruction.



**Figure 9.41:** Executed Instructions Breakdown %

## Chapter 10

# big.LITTLE Comparison

In this chapter we use our model to evaluate and compare the energy characteristics of the two processors in our study. As we described in 5.2, Cortex-A7 and Cortex-A15 cores are designed with different goals in mind. Cortex-A7 cores trade performance with energy efficiency while Cortex-A15 cores do the opposite and trade energy efficiency for performance. The processors have different datapath lengths, different numbers of execution units and different scheduling mechanisms.

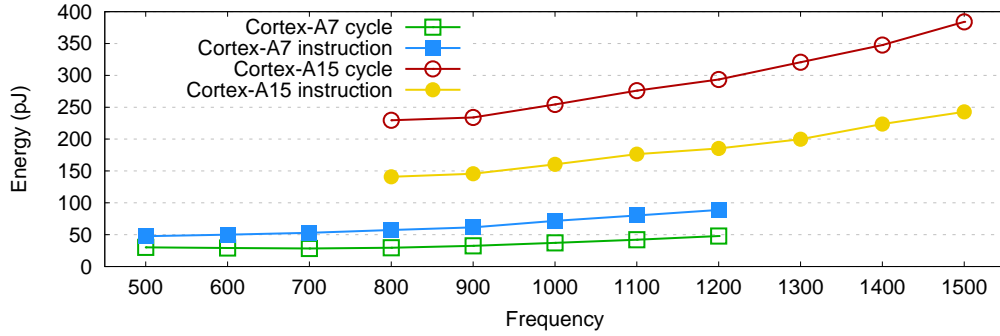
The differences in energy consumption are obvious when comparing the energy cost of the same instructions, as we have shown in chapter 7, even when the processors operate at the same frequency and achieve the same throughput for a specific instruction, the *big* cores can consume anywhere between three to five times more energy per instruction than the *little* ones. This comparison however is not specific enough to help us understand when it is better to use one processor and when to use the other.

The formulation of our model can help with this comparison and guide such decisions. Since we have separated the total energy cost of a program in two distinct factors, we can compare these two factors for both processors and draw more solid and meaningful conclusions.

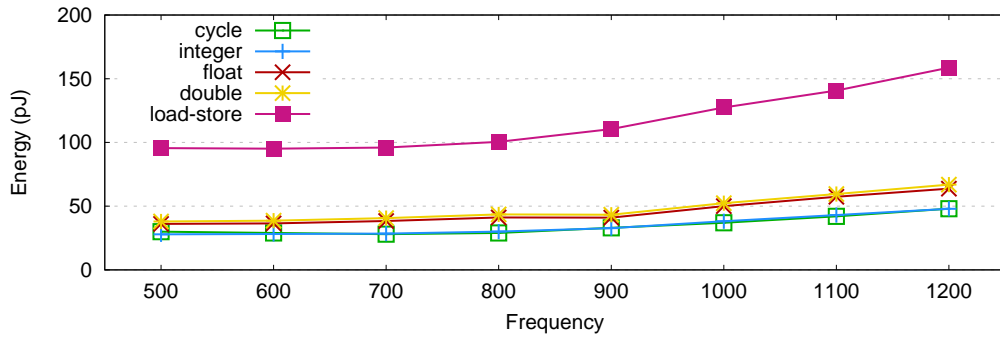
In 10.1 we show the cost per cycle of our model for both cores at the entire frequency range along with the average energy contribution for each instruction executed. It is clear that Cortex-A7 cores are more energy efficient cores than Cortex-A15 and that Cortex-A15 cores offer better performance. However, if absolute energy savings are not the only concern and other metric like Energy Delay Product (EDP) or Energy *Delay*<sup>2</sup> Product ( $ED^2P$ ) is the optimization target, things are not so clear. Our energy model can provide a useful insight towards this dilemma. If we examine the relative costs for every processor type, we can see that at Cortex-A7 cores, the cost per cycle is lower than the average cost for every instruction that is executed. In Cortex-A15 however, the opposite happens, the cost per cycle is bigger than the cost of each additional instruction.

This means that Cortex-A7 cores can afford to be underutilized, since the energy cost for every cycle that passes is lower than that for every instruction executed.

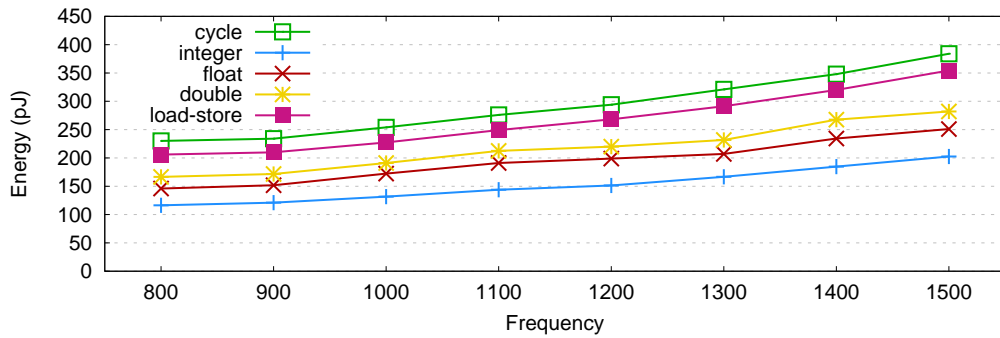
At Cortex-A15 cores, the opposite happens, the cost of an underutilized datapath is much greater relative to the additional cost of an instruction.



**Figure 10.1:** Cortex-A7 and Cortex-A15 energy factors comparison



**Figure 10.2:** Cortex-A7 energy factors comparison



**Figure 10.3:** Cortex-A15 energy factors comparison

The above remarks point to the fact that code with heavy inter-instruction dependencies (*low ILP*) and long latencies due to the memory access pattern, is

---

more efficiently executed at Cortex-A7 cores where the cost of stalling is lower than the cost of instruction execution.

A closer look however reveals more differences between instructions, in figures 10.2 and 10.3 we can see the relative cost between instruction categories for Cortex-A7 and Cortex-A15. These categories are integer, float, double and load/store instructions. The general rule is that integer instructions cost less than float, float cost less than double and loads/stores are more expensive than all. At Cortex-A7 loads/stores cost much more than all arithmetic instructions and almost three times the basic cost per cycle. At Cortex-A15 the differences between instructions are more visible and the differences are not as wide as in Cortex-A7.

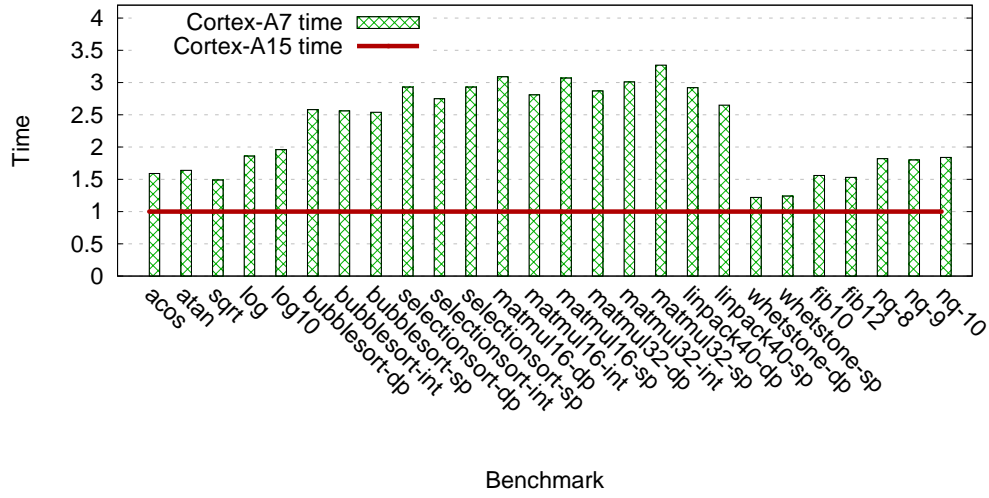
The general conclusion is that if the target code offers high ILP and few stalls due to memory it is more efficiently executed at Cortex-A15 cores. Figures 10.4 and 10.5 show the relative run-time and energy costs of our evaluation benchmarks for Cortex-A7 and Cortex-A15 at the same frequency of 1000 MHz.

For runtime, all values are normalized to the Cortex-A15 runtime since at these cores the benchmarks always achieve higher performance and therefore lower runtime. For energy, the values are always normalized to the Cortex-A7 cores since they always consume less energy there.

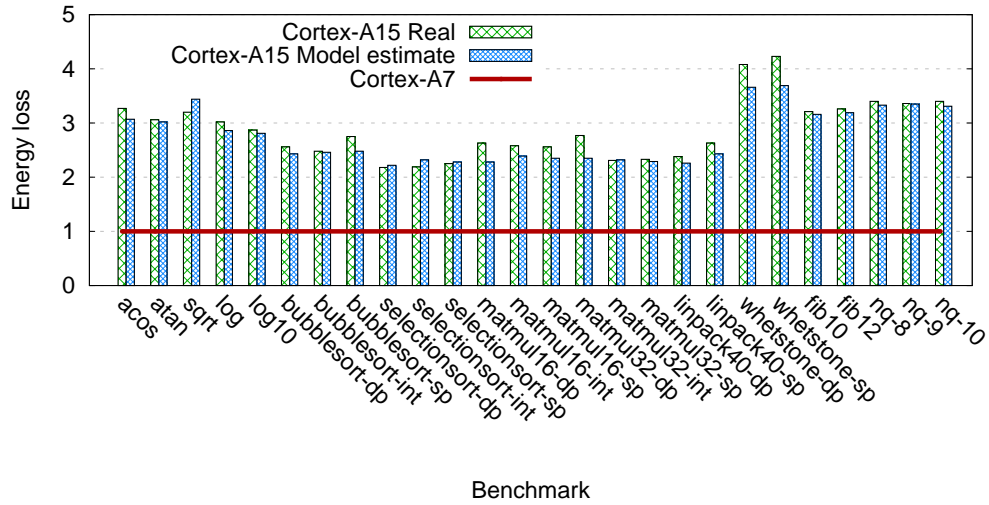
We see that runtimes at Cortex-A7 can be from 1.2 to 3.2 times the corresponding at Cortex-A15. Energy consumption for Cortex-A15 can be from 2.4 to 4.2 times that of Cortex-A7.

When examining the two graphs together, it is visible that they complement each other. In a way, this means that, the lowest the relative gap in performance between the two cores, the highest the energy difference will be. For example, the fibonacci benchmark is 1.5 times slower at Cortex-A7 but 3.4 times more energy hungry in Cortex-A15. Matmul, on the other hand is 3.2 times slower at Cortex-A7 and 2.4 times more energy hungry at Cortex-A15.

This verifies our earlier point that, when considering the energy performance trade-off and not solely energy or delay (runtime), benchmarks with high ILP like *Matmul* are more efficiently run at Cortex-A15 cores while benchmarks like *Fibonacci* that achieve low ILP are more efficiently run at Cortex-A7 cores. To put it simply, when running *Fibonacci* you can get 1.5 times more performance for 3.4 times more energy if you run at Cortex-A15 instead of Cortex-A7 at the same frequency. For *Matmul* one can get 3.2 times the performance for 2.4 times the energy for the same migration, a clearly better trade-off.

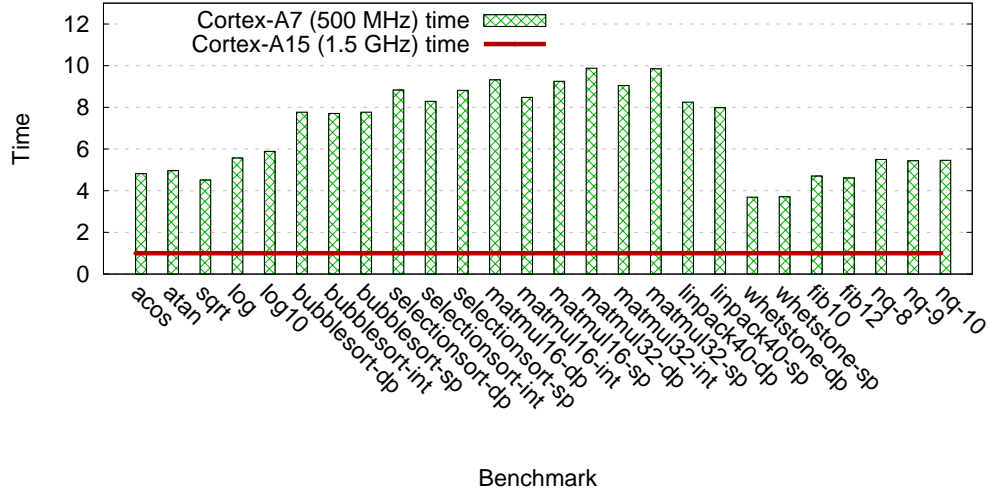


**Figure 10.4:** Execution time from Cortex-A7 to Cortex-A15 (1000MHz)

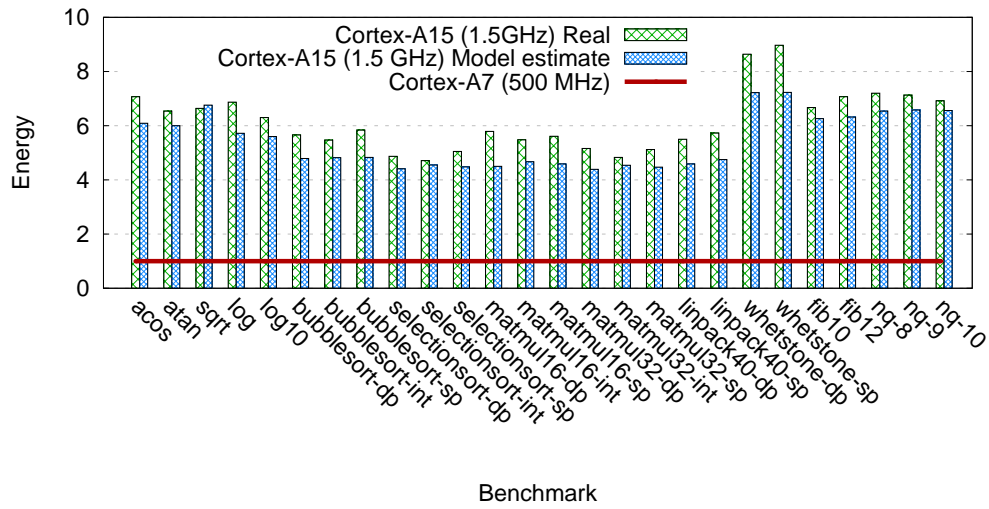


**Figure 10.5:** Energy from Cortex-A7 to Cortex-A15 (1000MHz)

The remarks above apply when comparing the two core types at the same frequency. Another interesting comparison is between the two extremes in terms of performance and energy consumption. When comparing Cortex-A7 at the lowest frequency of 500 MHz (lowest energy consumption) with Cortex-A15 at 1.5 GHz (highest performance) we can see that the energy consumption to completion is five to nine times higher at Cortex-A15 whereas the execution time is increased four to ten times at Cortex-A7. These measurements can be seen at Figures 10.6 and 10.7.



**Figure 10.6:** Execution time from Cortex-A7 (500 MHz) to Cortex-A15 (1500MHz)



**Figure 10.7:** Energy from Cortex-A7 (500 MHz) to Cortex-A15 (1500MHz)

## Chapter 11

# Conclusions and Future Work

This work has shown an instruction level energy characterization of ARM Cortex-A7 and Cortex-A15 processors.

For our characterization we studied the ARM instruction set in depth and separated the instructions into categories of similar semantics. We then developed two special purpose benchmarks for every instruction type taking into account the most common variations and operand types. One benchmark studies the CPI and energy characteristics of instruction when there are dependencies and the other when there are not. This allows us to study the effects of datapath underutilization on instruction energy.

We run our benchmarks on an *Odroid xu+e* development board that features two different ARM processor in a heterogeneous multiprocessing (*big.LITTLE*) configuration. We extrapolated the cost of different instructions in clock cycles (CPI) and energy per instruction (EPI).

We conducted careful and detailed measurements of the time and energy requirements of our benchmarks for both core types and a total of 16 different frequency settings.

Through this characterization we developed a simple linear energy model that takes into account two factors for estimating the energy consumption of a program.

The first is a basic cost for every cycle of program execution, and the second is an additional cost for every instruction that is executed within those cycles.

We tested the validity of our characterization and energy model on real applications like *matmul*, *linpack*, *whetstone* and *fibonacci* benchmarks and achieved an average mis-estimation of 8.5% for Cortex-A7 and 14% for Cortex-A15 cores.

Through our energy model we shed some light to why underutilized cores are not energy efficient, as well as some useful insight into heterogeneous architectures like ARM *big.LITTLE* and how to allocate computing resources more efficiently based on workload traits.

Using some of our earlier work on X86 instruction energy, we quantitatively compare the ARM and Intel processors in terms of energy per instruction and show some of the differences between the two architectures.

---

Our future plans include optimizing that energy model by incorporating memory energy consumption and correlating it with processor energy. Furthermore and interesting outlook would be an even deeper look into the ARM instruction set and proposing architectural energy and latency optimizations for future processors.

## Appendix A

# Using Architectural Counters to Evaluate the Cost of Instructions in x86 Architectures

### A.1 Introduction

This is some earlier work of the author on X86 architectures. We use the same methodology to study the energy and performance characteristics of integer arithmetic and logical instructions, scalar and packed floating point, data movements between registers, loads and stores to different levels of the cache hierarchy and more. For this work our measurements are made via the architectural counters provided by intel processors and not by actual energy measurements through dedicated sensors. We present the design of those benchmarks along with their results for a case-study Intel *Sandy Bridge* architecture CPU. While presenting our findings we share some characteristics of the CPU that we were able to infer.

### A.2 Methodology

Processors consume energy to execute instructions, but they also do so when idle. Our work relies on the energy readings provided by the Machine Specific Registers (MSRs) provided by the latest generations of Intel processors [55]. The provided measurements however, do not distinguish between static and dynamic energy. We overcome this by reporting on the mixed static and dynamic energy consumption and by running our benchmarks on all cores at the same time. That way, the static power is the lowest possible compared to dynamic power.

The registers/counters used are the Power Plane 0 (PP0) counter, which includes the energy of the processing cores and the caches, and the package energy counter which provides the energy of the entire processor chip package. The findings that we report are based on the PP0 counter, but we also measured the package energy mainly to see how they are correlated. Also available in Intel processors

is a Time Stamp Counter (TSC) that we can use to measure time in clock cycle granularity.

We have used the above mentioned energy and time counters in order to develop an energy model for the basic instructions and operations of the x86-64 instruction set.

### A.2.1 Benchmark Specification

We have designed a series of benchmarks that target specific instructions and measure their time and energy cost. We do that by running a number of instructions written in assembly, in a highly unrolled loop also written in assembly. The body of the loop consists of 1000 instructions for arithmetic benchmarks and 1024 instructions for memory and cache benchmarks in order to keep memory footprints a power of 2. The code for the loop itself is merely 2 instructions per iteration. This is small enough to attribute the energy and time cost entirely to the instructions in the loop body. Listing A.1 shows the loop code with the body of the loop and *rcx* register initialization omitted. In all iterations of the loop except the last, only one decrement “*dec %rcx*” and one jump “*jz .LOOP\_END*” are executed, this is 0.2% of the executed instructions.

```

1 .LOOP_START:
2 1:      jz      .LOOP_END
3
4      \\loop body
5
6 1001:    dec     %rcx
7 1002:    jmp     .LOOP_START
8 .LOOP_END:

```

**Listing A.1:** Loop code

The measurements of the energy and time counters are taken after all initializations are finished and as close as possible to before the start and after the end of the loop. After the measurements are taken, we calculate an energy and time cost for a single type of instruction. This is possible, since the measured quantities are attributable to a certain number of instructions with little pollution from other parts of the benchmark. Running similar benchmarks for different types of instructions allows us to compare the energy and time cost of different instructions.

We also study the same instruction types under different conditions, e.g. memory loads from different parts of the memory hierarchy, series of instruction with no dependencies between them or with dependencies purposefully introduced. Instructions with no dependencies allow us to study the maximum throughput of the processor in terms of instructions per cycle, while dependent series of instructions show us the relative latency of instructions and the subsequent effect on energy. Furthermore, both of these scenarios are encountered in applications, as there are parts where instructions form a sequence where the results of the previous instruction are needed by the next, and there are scenarios where many instructions are

independent of each other. In listing A.2 we can see an example of dependent instructions: The `addq` instruction in line 1 writes to register `r9` and the following instruction reads it and writes to register `r10` from which the next `addq` reads and so on. Listing A.3 shows an example of independent instructions; as we can see there is no register shared between them.

```

1:      addq      %r8,    %r9
2:      addq      %r9,    %r10
3:      addq      %10,    %r11

```

**Listing A.2:** Dependent instructions

```

1 1:      addq      %r8,    %r9
  2:      addq      %r10,   %r11
3 3:      addq      %12,    %r13

```

**Listing A.3:** Independent instructions

There is no direct interaction between the benchmarks on different cores for arithmetic benchmarks. For the memory related benchmarks there is some interaction at the shared L3 cache and memory controller whenever the memory footprint of the benchmarks exceeds the local L1 and L2 cache capacity.

The metrics we use are Energy per Instruction (EPI), Energy Per Cycle (EPC), Cycles per Instruction (CPI) and its reverse Instructions per Cycle (IPC) to facilitate understanding for the reader.

### A.2.2 Experimental Setup

All the benchmarks are run on a Intel® Core™i5-2500 Sandy Bridge CPU running at 3.30 GHz. The processor has 4 cores, 32 KB of private L1 Cache, 256 KB of private L2 cache and 6 MB of shared L3 cache. The clock rate has been fixed through the BIOS to 3.30 GHz to prevent any interference from dynamic frequency scaling.

The benchmarks are run in 1 to 4 cores each time, all threads are pinned to a specific core at creation and are not allowed to migrate.

## A.3 Benchmark Description and Results

The X86-64 Instruction set consists of hundreds of instructions, each of which can have a varying number and types of arguments. Trying to figure out a time and energy cost for all instructions would require a great deal of effort. In this work we have grouped instructions with similar semantics and characteristics.

In most programs some instruction types dominate the execution, in this work we have separated instructions into what we think are the dominant categories and have developed benchmarks to study their time and energy cost. These categories are integer arithmetic and logical instructions which are present in all programs, floating point arithmetic instructions both scalar and packed as they are dominant

Instruction	CPI	IPC	EPI (nJ)	EPC (nJ)
<b>addq</b>	0.349	2.81	1.115	3.192
<b>subq</b>	0.332	3.01	1.063	3.200
<b>imulq</b>	1.039	0.96	2.782	2.667
<b>divq</b>	29.607	0.034	93.223	3.149
<b>idivq</b>	39.309	0.025	132.244	3.364
<b>andq</b>	0.349	2.86	1.112	3.185
<b>orq</b>	0.355	2.814	1.120	3.151

**Table A.1:** Integer arithmetic and logic instruction results with no dependencies

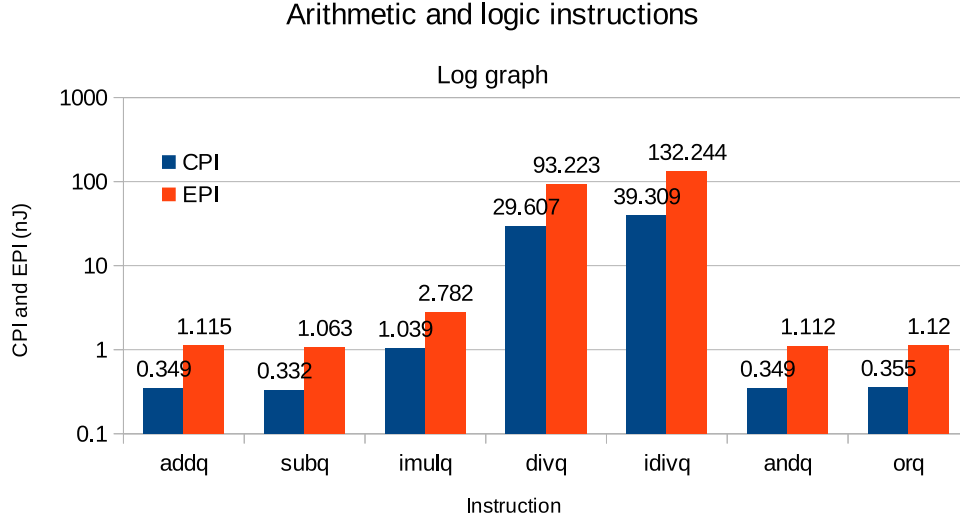
in some types of programs. Furthermore we have studied data movement instructions, both between registers in each core as well as between the cache and memory hierarchy and the processor cores. We have also studied the energy behaviour of nops as we think they can provide valuable insight on the CPU characteristics. Additionally we have tried to evaluate the cost of false sharing between cores. False sharing happens when different cores write different variables located on the same cache line.

### Integer Arithmetic and Logic

Integer arithmetic and logic instructions are in the heart of every program, a good estimate of their cost cannot be omitted from this study. We have evaluated the cost of 64 bit addition, subtraction, multiplication, division and logical *and* and *or* instructions between registers. We have designed benchmarks where the instruction operands are randomly chosen from the set of general purpose x86-64 registers. We have been careful in order to eliminate all dependencies between neighbouring instructions as much as possible.

Table A.1 shows the cost of the different arithmetic instructions. The first column shows the opcode of the instructions, the second is the cycles per instruction metric (CPI), next is the energy per instruction in nJoules EPI (nJ) and on the last column the energy per cycle of one core is shown when executing only that kind of instructions. The fourth column of table A.1 shows that the energy cost of arithmetic and logical instructions varies from approximately 1.1 nJ for additions, subtractions, and logic operations to 132 nJ for signed divisions.

There is also a clear correlation between the energy cost of an instruction and the clock cycles it takes. This trend is visible when contrasting the second and fourth column (CPI and EPI) of table A.1. However, we can also see that the energy per cycle (EPC) metric fluctuates from 2.667 nJ for multiply instructions to 3.364 nJ for unsigned division. This means that the cost of an instruction is not always a constant function of the clock cycles it takes. For example, a signed



**Figure A.1:** Arithmetic and logic Instruction results

Instruction	CPI	IPC	EPI (nJ)	EPC (nJ)
addq	0.336	2.98	1.075	3.205
subq	0.34	2.943	1.082	3.184
imulq	1.024	0.976	2.762	2.697
andq	0.336	2.978	1.072	3.194
orq	0.336	2.976	1.078	3.209

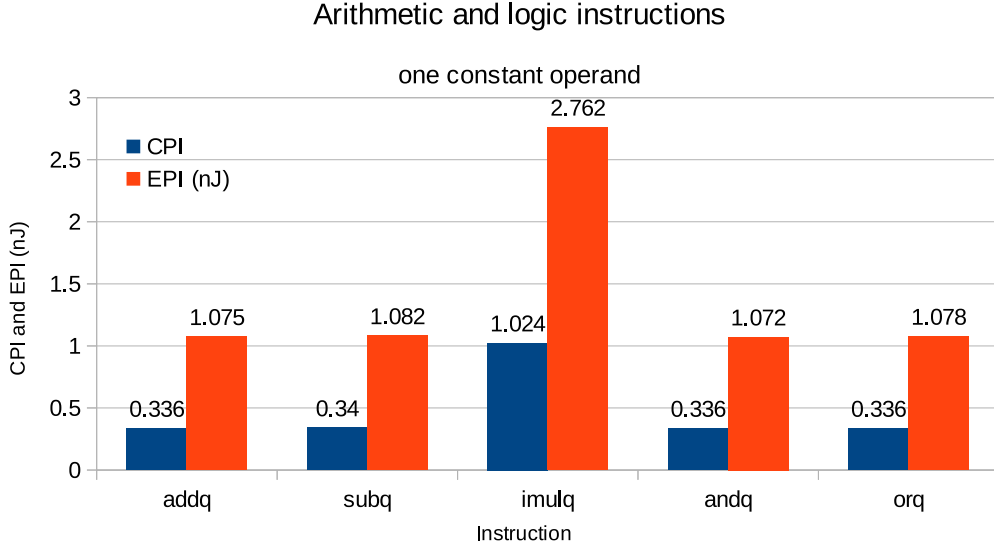
**Table A.2:** Integer arithmetic and logic instruction results with no dependencies and a constant operand

multiplication takes 3 times the time of a subtraction, but it consumes only 2.5 times the energy.

The relative cost between the different arithmetic and logical instructions can be visualized in the logarithmic graph A.1. On the  $X$  axis we have the different types of instructions, the left column for each instruction represents the time in cycles per instruction (CPI) and the right column the energy per instruction (EPI) in nano Joules.

In addition to the arithmetic instructions between registers, we have carried out some experiments with a constant value as one of the operands. The results are shown in table A.2 and visually in graph A.2. From these we can see that the fact that one operand is a constant value instead of a register plays no role in the energy consumption of an instruction.

Furthermore, we have created benchmarks where the instructions are depen-



**Figure A.2:** Arithmetic and logic Instruction results with one constant operand

Instruction	CPI	IPC	EPI (nJ)	EPC (nJ)
addq	1.001	0.999	2.725	2.723
subq	1.000	1.000	2.719	2.719
imulq	3.001	0.333	7.611	2.536
andq	1.005	0.995	2.718	2.705
orq	1.032	0.969	2.750	2.665

**Table A.3:** Integer arithmetic and logic instruction results with dependencies

dent, this means that each instruction depends on the outcome of the previous. This scenario, although far fetched, is not very unrealistic as it is common to have pieces of code where each instruction computes on the results of the previous.

Table A.3 shows that due to the dependencies between the instructions, the CPI triples for all benchmarks, increasing to 1 cycle per instruction for additions, subtraction and logical instructions and to 3 cycles per instruction for multiplications. This is because, when instructions are dependent the next instruction cannot reach the execution stage unless the previous has produced its result. In dependent instructions the energy per instruction is once again correlated positively, but sub linearly, with execution time. In additions, for example, the energy per instruction (EPI) is 2.725 nJ for dependent instructions where only 1 is executed per cycle and 1.115 nJ for independent instructions where 3 are executed per cycle.

From this we can see that underutilization of the CPU is not energy efficient. The difference between executing 3 instructions and executing only 1 is less than

20%, Ideally, the energy would be proportional to the IPC metric if we assume no static energy consumption.

### Floating Point Arithmetic

We have implemented our floating point computation benchmarks using the SSE extensions of the x86 architecture, we have studied both scalar and packed arithmetic instructions on the *XMM* registers for addition, subtraction and multiplication operations. All *XMM* registers are 128 bits wide and can be used either for scalar operations of 32 or 64 bits (the rest of the bits are just propagated along with the computation result to the destination register) or as packed instructions of  $4 \times 32$  bits single precision floating point values or  $2 \times 64$  bits double precision floating point values.

Similarly to the integer benchmarks, we have studied floating point instructions independent of each other and instructions dependent on the result of their previous instruction.

Table A.4 shows the results for all four different cases of independent scalar and packed instructions for single and double precision operands. They are separated into four distinct categories shown one below the other. The first is single precision (32 bit) scalar instructions, this means that only one 32 bit floating point operation takes place between the two 128 bit registers. Double precision scalar instructions are the same as single precision but for 64 bits. The last two categories involve packed operations on all 128 bits of the registers, either as  $4 \times 32$  bits (single precision) or  $2 \times 64$  bits (double precision). The columns of the table follow the same pattern as in previous tables, the first column is the instruction type the next two are cycles per instruction (CPI) and its reverse instructions per cycle (IPC), while the last two are energy per instruction (EPI) and energy per cycle (EPC) in nano Joules.

Observe that there is no difference in the energy consumption between single or double precision operands or even between packed and scalar operations. Furthermore, we can see that the floating point instructions cost approximately 3 times as much as their integer counterparts for additions and subtractions in both time and energy.<sup>1</sup>

The correlation between the IPC and EPI metrics is still visible in floating point as it is for integer arithmetic instructions. The energy per cycle (EPC) metric for floating point is generally about 20% lower than that of integer instructions.

In par with the methodology for integer instructions, we studied the behaviour of floating point instructions with dependencies introduced. Table A.5 shows the same measurements for instructions with dependencies.

For dependent, as well as independent instructions, there is no difference in energy or time between single and double precision or between packed and scalar variants of the instructions.

---

<sup>1</sup>We cannot directly compare multiplications because the floating point instruction is unsigned while the integer multiplication we studied is signed.

Instruction	CPI	IPC	EPI (nJ)	EPC (nJ)
Single precision scalar				
<b>addss</b>	1.113	0.898	3.190	2.865
<b>subss</b>	1.113	0.898	3.193	2.869
<b>mulss</b>	1.664	0.601	4.523	2.718
Double precision scalar				
<b>addsd</b>	1.125	0.889	3.205	2.849
<b>subsd</b>	1.113	0.899	3.192	2.868
<b>mulsd</b>	1.652	0.605	4.499	2.723
Single precision packed				
<b>addps</b>	1.109	0.902	3.185	2.873
<b>subps</b>	1.114	0.897	3.197	2.869
<b>mulps</b>	1.644	0.608	4.496	2.734
Double precision packed				
<b>addpd</b>	1.110	0.901	3.190	2.875
<b>subpd</b>	1.107	0.903	3.186	2.877
<b>mulpd</b>	1.664	0.601	4.511	2.712

**Table A.4:** FP arithmetic instructions with no dependencies

The second column of table A.5 shows that for additions and subtractions the CPI almost triples from approximately 1.1 instruction per cycle when the instructions are independent, to slightly over 3 instructions per cycle when there are dependencies. The same applies to multiplications where it increases from 1.6 to 5.1 instructions per cycle. This reveals the issue width for independent SSE floating point instructions and also the latency cycles for a floating point instruction execution, both of which seem to be 3.

### Data Movement Between Registers

Data movement between registers is also very important, both for integer and floating point registers. In our benchmarks we have designed and measured five different data movement schemes between registers. These come from moving either 64 bit general purpose registers or 128 bit XMM registers with or without dependencies between the instructions. Each move instruction has a source and destination register, when executed the source is copied to the destination register. The dependencies come in a similar way as in the computation instructions but now the destination register is treated as the result register. The fifth configuration

Appendix A. Using Architectural Counters to Evaluate the Cost of Instructions in x86 Architectures

---

Instruction	CPI	IPC	EPI (nJ)	EPC (nJ)
Single precision scalar				
<b>addss</b>	3.055	0.327	7.912	2.589
<b>subss</b>	3.047	0.328	7.903	2.593
<b>mulss</b>	5.024	0.199	12.752	2.538
Double precision scalar				
<b>addsd</b>	3.037	0.329	7.867	2.591
<b>subsd</b>	3.014	0.332	7.829	2.597
<b>mulsd</b>	5.022	0.199	12.748	2.538
Single precision packed				
<b>addps</b>	3.038	0.329	7.868	2.590
<b>subps</b>	3.068	0.326	7.885	2.570
<b>mulps</b>	5.028	0.199	12.733	2.532
Double precision packed				
<b>addpd</b>	3.032	0.330	7.848	2.588
<b>subpd</b>	3.064	0.326	7.878	2.571
<b>mulpd</b>	5.102	0.196	12.836	2.516

**Table A.5:** FP arithmetic instructions with dependencies

Instruction	CPI	IPC	EPI (nJ)	EPC (nJ)
<b>movq</b>	0.336	2.978	1.076	3.206
<b>movq (const)</b>	0.336	2.978	1.073	3.196
<b>movq (dep)</b>	0.998	1.002	2.704	2.710

comes from moving constants to general purpose register, this is not possible for XMM registers which can only be loaded from memory.

Table A.7 and table A.6 show the results from the measurements for general purpose registers and XMM registers accordingly. The columns from left to right show the type of instruction, cycles per instruction (CPI), instructions per cycle (IPC), energy per instruction (EPI) and energy per cycle (EPC). There is a series of remarks that needs to be made on the results of moving data between registers.

First, from the first 2 rows of table A.7 we see that there is no significant difference in time or energy cost when moving constant values to general purpose registers and moving data between registers, this is to be expected from the same behaviour we noticed at arithmetic instructions.

**Table A.6:** 128 bit register movements

Instruction	CPI	IPC	EPI (nJ)	EPC (nJ)
<b>movdqu</b>	0.346	2.889	1.108	3.200
<b>movdqu (dep)</b>	1.019	0.981	2.739	2.687

**Table A.7:** 64 bit register movements

Second, when comparing the first 2 rows of table A.7 with the third we can see that dependencies have the same effect on energy as well as on time as they do in arithmetic instructions. This applies to both 64 bit and 128 bit registers the same.

Third, and this is also to be expected, moving 64 bit registers takes almost the same time as moving 128 bit registers. Apparently, there is no reason why this would differ because data movements are usually done in parallel.

Fourth and final remark on these results, and probably the most surprising one, is that the energy cost of 64 and 128 bit data movements is hardly distinguishable. According to the measurements, it costs 1.076 nJ to move data between two 64 bit registers and 1.108 nJ to move data between two 128 bit registers. This can be seen by contrasting the EPI columns of the first rows of table A.7 and table A.6.

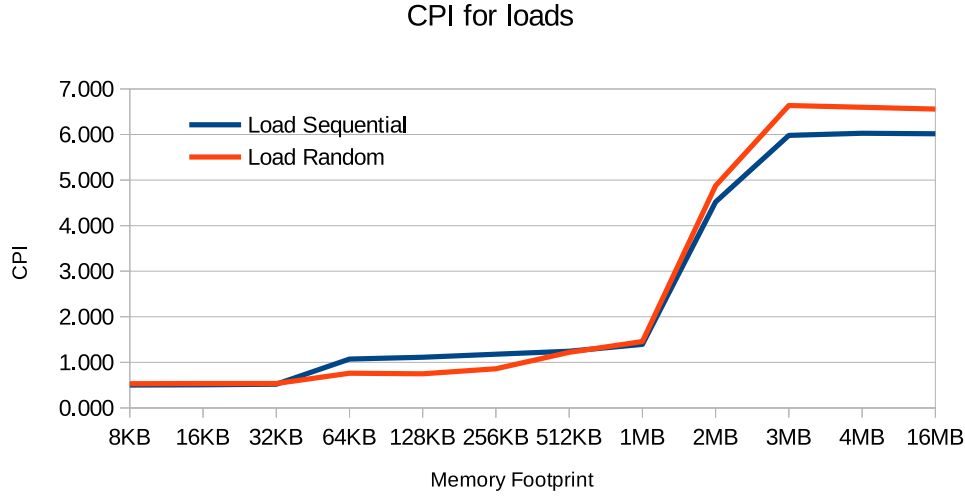
### Loads/Stores

Loads and stores are used to bring data back and forth from the processor to the cache hierarchy and main memory; understanding and estimating their cost is vital to evaluating any program. The cost of any load or store operation cannot be determined on its own, how many clock cycles a load instruction will take depends on where the data is in the cache and memory hierarchy. A store also depends on many factors like cache policies, write buffer configurations and more.

To study these types of instructions we have designed a series of benchmarks that intend to show how the time and energy metrics for these instructions can vary, depending on the data locality.

These benchmarks are designed to have a fixed memory footprint, in each iteration of the main loop, as also described in the methodology section, 1024 instructions are executed that are either loads or stores of 8-byte words. These 8 KBytes of data can be read either sequentially or randomly in a statically set sequence. When the iteration finishes, the base address is incremented by 8 KBytes and the next loop iteration reads or writes the same amount of data. This continues until the maximum memory footprint is reached for the benchmark and then the base address is reset to its first value. The loop continues for a statically set number of iterations that is the same for all benchmarks.

The memory footprints we have measured are exponentially growing from 8 KB to 16 KB to 32 KB and so on until 16MB are reached. It is important to remind the characteristics of the cache hierarchy of our experimental setup. Each core has



**Figure A.3:** Random and Sequential load Instructions CPI

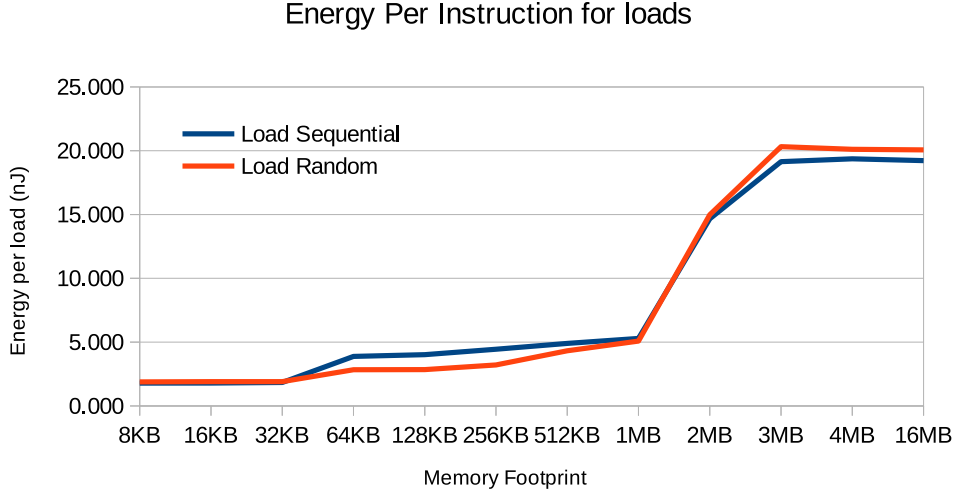
32 KB of private L1 cache, 256 KB of private L2 and 6 MB of shared L3 cache.

All the load instructions write data to random general purpose registers, the store instructions can, depending on the benchmark, either write the value of a register to memory, or write a constant.

Graph A.3 shows the difference between loading sequentially and randomly and the overall behaviour of load instructions' execution time as the memory footprint increases. On the X axis are the different memory footprints starting from 8 KB and doubling at each step until 16 MB. On the Y axis are the cycles per instruction. We can see that, as the memory footprint increases, so does the CPI as we have to travel deeper into the cache hierarchy and the main memory. However, for memory footprints of 64 KB to 256 KB loading random addresses is up to 35% faster compared to sequential loads. We speculate that this could be attributed to some cache policy optimized for random requests.

Another point to be made is that the performance starts to deteriorate after the footprint surpasses the size of the L1 cache (32 KB). However, there is no similar behaviour at 256 KB, which is the L2 capacity, instead there is a dramatic increase after the memory footprint reaches 1 MB up until 3 MB where it flat-lines. This could be attributed to effective prefetching to the L2 from the L3 cache.

Graph A.4 shows the same behaviour for energy per instruction. As the memory footprint increases in the X axis, EPI is around 1.8 nJ up to 32 KB and then doubles at 64KB slowly reaching 5 nJ per instruction at 1 MB. Then it drastically increases to 20 nJ per instruction at a memory footprint of 3MB. The same performance benefit for random loads between footprints of 64 KB to 256 KB is evident in energy per instruction shown in graph A.3, as in cycles per instruction shown in



**Figure A.4:** Random and Sequential load Instructions EPI

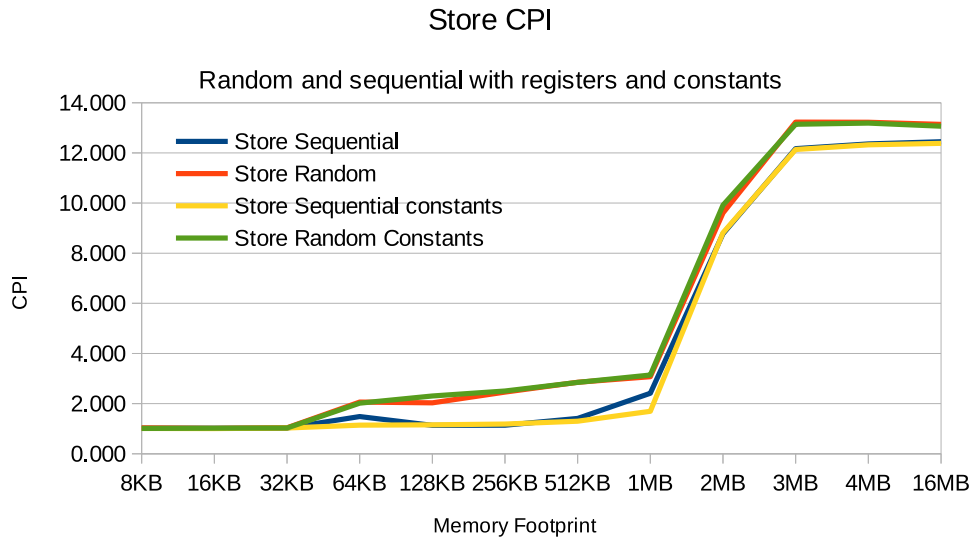
graph A.4.

When studying the store instruction benchmark results, we see a reverse pattern from what we saw for loads. When executing stores, randomness has a noticeable impact on performance. The same is observed whether storing constants or data from registers. The differences are more pronounced when the memory footprint is between 64KB and 1 MB. Graph A.5 shows that randomness can even double the cycles per instruction (CPI) from 1.139 to 2.460 and from 1.409 to 2.852 cycles per instruction for 256 KB and 512 KB footprints respectively.

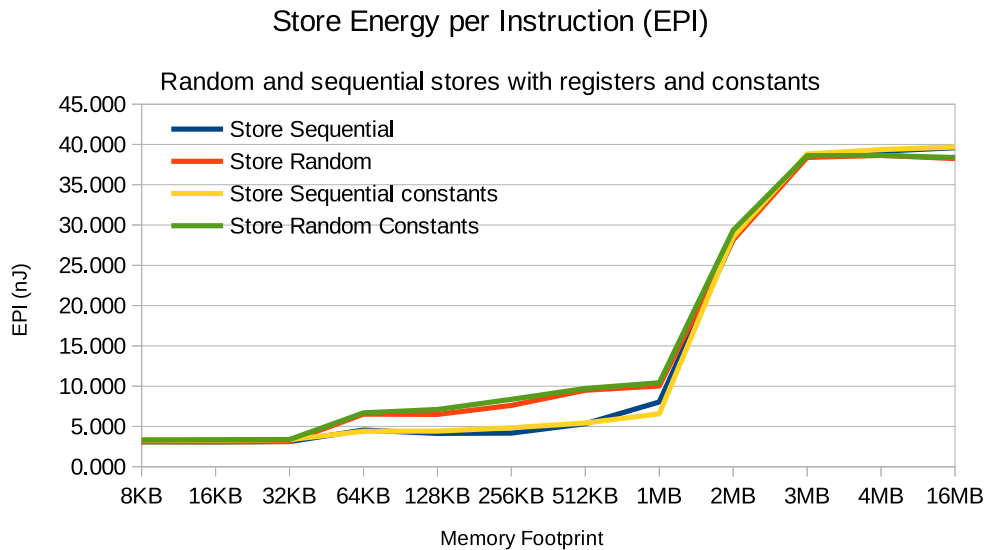
The same behaviour is observed for the energy per instruction (EPI) metric in graph A.6. Sequential stores cost less energy per instruction than random ones when dealing with memory footprint between 64KB and 1MB. This shows that the same strong correlation between CPI and EPI holds in load and store instructions as in the arithmetic ones.

In general, store instructions cost both in cycles and energy, two times more than the corresponding loads. Table A.8 gives a summary of the best and worst costs of each instruction. Each row shows the best and the worst CPI and EPI for loads in the first row, and stores in the second. Another remark on the cost of load and store instructions is their higher energy cost per cycle than arithmetic instructions. Load instructions can cost anywhere from 3 to 4 nJ per instruction, when integer instructions could hardly reach 3.2 nJ. This can of course be attributed to the energy consumption of the cache hierarchy.

Another interesting perspective, is the effect of the number of cores that operate on the cache hierarchy and memory through load and store instructions. The above mentioned measurements come from running our benchmarks on all 4 cores of the



**Figure A.5:** Random and Sequential store Instructions CPI



**Figure A.6:** Random and Sequential store Instructions EPI

system at the same time, Although each benchmark touches a different part of the address space, the shared L3 cache and memory controller have an effect on the performance all cores when there is contention for them. This contention affects only the worst cases, when the operations involve the highest levels of the cache and the memory.

Summarized in table A.9 is the effect of contention on the worst time per load

Instruction	Best CPI	Best EPI (nJ)	Worst CPI	Worst EPI(nJ)
Load	0.502	1.993	6.636	20.319
Store	1.014	3.075	13.222	38.639

**Table A.8:** Best and worst performance for load and store instructions

Instruction	CPI 1 core	CPI 2 cores	CPI 3 cores	CPI 4 cores
Load	2.765	3.880	5.140	6.636
Store	4.601	6.907	9.870	13.222

**Table A.9:** Load and store performance under different contention

and store. Each column shows the CPI for a number of cores, the first row is for loads and the second for stores. We can see that under no contention (1 core), the worst case average load time is 2.765 cycles which escalates to 6.636 cycles under full contention (4 cores). The same is observed for store instructions, from 4.601 at 1 core to 13.222 cycles per instruction at 4 cores.

## Nops

The study of the energy and time cost of nops has no intrinsic benefit because they perform no usefull computation or data transfer. However, their results can be important when correlated with other instructions and also to reveal some characteristics of the CPU.

When running a benchmark consisted entirely of nops, we notice that the CPI is 0.257, or a corresponding 3.887 instructions per cycle, we can easily conclude that is the issue width of the processor is 4. Although the integer instructions were independent and theoretically could be issued with no dependencies an IPC count of more that 3 was never encountered in any benchmark. This is perhaps a clue for the number of integer ALUs in each core. Additionally, when the integer instruction were dependent we saw a CPI of 1. This confirms that the integer ALUs can produce a result in a single cycle and that there are 3 of them.

Similarly, from our results for integer multiplication, we can see that the integer multiply unit takes at most 3 pipeline stages to compute and that there is only one used. Similar conclusions can be reached about the floating point (SSE) datapath when studying the performance of the same type of instructions with or without dependencies.

## False Sharing

At an attempt to quantify the effects of false sharing on energy and time per instruction we have developed corresponding benchmarks. In these benchmarks, we

have all threads incrementing a private variable from 0 to a statically set number. In the false sharing case, the variables for the four threads are located in the same cache line. In the second case, the variables are located in different cache lines. The time and energy measurements of these two benchmarks are not comparable with those we studied before because they do not measure a specific primary instruction. However, comparing their results can give a good estimate on the cost of false sharing.

The false sharing benchmark, was approximately 30% slower and 35% more energy consuming than the other. The increments were not done in lockstep, so it is not safe to say that the difference of incrementing a variable residing in a local cache against fetching it from another cache with the subsequent invalidations is only 30%, but it is worth noting.

### Static Energy

All the measurements mentioned in the previous section include the static and dynamic energy that is consumed by the processor cores and caches. In order to estimate the static energy from the mixed measurement, we ran our benchmark suite on 1 to 4 cores. We did this to measure the difference in energy consumption when a core is idle against the same when it is executing a benchmark.

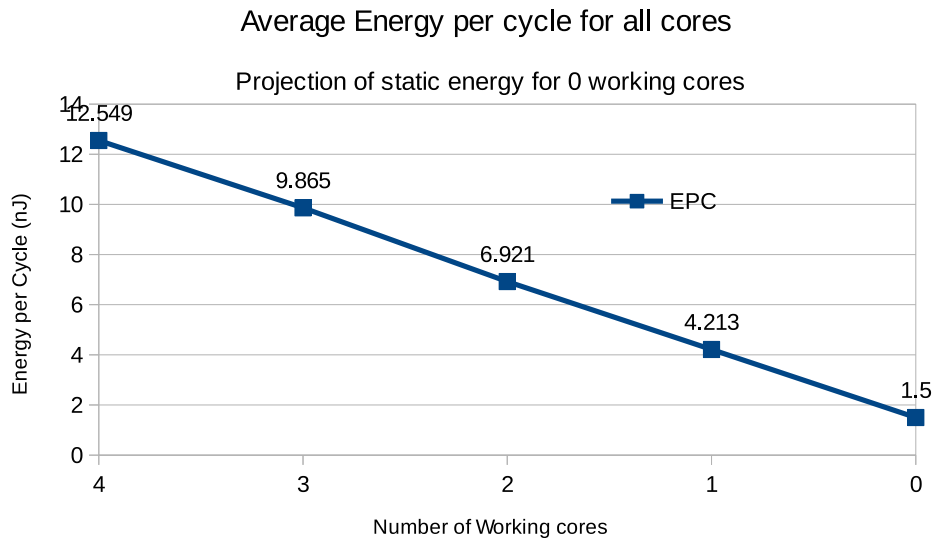
However, this approach does not allow us to measure the energy with all cores being idle. Nevertheless, we chose this because there is no elegant way to wait for some time between measurements without actually executing something, even if that is nops, that we showed have a great impact on energy.

In graph A.7 we show the average over all the benchmarks of the sum of the energy per cycle for 1 to 4 running cores. Also we show a projection to no cores running benchmarks, which also is the estimated static power consumption of all the cores combined. We can see that there is an average 2.7 nJ of increase in power as we add another running core. If we subtract this from the energy per cycle we get when only one core runs, which is 4.2 nJ, we get the total static power of all 4 cores which is approximately 1.5 nJ per cycle. Equivalently, about 0.37 nJ of static energy per cycle per core.

## A.4 Conclusion

In this work we have shown a method for extrapolating the time and energy cost of instructions in the x86-64 instruction set. We have done so by utilizing the provided energy and time counters of the latest generation Intel processors with the use of specialized benchmarks that focus on specific instructions.

We have shown the results from running those benchmarks on a Intel i5 processor and have presented our findings about the cost of basic instructions under different conditions. This has also allowed us to infer a few internal characteristics of that processor and make an estimation on the static energy consumption.



**Figure A.7:** Random and Sequential store Instructions EPI and CPI

We believe that our work can be generalized to target other processor architectures and instructions sets as long as a way to measure energy is provided, either through architectural counters, or via physical measurements with adequate granularity.

# Bibliography

- [1] “Jonathan g. koomey: Estimating total power consumption by servers in the us and the world,” <http://www-sop.inria.fr/mascotte/Contrats/DIMAGREEN/wiki/uploads/Main/svrpwrusecompletefinal.pdf>.
- [2] “In the data center, power and cooling costs more than the it equipment it supports,” <http://www.electronics-cooling.com/2007/02/in-the-data-center-power-and-cooling-costs-more-than-the-it-equipment-it-supports/>.
- [3] “Mandy patts: Microprocessor power impacts,” [https://tbach.web.cern.ch/tbach/thesis/literature/power\\_density\\_Pant-DASS.pdf](https://tbach.web.cern.ch/tbach/thesis/literature/power_density_Pant-DASS.pdf).
- [4] “Cisco visual networking index: Global mobile data traffic forecast update, 20142019,” [http://www.cisco.com/c/en/us/solutions/collateral/service-provider/visual-networking-index-vni/white\\_paper\\_c11-520862.html](http://www.cisco.com/c/en/us/solutions/collateral/service-provider/visual-networking-index-vni/white_paper_c11-520862.html).
- [5] S. M. Martin, K. Flautner, T. Mudge, and D. Blaauw, “Combined dynamic voltage scaling and adaptive body biasing for lower power microprocessors under dynamic workloads,” in *Proceedings of the 2002 IEEE/ACM International Conference on Computer-aided Design*, ser. ICCAD ’02. New York, NY, USA: ACM, 2002, pp. 721–725. [Online]. Available: <http://doi.acm.org/10.1145/774572.774678>
- [6] J. Kin, M. Gupta, and W. Mangione-Smith, “The filter cache: an energy efficient memory structure,” in *Microarchitecture, 1997. Proceedings., Thirtieth Annual IEEE/ACM International Symposium on*, 1997, pp. 184–193.
- [7] L. H. Lee, B. Moyer, and J. Arends, “Instruction fetch energy reduction using loop caches for embedded applications with small tight loops,” in *Low Power Electronics and Design, 1999. Proceedings. 1999 International Symposium on*, 1999, pp. 267–269.
- [8] O. Ergin, D. Balkan, K. Ghose, and D. Ponomarev, “Register packing: Exploiting narrow-width operands for reducing register file pressure,” in *Proceedings of the 37th Annual IEEE/ACM International Symposium on Microarchitecture*, ser. MICRO 37. Washington, DC, USA: IEEE Computer

Society, 2004, pp. 304–315. [Online]. Available: <http://dx.doi.org/10.1109/MICRO.2004.29>

- [9] R. Canal, A. González, and J. E. Smith, “Very low power pipelines using significance compression,” in *Proceedings of the 33rd Annual ACM/IEEE International Symposium on Microarchitecture*, ser. MICRO 33. New York, NY, USA: ACM, 2000, pp. 181–190. [Online]. Available: <http://doi.acm.org/10.1145/360128.360147>
- [10] P. Rosenfeld, E. Cooper-Balis, and B. Jacob, “Dramsim2: A cycle accurate memory system simulator,” *Computer Architecture Letters*, vol. 10, no. 1, pp. 16–19, Jan 2011.
- [11] S. Wilton and N. Jouppi, “Cacti: an enhanced cache access and cycle time model,” *Solid-State Circuits, IEEE Journal of*, vol. 31, no. 5, pp. 677–688, May 1996.
- [12] A. B. Kahng, B. Li, L.-S. Peh, and K. Samadi, “Orion 2.0: A fast and accurate noc power and area model for early-stage design space exploration,” in *Proceedings of the Conference on Design, Automation and Test in Europe*, ser. DATE ’09. 3001 Leuven, Belgium, Belgium: European Design and Automation Association, 2009, pp. 423–428. [Online]. Available: <http://dl.acm.org/citation.cfm?id=1874620.1874721>
- [13] 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*, June 2000, pp. 83–94.
- [14] S. Li, J.-H. Ahn, R. Strong, J. Brockman, D. Tullsen, and N. Jouppi, “Mcpat: An integrated power, area, and timing modeling framework for multicore and manycore architectures,” in *Microarchitecture, 2009. MICRO-42. 42nd Annual IEEE/ACM International Symposium on*, Dec 2009, pp. 469–480.
- [15] N. Binkert, B. Beckmann, G. Black, S. K. Reinhardt, A. Saidi, A. Basu, J. Hestness, D. R. Hower, T. Krishna, S. Sardashti, R. Sen, K. Sewell, M. Shoaib, N. Vaish, M. D. Hill, and D. A. Wood, “The gem5 simulator,” *SIGARCH Comput. Archit. News*, vol. 39, no. 2, pp. 1–7, Aug. 2011. [Online]. Available: <http://doi.acm.org/10.1145/2024716.2024718>
- [16] S. Lee, A. Ermedahl, S. L. Min, and N. Chang, “An accurate instruction-level energy consumption model for embedded risc processors,” *SIGPLAN Not.*, vol. 36, no. 8, pp. 1–10, Aug. 2001. [Online]. Available: <http://doi.acm.org/10.1145/384196.384201>
- [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, no. 2, pp. 146–154, April 2002.

- [18] —, “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, no. 2, pp. 146–154, April 2002.
- [19] W. Wang and M. Zwolinski, “An improved instruction-level energy model for risc microprocessors,” in *Ph.D. Research in Microelectronics and Electronics (PRIME), 2013 9th Conference on*, June 2013, pp. 349–352.
- [20] “Wang, wei and zwolinski, mark (2014) an improved instruction-level power model for arm11 microprocessor. in, high performance energy efficient embedded systems,” *Systems(HIP3ES), Berlin, DE, 23Jan2013*. 7pp..
- [21] Y. Wang and N. Ranganathan, “An instruction-level energy estimation and optimization methodology for gpu,” in *Computer and Information Technology (CIT), 2011 IEEE 11th International Conference on*, Aug 2011, pp. 621–628.
- [22] C. Luo and R. Suda, “A performance and energy consumption analytical model for gpu,” in *Dependable, Autonomic and Secure Computing (DASC), 2011 IEEE Ninth International Conference on*, Dec 2011, pp. 658–665.
- [23] D. Molka, D. Hackenberg, R. Schone, and M. Muller, “Characterizing the energy consumption of data transfers and arithmetic operations on x86-64 processors,” in *Green Computing Conference, 2010 International*, Aug 2010, pp. 123–133.
- [24] Y. S. Shao and D. Brooks, “Energy characterization and instruction-level energy model of intel’s xeon phi processor,” in *Proceedings of the 2013 International Symposium on Low Power Electronics and Design*, ser. ISLPED ’13. Piscataway, NJ, USA: IEEE Press, 2013, pp. 389–394. [Online]. Available: <http://dl.acm.org/citation.cfm?id=2648668.2648758>
- [25] J. Peddersen and S. Parameswaran, “Clipper: Counter-based low impact processor power estimation at run-time,” in *Design Automation Conference, 2007. ASP-DAC ’07. Asia and South Pacific*, Jan 2007, pp. 890–895.
- [26] R. Jordans, R. Corvino, L. Jozwiak, and H. Corporaal, “An efficient method for energy estimation of application specific instruction-set processors,” in *Digital System Design (DSD), 2013 Euromicro Conference on*, Sept 2013, pp. 471–474.
- [27] 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, no. 9, pp. 998–1010, Sep 2002.

- [28] P. Gschwandtner, M. Knobloch, B. Mohr, D. Pleiter, and T. Fahringer, "Modeling cpu energy consumption of hpc applications on the ibm power7," in *Parallel, Distributed and Network-Based Processing (PDP), 2014 22nd Euromicro International Conference on*, Feb 2014, pp. 536–543.
- [29] G. Contreras and M. Martonosi, "Power prediction for intel xscale reg; processors using performance monitoring unit events," in *Low Power Electronics and Design, 2005. ISLPED '05. Proceedings of the 2005 International Symposium on*, Aug 2005, pp. 221–226.
- [30] R. Joseph and M. Martonosi, "Run-time power estimation in high performance microprocessors," in *Proceedings of the 2001 International Symposium on Low Power Electronics and Design*, ser. ISLPED '01. New York, NY, USA: ACM, 2001, pp. 135–140. [Online]. Available: <http://doi.acm.org/10.1145/383082.383119>
- [31] C. Isci and M. Martonosi, "Runtime power monitoring in high-end processors: methodology and empirical data," in *Microarchitecture, 2003. MICRO-36. Proceedings. 36th Annual IEEE/ACM International Symposium on*, Dec 2003, pp. 93–104.
- [32] K. Singh, M. Bhadauria, and S. A. McKee, "Real time power estimation and thread scheduling via performance counters," *SIGARCH Comput. Archit. News*, vol. 37, no. 2, pp. 46–55, Jul. 2009. [Online]. Available: <http://doi.acm.org/10.1145/1577129.1577137>
- [33] X. Zhou, B. Guo, Y. Shen, and Q. Li, "Design and implementation of an improved c source-code level program energy model," in *Embedded Software and Systems, 2009. ICESS '09. International Conference on*, May 2009, pp. 490–495.
- [34] 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*, Aug 2009, pp. 651–656.
- [35] K. Natarajan, H. Hanson, S. Keckler, C. Moore, and D. Burger, "Microprocessor pipeline energy analysis," in *Low Power Electronics and Design, 2003. ISLPED '03. Proceedings of the 2003 International Symposium on*, Aug 2003, pp. 282–287.
- [36] E. Blem, J. Menon, and K. Sankaralingam, "Power struggles: Revisiting the risc vs. cisc debate on contemporary arm and x86 architectures," in *High Performance Computer Architecture (HPCA2013), 2013 IEEE 19th International Symposium on*, Feb 2013, pp. 1–12.
- [37] R. Kumar, K. I. Farkas, N. P. Jouppi, P. Ranganathan, and D. M. Tullsen, "Single-isa heterogeneous multi-core architectures: The potential for

- processor power reduction,” in *Proceedings of the 36th Annual IEEE/ACM International Symposium on Microarchitecture*, ser. MICRO 36. Washington, DC, USA: IEEE Computer Society, 2003, pp. 81–. [Online]. Available: <http://dl.acm.org/citation.cfm?id=956417.956569>
- [38] J. L. Henning, “Spec cpu2006 benchmark descriptions,” *SIGARCH Comput. Archit. News*, vol. 34, no. 4, pp. 1–17, Sep. 2006. [Online]. Available: <http://doi.acm.org/10.1145/1186736.1186737>
- [39] S. C. Woo, M. Ohara, E. Torrie, J. P. Singh, and A. Gupta, “The splash-2 programs: Characterization and methodological considerations,” in *Proceedings of the 22Nd Annual International Symposium on Computer Architecture*, ser. ISCA '95. New York, NY, USA: ACM, 1995, pp. 24–36. [Online]. Available: <http://doi.acm.org/10.1145/223982.223990>
- [40] R. P. Weicker, “Dhrystone: A synthetic systems programming benchmark,” *Commun. ACM*, vol. 27, no. 10, pp. 1013–1030, Oct. 1984. [Online]. Available: <http://doi.acm.org/10.1145/358274.358283>
- [41] “Simulation program with integrated circuit emphasis,” <http://bwrcs.eecs.berkeley.edu/Classes/IcBook/SPICE/>.
- [42] “Odroid xu+e,” [http://www.hardkernel.com/main/products/prdt\\_info.php?g\\_code=G137463363079](http://www.hardkernel.com/main/products/prdt_info.php?g_code=G137463363079).
- [43] “Samsung application processor packaging: Package on package,” <http://www.samsung.com/global/business/semiconductor/support/package-info/package-datasheet/application-processor#none>.
- [44] “Dfi ddr protocol,” <http://ddr-phy.org/>.
- [45] “Exynos 5 octa,” [http://www.samsung.com/global/business/semiconductor/file/media/Exynos\\_5\\_Octa.pdf](http://www.samsung.com/global/business/semiconductor/file/media/Exynos_5_Octa.pdf).
- [46] “Arm cortex a7 processor,” <http://www.arm.com/products/processors/cortex-a/cortex-a7.php>.
- [47] “Arm cortex a15 processor,” <http://www.arm.com/products/processors/cortex-a/cortex-a15.php>.
- [48] “Arm big.little technology,” [http://www.arm.com/files/downloads/big\\_LITTLE\\_Final\\_Final.pdf](http://www.arm.com/files/downloads/big_LITTLE_Final_Final.pdf).
- [49] “Samsung high-k metal gate (hkmg),” [http://www.samsung.com/us/business/oem-solutions/pdfs/Foundry\\_32-28nm\\_Final\\_0311.pdf](http://www.samsung.com/us/business/oem-solutions/pdfs/Foundry_32-28nm_Final_0311.pdf).
- [50] “Arm cci 400,” <http://www.arm.com/products/system-ip/interconnect/corelink-cci-400.php>.

- [51] “Arm big.little: The future of mobile,” [http://www.arm.com/files/pdf/big\\_LITTLE\\_Technology\\_the\\_Future\\_of\\_Mobile.pdf](http://www.arm.com/files/pdf/big_LITTLE_Technology_the_Future_of_Mobile.pdf).
- [52] “Arm architecture reference manual armv7-a and armv7-r edition,” <http://infocenter.arm.com/help/index.jsp?topic=/com.arm.doc.subset.architecture.reference/index.html>.
- [53] “The arm neon general-purpose simd engine,” <http://www.arm.com/products/processors/technologies/neon.php>.
- [54] K. DeVogeleer, G. Memmi, P. Jouvelot, and F. Coelho, “Modeling the temperature bias of power consumption for nanometer-scale cpus in application processors,” in *Embedded Computer Systems: Architectures, Modeling, and Simulation (SAMOS XIV), 2014 International Conference on*, July 2014, pp. 172–180.
- [55] “Intel 64 and ia-32 architectures software developer manuals,” <http://www.intel.com/content/dam/www/public/us/en/documents/manuals/64-ia-32-architectures-software-developer-manual-325462.pdf>.
- [56] D. A. Patterson and J. L. Hennessy, *Computer Organization and Design, Fourth Edition, Fourth Edition: The Hardware/Software Interface (The Morgan Kaufmann Series in Computer Architecture and Design)*, 4th ed. San Francisco, CA, USA: Morgan Kaufmann Publishers Inc., 2008.
- [57] J. L. Hennessy and D. A. Patterson, *Computer Architecture, Fourth Edition: A Quantitative Approach*. San Francisco, CA, USA: Morgan Kaufmann Publishers Inc., 2006.
- [58] H.-J. Cheng, Y.-S. Hwang, R.-G. Chang, and C.-W. Chen, “Trading conditional execution for more registers on arm processors,” in *Embedded and Ubiquitous Computing (EUC), 2010 IEEE/IFIP 8th International Conference on*, Dec 2010, pp. 53–59.
- [59] H.-H. Chiang, H.-J. Cheng, and Y.-S. Hwang, “Doubling the number of registers on arm processors,” in *Interaction between Compilers and Computer Architectures (INTERACT), 2012 16th Workshop on*, Feb 2012, pp. 1–8.
- [60] “Armv8-a architecture,” <http://www.arm.com/products/processors/armv8-architecture.php>.
- [61] J. Dongarra, “The linpack benchmark: An explanation,” in *Proceedings of the 1st International Conference on Supercomputing*. London, UK, UK: Springer-Verlag, 1988, pp. 456–474. [Online]. Available: <http://dl.acm.org/citation.cfm?id=647970.742568>