Feasibility of Accelerator Generation to Alleviate Dark Silicon in a Novel Architecture

Antonio ARNONE

PhD

UNIVERSITY OF YORK
Computer Science

June 2017

Abstract

This thesis presents a novel approach to alleviating Dark Silicon problem by reducing power density.

Decreasing the size of transistor has generated an increasing on power consumption. To attempt to manage the power issue, processor design has shifted from one single core to many cores. Switching on fewer cores while the others are off helps the chip to cool down and spread power more evenly over the chip. This means that some transistors are always idle while others are working. Therefore, scaling down the size of the chip, and increasing the amount of power to be dissipated, increases the number of inactive transistors. As a result it generates Dark Silicon, which doubles every chip generation [63]

One of the most effective techniques to deal with Dark Silicon is to implement accelerators that execute the most energy consumer software functions. In this way the CPU is able to dissipate more energy and reduce the dark silicon issue.

This work explores a novel accelerator design model which could be interfaced to a Stack CPU and so could optimise the transistor logic area and improve energy efficiency to tackle the dark silicon problem based on heterogeneous multi-accelerators (co-processor) in stack structure.

The contribution of this thesis is to develop a tool to generate coprocessors from software stack machine code. But it also employs up-to-date code optimisation strategies to enhance the code at the input stage. Analysis of the cores using key metrics, based on 65nm synthesis experiments and industry standard tool-sets. It further introduces a novel architecture to decrease the power density of the accelerator.

In order to test these expectations, a large-scale synthesis translation experiment was conducted, covering widely recognised benchmarks, and generating a large number of cores (in the thousands). These were evaluated for a range of key metrics: silicon di-area, timing, power, instructions-per-clock, and power density, both with and without code optimisation applied.

The results obtained demonstrate that one of two competing core models, 'Wavecore' (which is proposed in this thesis), delivers superior power density to the standard approach (which it refers to as Composite core), and that this is achieved without significant cost in terms of critical metrics of overall power consumption and critical path delays.

Finally, to understand the benefit of these accelerators, these auto-generated cores are analysed in comparison to a standard stack-machine CPU executing the same code sequences. Both the cores generation work and the benchmark CPU assume a 65nm CMOS process node, and are evaluated with industry standard design tools. It is demonstrated that the generated cores achieve better power efficiency improvements over the relatively CPU core.

Contents

Abstract iii

Acknowledgements xv

Declaration of Authorship xvii

1 Introduction 1
- 1.1 Hypothesis 2
- 1.2 Novel Contributions 3
- 1.3 Scaling vs Power 4
- 1.4 ARM architecture, 35 years of RISC architecture 6
  - 1.4.1 Register File and Pipeline architecture 6
- 1.5 The History of Stack Architecture 8
  - 1.5.1 Stack Architecture, a simple architecture 10
- 1.6 Dark Silicon 13
  - 1.6.1 The Key approaches to improving dark silicon 13
- 1.7 Summary: 15

2 Accelerator cores and Translator tool 17
- 2.1 Accelerators to reduce power consumption 19
- 2.2 Low Power Concept 21
- 2.3 Design Flow 24
- 2.4 Accelerator generation; design flow 26
- 2.5 SW/HW Function translation 28
  - 2.5.1 The Translator tool in detail 32
  - 2.5.2 Testing the translator tool 34
- 2.6 The Verilog test bench 41
- 2.7 The Composite Cores Architecture 49
- 2.8 The Wave-core Architecture 49
  - 2.8.1 The Wave-core architecture: An idea to reduce power 50
- 2.9 Summary: 51

3 Standard Core Generation Experiment 53
- 3.1 The Baseline core bench marks Input Comparison 55
  - 3.1.1 Input operands per core (Smaller benchmarks, baseline case) . 55
  - 3.1.2 Input operands per core (Larger benchmarks, baseline case) . . 58
  - 3.1.3 Input operands per core (All benchmarks, baseline case) 58
- 3.2 The Baseline core benchmarks Output Comparison 59
  - 3.2.1 Output operands per core (Smaller benchmarks, baseline case) 59
  - 3.2.2 Output operands per core (Larger benchmarks, baseline case) . 59
  - 3.2.3 Output operands per core (All benchmarks, baseline case) . . . 59
- 3.3 The number of states in the Finite State Machine 62
  - 3.3.1 Number of states per core (Smaller benchmarks, baseline case) 62

3.3.2 Number of states per core (Larger benchmarks, baseline case) . 65
3.3.3 Number of states per core (All benchmarks, baseline case) . . . 65
3.4 The Number of Instructions per Clock Cycle 65
3.5 Composite vs Wave-core Total Area 69
3.6 Composite vs Wave-core Timing 72
3.7 Composite vs Wave-core Leakage 73
3.8 Composite vs Wave-core Dynamic power 74
3.9 Composite vs Wave-core Total power 76
3.10 Composite vs Wave-core Power Density 77
3.11 Composite vs Wave-core Dynamic power for cores with Multiple States 79
3.12 Summary: 82

4 Impact of Code Optimisation (Stack Scheduling Code) 85
4.1 Comparison of Optimised Cores' Input 86
4.1.1 Input operands per core (Smaller benchmarks, with optimization) 87
4.1.2 Input operands per core (Larger benchmarks, with optimization) 89
4.1.3 Input operands per core (all benchmarks, with optimization) . 89
4.2 Comparison of Optimised Cores' Output 89
4.2.1 Output operands per core (Smaller benchmarks, with optimization) 91
4.2.2 Output operands per core (Larger benchmarks, with optimization) 93
4.2.3 Output operands per core (All benchmarks, with optimization) 93
4.3 Comparison of Optimised Cores' States 95
4.3.1 Number of states per core (Smaller benchmarks, with optimization) 96
4.3.2 Number of states per core (Larger benchmarks, with optimization) 98
4.3.3 Number of states per core (All benchmarks, with optimization) 98
4.4 Comparison of Optimised Cores' IPC 100
4.4.1 IPC per core (Smaller benchmarks, with optimization) 101
4.4.2 IPC per core (Larger benchmarks, with optimization) 101
4.4.3 IPC per core (All benchmarks, with optimization) 104
4.5 Comparing the Area of the optimised benchmarks in Composite and Wave-core architecture 105
4.5.1 Comparing the Area of the optimised benchmarks in Composite architecture 105
4.5.2 Comparing the Area of the optimised benchmarks in Wave-core architecture 107
4.5.3 Comparing Composite vs Wave-core architecture total Area for the optimized benchmarks 107
4.6 Comparing the Timing of the optimised benchmarks in Composite and Wave-core architecture 110
4.6.1 Comparing the Timing of the optimised benchmarks in Composite architecture 111
4.6.2 Comparing the Timing of the optimised benchmarks in Wave-core architecture 111
4.6.3 Comparing Composite vs Wave-core Timing of the optimised benchmarks 114

4.7 Comparing the Static Power of the optimised benchmarks in Composite and Wave-core architecture . . . 115
4.7.1 Comparing the Leakage of the optimised benchmarks in Composite architecture . . . 115
4.7.2 Comparing the Leakage of the optimised benchmarks in Wave-core architecture . . . 117
4.8 Comparing the Dynamic Power of the optimised benchmarks in Composite and Wave-core architectures . . . 119
4.8.1 Comparing the Dynamic power of the optimised benchmarks in Composite architecture . . . 119
Dynamic power per state . . . 119
4.8.2 Comparing the Dynamic power of the optimised benchmarks in Wave-core architecture . . . 121
4.8.3 Composite vs Wave-core Dynamic power of the optimised benchmarks . . . 122
4.9 Comparing the Total Power of the optimised benchmarks in Composite and Wave-core architecture . . . 124
4.9.1 Composite total power for the optimized benchmarks . . . 125
Total power applying clock gating in Composite architecture . . . 127
4.9.2 Wave-core total power for the optimized benchmarks . . . 127
4.9.3 Power gating and Clock gating Composite vs Wave-core Total Power . . . 131
4.10 Comparing the Power Density of the optimised benchmarks in Composite and Wave-core architectures . . . 132
4.10.1 Composite Power density for the optimized benchmarks . . . 132
Power Density for the Composite architecture when the clock gating is applied . . . 132
4.10.2 Wave-core Power density for the optimized benchmarks . . . 136
4.10.3 Wave-core vs Composite Power density when when power gating and clock gating are applied . . . 136
4.11 Pearson Correlation review of statistical significance of results in Composite and Wave-core architecture . . . 137
4.12 Summary . . . 139

5 CPU vs Core Power Analysis **141**
5.1 NOMAD MACHINE ARCHITECTURE . . . 141
5.1.1 Evaluation of NOMAD in terms of selected cores power analysis is undertaken as follows . . . 142
5.2 Methodology of comparison (NOMAD versus Accelerator Cores) . . . 142
5.2.1 Results of selected cores analysis for both CPU and Composite and Wave-core architectures . . . 144
5.3 Analysis using overage power for Lund CPU model . . . 145
5.3.1 Methodology of comparison IP cores vs Lund CPU model . . . 145
5.3.2 Comparing Composite and Wave-core architecture vs Lund CPU architecture . . . 148
5.4 Analysis using average power model for NOMAD CPU . . . 150
5.5 Summary of power models and results . . . 150

6 Conclusion **155**
6.1 Main Contributions and Implications . . . 155
6.2 Future Work . . . 159

A VHDL Code- Composite and Wave-core **161**

B Hamming distance and Verilog test-bench **175**

C Total Area **191**

D Power report **193**

E Timing report **195**

Bibliography **199**

List of Tables

3.1 Combined benchmarks Total Area differences quartile 70
3.2 Combined benchmarks Timing differences quartile 72
3.3 Combined benchmarks Leakage differences quartile 74
3.4 Combined benchmarks Dynamic Power 75
3.5 Combined benchmarks Total Power differences quartile 76
3.6 Combined benchmarks Power Density differences quartile 78
3.7 Combined benchmarks Dynamic Power multi states differences quartile 79
3.8 Combined benchmarks Power Density multi states differences quartile 81

4.1 Pearson correlation coefficient total area 110
4.2 Pearson correlation coefficient timing 115
4.3 Pearson correlation coefficient Static Power 119
4.4 Dynamic power per state. 121
4.5 Pearson correlation coeffcient Dynamic power 124
4.6 Pearson correlation coefficient for the four factors 139

5.1 Lund CPU vs IP cores 148

List of Figures

1.1 Moore's law . . . 1
1.2 Stack CoDA . . . 3
1.3 Processor Performance . . . 5
1.4 Register Bank . . . 7
1.5 Reverse Polish Equation . . . 10
1.6 Stack Architecture . . . 11
1.7 Split Cache . . . 12
1.8 Implicit Model . . . 13
1.9 Trende frequency-year . . . 14
1.10 multicore scaling . . . 15
1.11 Energy Efficiency . . . 16

2.1 CoDA Architecture . . . 18
2.2 CoDAs Architecture . . . 18
2.3 PICO . . . 20
2.4 Power Density . . . 22
2.5 Hardware Optmization . . . 22
2.6 CMOS dynamic power . . . 23
2.7 CMOS static power . . . 24
2.8 Power consumption . . . 25
2.9 Niagra2 . . . 25
2.10 Methodology . . . 27
2.11 High level . . . 30
2.12 Stack effect . . . 30
2.13 Backtracking algorithm . . . 31
2.14 Distruption effect . . . 31
2.15 Generic State Machine . . . 33
2.16 Testing methodology . . . 35
2.17 Assembly code example . . . 36
2.18 Composite architecture Model . . . 37
2.19 Composite VHDL 1 code . . . 38
2.20 Composite VHDL 2 code . . . 39
2.21 Composite VHDL 3 code . . . 40
2.22 Wave-core model . . . 42
2.23 Wave-core VHDL 1 code . . . 43
2.24 Wave-core VHDL 2 code . . . 44
2.25 Wave-core VHDL 3 code . . . 45
2.26 Wave-core VHDL 4 code . . . 46
2.27 Wave-core VHDL 5 code . . . 47
2.28 Energy transition . . . 48
2.29 Switching transistors . . . 48
2.30 Composite Architecture . . . 50

2.31 Wave-core Architecture 51

3.1 Composite Diagram Block 54
3.2 Wave-core Diagram Block 54
3.3 Input Core Baseline 56
3.4 Input Core Baseline 2 57
3.5 Totl Number of Input Baseline 58
3.6 Output Core Baseline 1 60
3.7 Output Core Baseline 2 61
3.8 Totl Number of Output Baseline 62
3.9 Number of States Core Baseline 1 63
3.10 Number of States Core Baseline 2 64
3.11 Total Number of States Baseline 66
3.12 Number of IPC Baseline 1 67
3.13 Number of IPC Baseline 2 68
3.14 Total Number of IPC 69
3.15 Total Area Baseline 70
3.16 Combined benchmak Total Area 71
3.17 CMOS Foot Print 71
3.18 Baseline Timing 72
3.19 Baseline leakage 73
3.20 Scatter Plot Leakage 74
3.21 Baseline Dynamic Power 75
3.22 Baseline Dynamic Power Scatter Plot 76
3.23 Baseline Total Power 77
3.24 Baseline Power Density 78
3.25 Power Density Multi States 79
3.26 Power Density for cores with Multi States 80
3.27 MIPS Composite vs Wave-core 81
3.28 Coparind Energy Efficiency MIPS 82

4.1 Scheduling 86
4.2 Optimized Input BFNS 88
4.3 Optimized Input CFS 90
4.4 Optimized Input Overal benchmarks 91
4.5 Optimized Output BFNS 92
4.6 Optimized Output CSF 94
4.7 Optimized Output Overal benchmarks 95
4.8 Optimized States BFNS smaller core 97
4.9 Optimized States BFNS Bigger cores 99
4.10 Optimized States Overal benchmarks 100
4.11 Optimized IPC BFNS smaller cores 102
4.12 Optimized IPC CSF Bigger cores 103
4.13 IPC Overal benchmarks 104
4.14 Total Area Composite Optimized B. 106
4.15 Total Area Wave-core Optimized B. 108
4.16 Venn diagram 109
4.17 Composite vs Wave-core architecture total Area for the optimized benchmarks 110
4.18 Composite Timing for the Optimized B. 112
4.19 Wave-core Timing for the Optimized B. 113

4.20 Composite vs Wave-core timing for the Optimized B. 114
4.21 Composite Leakage for the Optimized B. 116
4.22 Wave-core Leakage for the Optimized B. 118
4.23 Composite Dynamic Power for the Optimized B. 120
4.24 Composite Baseline vs Shannon Dynamic Power per Total States . . . 122
4.25 Wave-core Dynamic Power optimized benchmarks 123
4.26 Composite vs Wave-core Dynamic Power Shannon 124
4.27 Composite vs Wave-core Dynamic Power Shannon for the most changing cores 125
4.28 Composite total power for the optimized benchmakrs 126
4.29 Composite total power for the shannon optimized benchmark 128
4.30 Baseline vs Shannon States 128
4.31 Comparison of Baseline vs Shannon power gating applied 129
4.32 Wave-core total power for the optimized benchmakrs 130
4.33 Wave-core total power for the shannon optimized benchmark 131
4.34 Composite vs Wave-core total Power, power gating and Clock gating . 133
4.35 Composite Power Density Optimized Benchmarks 134
4.36 Comparison N. of States in optimized core vs Baseline 135
4.37 Baseline vs Shannon Power Density Optimised Benchmarks 136
4.38 Wave-core Power Density for the four benchmarks 137
4.39 Composite vs Wave-core Power Density, power gating and clock gating aplied 138

5.1 CPU vs Core model 145
5.2 Nomad CPU vs Core 146
5.3 NOMAD CPU; Composite vs Wave-core 147
5.4 Lund vs Core 149
5.5 NOMAD CPU' model vs Core 151
5.6 Ration Lund CPU vs core 153
5.7 Ration NOMAD CPU vs core 154

B.1 Hamming Dstance 175
B.2 Hamming Dstance cube 4D 176

Acknowledgements

"This work was funded by the Engineering and Physical Sciences Research Council (EPSR) (Grant No. GR/R67668/01) of the Government of the United Kingdom"

The translator tool discussed in this thesis was developed in collaboration with Dr Chris Bailey. I am very grateful for his help.

First at all, I would like to thanks my parents, Teresa Casciello and Arnone Giacomo, and my sisters and husbands Mariarosaria and Marialba Arnone, Ernesto and Luigi for their continuous stimulation. Furthermore my nieces; Emanuela, Aurora, Gloria, Chiara and Eliana.

A special thanks to Asselya Katenbayeva to follow me in a part of this adventure.

I would like to thank my supervisor Dr Chris Bailey for his support and guidance.

I would also like to thank my internal assessor Dr James Walker for his insightful suggestions and discussions during my research and thesis advisory panel meetings.

I would like to thank Dr Nick Pears for serving as my internal assessor for the first year.

I would also like to thank my colleagues in the Advanced Computer Architecture Group and Intelligent Systems Research Group for their hints, in particular Dr Martin Albrecht Trefzer, Dr Gianluca Tempesti and Dr Mike Freeman.

Finally, I would like to thanks my friends Adolfo, Silvia, Bharath, Roshanth, Federico, Fabiana, Luca, Giovanni and Anna, and also my old friends Pip, Antonio, Lorenzo, Giannino and Jessica, for the amazing and funny time spent together. . . .

Declaration of Authorship

I, Antonio ARNONE, declare that this thesis titled, “Feasibility of Accelerator Generation to Alleviate Dark Silicon in a Novel Architecture” is original except where otherwise stated or cited to a source. This work has not previously been presented for an award at this, or any other, university. Some of this work in this thesis has resulted in the following publication:

- Comparing Composite vs. Wave-cores in a Novel Dark-Silicon Methodology, Antonio Arnone and Christopher Crispin-Bailey, University of York (paper accepted) at Prime 2016 12th Conference on PhD Research in Microelectronics and Electronics.
- Dark Silicon, Comparing Wave Cores with Composite State Machines, Antonio Arnone and Christopher Crispin-Bailey, University of York (poster presentation) at Design with Uncertainty Opportunity and Challenges workshop The University of York, York (UK), 17-19 March 2014.
- Dark Silicon – MULTICORE PROCESSOR, Antonio Arnone and Christopher Crispin-Bailey, University of York (poster presentation) at NANO- TERA/ARTIS INTERNATIONAL SUMMER SCHOOL on Embedded System Design EPFL, nano-Tera.ch, ETH, 9-13 September 2014.

"Our responsibility is to do what we can, learn what we can, improve the solution, AND PASS THEM ON"

Richard Feynman

Dedicate To ALBA MARIA GNARRA (grandmother)...

Chapter 1

Introduction

The topic of this thesis centres on the Dark silicon area. The dark silicon area is the percentage of the active transistor during any one operation.This thesis introduces a novel approach to decrease the power density and so ameliorating the Dark Silicon area.

Going back in time, the first integrated circuit was built by Jack Kilby in 1958 using two transistors. Today an integrated circuit can contain up to 9 million transistors mm2 [31]. This remarkable growth has come from the constant shrinking of transistors and advancement in the manufacturing processes. However, as transistors become smaller and faster, they dissipate less power.

Moore's law Figure 1.1 has described this as an exponential growth, with a doubling of the number of transistors every 18 months. However, according to the International Technology Roadmap for Semiconductors, the growth was to decelerate around 2013 and reach saturation point in 2015 [2]. Today it is predicted that the decreasing of transistors will stop in a decade or so.

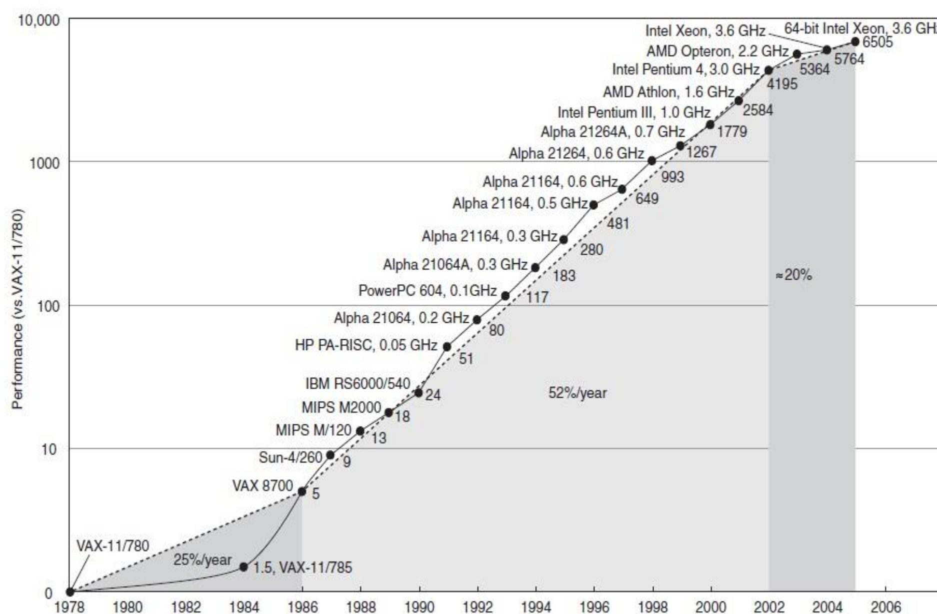


FIGURE 1.1: shows the growing of transistor inside the chip across years.

Decreasing the size of the transistor has increased the challenge of dissipate power. Nowadays, the main factor to take into account in designing processor architecture is performance and more specially the power consumption. The short-term

solution to improving power dissipation – increasing the number of transistors inside the chip and decreasing the cost– has forced CPU design engineers to move into multicore, many-core processors. However, this technique does not fully address the problem. Increasing the number of cores inside the chip does not give higher performance in design faster processor, but it gives a better response to dissipate power. In fact, to manage the temperature on the top of the chip, only a few cores are turned on at any one time [31]. Moreover, while these cores are operating, the others are switched off. Consequently, this technique generates dark silicon, "which will reach the 90 % of the chip at 11nm size by 2020 [48]" or dim silicon. Dim silicon means that the chip is clocked at less frequency To put it more simply, dark silicon refers to percentage of inactive transistors at an operative moment.

1.1 Hypothesis

The increase in the number of transistors inside a chip has increased the power density, therefore the chip simply overheats if it is used at full capacity and requires a bespoke cooling solution. The Register File consumes about 20 % of power inside the chip [10] and pipelines produce hotspots if they are used for a long time. One of the techniques to improve the power efficiency of the CPU design is Co-processors Dominant Architecture (CoDAs). Many researchers have investigated this concept to explore this type of architecture and devise new techniques to decrease the power consumption. However, all investigations are exclusively applied on a Reduced instruction set computing (RISC) or Very long instruction word (VLIW).

This work aims to generate coprocessors from software functions to be interconnected to Stack CoDAs architecture model. To translate software into hardware unit cores, a translator tool in (ANSI C) will be developed to map machine code (stack assembly code) into a hardware structure or coprocessor. As will be explained in more detail, starting from a stack assembly code generated from a stack compiler, the translator tool generates two types of VHDL code called Composite and Wavecore to be synthesised to FPGA and ASIC target. The translator tool is able to generate thousands of cores or hardware units, to be observed in respect of time, area and power.

A stack CoDA architecture differs from a traditional CoDA architecture [71] because it uses stack memory systems and a stack processor as the host CPU. The coprocessors share the stack memory with the host CPU and use a transferring data model to switch on a single core per time. Figure 1.2 shows a stack CoDA architecture model.

The shared memory helps to maximize the performance, because, it gives access to the data from either the master CPU or accelerator as more suitable. To minimize the power consumption, it uses switching enable architecture to power up a single core per time. The larger part of the architecture is off at any time. It can also be expanded to have a higher aggregate throughput. If Figure 1.2 is imagined as a tile, it can interconnect many tiles by point to point L2 cache [71]

While this work does not want to compare a classic architecture with Stack Machine Architecture – such could not be done in a single work – the main objective of this thesis is:

"Is there a benefit of applying accelerators cores technique in a stack machine processor context?"

In more detail:

1. Demonstrate that a translation of stack machine code, software function, into cores, hardware units, is feasible.
2. Evaluate how cores perform in term of power, area and timing when comparing the classical implementation of state machine with a novel implementation of state machine architecture core design.
3. Observe whether advanced code optimization can contribute to better core performance in terms of power, area and timing.
4. Compare CPU versus cores to understand what performance the cores give.

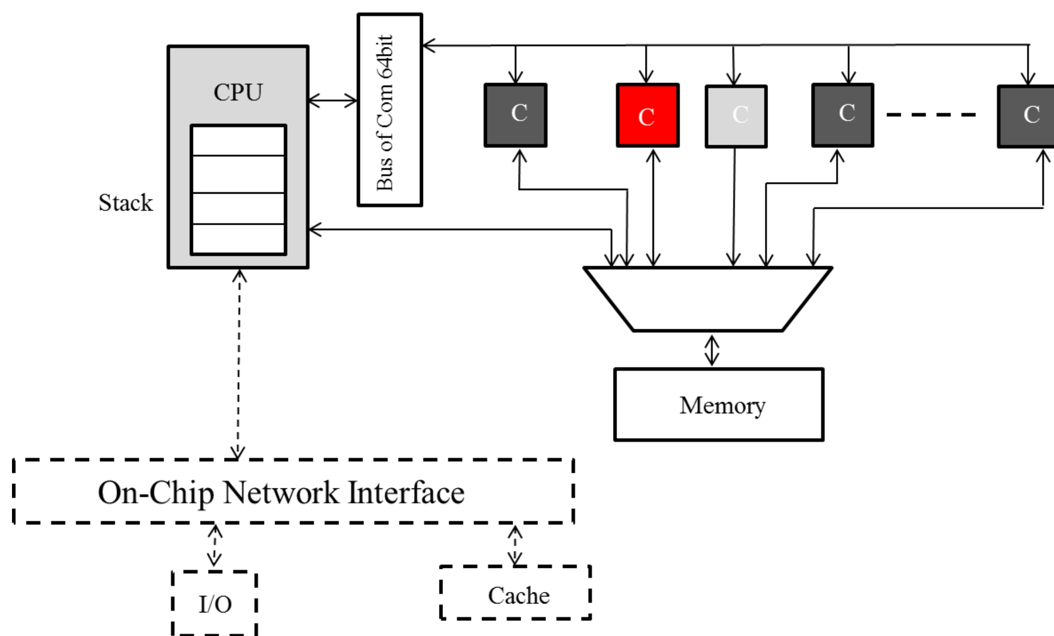


FIGURE 1.2: A stack CoDA architecture, the coprocessor shares the stack memory with the host CPU. And more it has a switching system technology to empower a single core per time.

1.2 Novel Contributions

The main novel contributions of this thesis are:

1. The translator tool: a fully automated technology to translate SW/HW. The translator tool translates software functions, machine code fragments generated from a stack compiler into two VHDL types of hardware units, cores, which here is referred to as Composite and Wave-core architecture, to be synthesised to FPGA and ASIC target and then analysed in terms of power, area and timing
2. A Composite architecture to generate IP core. The Composite architecture is implemented using a classical state machine but in a stack structure data flow.

The model uses a shared logic structure to process a sequence of instruction groups in successive hardware states, and is closely related to the Mealy State Machine model.

3. Wave-core architecture. It uses an evolution of the traditional state machine to improve the power density. In contrast to the Composite architecture, Wave-core treats each state in a computational flow as a separate hardware structure, operating in a fashion that is similar to a systolic array. Division into states is identical to that of the Composite core model, bounded by load or store operations. However, the computation structure for each state is implemented as a separate circuit and these are activated in turn, hence a 'wave' of activity rolling through a 'daisy chained' structure, stage by stage. This means no logic circuit is used in successive states, reducing localized hot-spot occurrence.

4. Insight into how code structure, scheduling, affects core efficiency. The translator tool starts generating VHDL from a stack machine code generated from a stack compiler. Nowadays compiler optimization techniques are widely used in a compiler to generate more efficient machine code. This work uses stack scheduling techniques to permit to the stack compiler to generate better machine codes. It eliminates memory references and so decreases the number of states in the state machine. This technique aims to improve the power consumption of the cores

5. Using clock and power gating to improve the efficiency of the architecture. Stack scheduling helps to reduce the number of states in the state machine. The reduction gives the opportunity to use a clock gating technique. Reducing the frequency of the clock reduces the dynamic power without modifying the execution time. Also, the characteristic of the Wave-core architecture which implements the architecture in separate circuits triggered in turn permits the application of power gating technique to switch off the inactivated circuits. As a consequence the static power is improved.

6. Comparing CPU vs Cores. To understand the benefit that the cores can give when interconnected to a CPU, a comparison is done between functions run at software level selecting two topologies of stack architecture and the same functions run at hardware level using the Composite and the Wave-core architecture.

1.3 Scaling vs Power

Computers play a significant role in modern society. The integrated circuit has revolutionised the electronics industry, from medical equipment to agriculture machinery, from automotive to domestic appliances. It has changed the way we socialise and interact with each other. Science has achieved astonishing developments by applying computational data to experimental theory.

This success is attributed to the fact that in the last 30 years computing machines have made extraordinary progress. Nowadays, devices with a very high performance in CPU architecture, main memory and disk storage have become much cheaper compared with the devices built in the 1980s [31]

This improvement of performance has come from advances in the VLSI manufacturing process and innovation in architecture and design.

In fact, major progress started at the beginning of the 1970s. In this era, technology realised a very large integrated circuit (VLSI). Thousands of transistors were fitted inside a single chip, and this increase in the the number of transistors made it possible to integrate CPU, ROM, RAM and glue logic on one piece of silicon. Figure 1.3 shows this progress as roughly 30% per year.

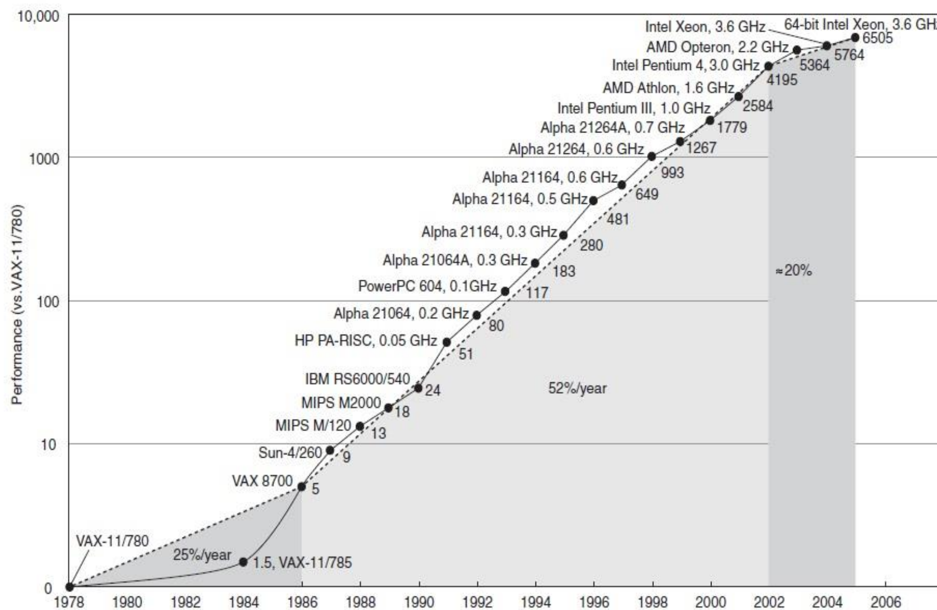


FIGURE 1.3: Growth in processor performance since the mid-1980s. The graph shows the increasing on performance in Computer. In the 90s it grew 52% per year [31].

The increase of transistors created a power issue. To deal with power, design engineers needed to make a major impact at architectural level. In fact, the best known techniques for reducing power are turning on the circuit for less time, parallelism and decreasing the frequency.

To reduce power, processors have combined voltage scaling down and pipeline. Power is expressed as: P=CV2F. Scaling down voltage reduces the quadratic power; while increasing the number of functions to be pipelined reduces the amount of logic per clock cycle at the cost of increasing registers.

Today the most consolidate technique in reducing voltage could not be longer applied, because threshold voltage cannot be scaled down, due to rising leakage of current. For instance the shrinking of transistors decreases the gap between the source and drain and so increases the leakage [23].

New multicore architectures are emerging to control the power on the top of the chip. Smaller cores have less logic, short wires and faster local memory access. The register file is replaced with a larger memory. Co-processors are more energy efficient than general purpose processors.

However, caches and co-processors do not increase the percentage of active transistors; they improve performance and lower power density per square millimetre [26].

In fact, today many specialised cores such as graphic units and DSPs are integrated inside the chip, which suggests that from today to the future the market will develop more co-processor dominated architectures (CoDAs).

1.4 ARM architecture, 35 years of RISC architecture

Acorn Computer Limited of Cambridge designed, implemented and developed the first ARM RISC architecture between 1983 and 1985. In the 1990s it became Acorn RISC Machine and quickly dominated the IC market due to its developing the British Broadcasting Corporation microcomputer (BBC). This powerful personal computer was used in many research laboratories and schools. The unexpected success of the BBC microcomputer pushed Acorn engineers to design and upgrade the CPU hardware architecture, building the second generation of the machine. At this time the only promising and simple architecture was the Berkeley RISC I, so the new chip was constructed starting from this concept

ARM was designed to supply the embed market, therefore to minimize the power consumption in the design concept only the essential units were implemented, while others were rejected: register windows (because they occupied much space); delayed branches (because they increased the complexity of managing them), and so on.

Another key point of ARM is the instruction set. It was built referring to the RISC architecture but, at the same time, in order to achieve a better density in code, few concepts referring to the CISC architecture were implemented.

1.4.1 Register File and Pipeline architecture

Increasing the number of transistors inside the chip made it possible to increase the amount of memory. To improve the speed of the CPU and reduce the amount of time to access the main memory, CPU architects and engineers expanded the CPU register and implemented a fast L1 cache [55]

RISC architecture incorporates the register file. The register file is a unit that incorporates all general purpose registers and is formed using an array of registers interconnected to the CPU.

RISC architecture opened a new way to design the CPU. As we have seen, the architecture was created at the compiler level. The register file is more flexible for ordering, executing and holding data, all of which advantages of the compiler allow for order and pipeline expression and result in a faster operating CPU.

Another important feature of register files is that caching variables reduces the memory traffic. This is another aspect that makes the RISC processor faster. In addition, because the register locations can be named using fewer bits than would be the case in the main memory, the code appears more compact [42]

Register files have multiple separated ports to read and write in the register. Another option is that the register file is allocated a specific register by the CPU to communicate directly with the main memory. However, a new RISC CPU has the option to remap the remained register file, which gives more flexibility in allocating data.

A better and simpler description of register files could be expressed as an implementation of many registers and a decoder, with Address, DataIn, Enable inputs and a DataOut output. Typically, to speed up data execution the CPU reads two values in a clock cycle and writes back the result.

To address 4 registers 2 bits are used. Therefore, in a small word machine, to address many registers few registers will be selected for the special purpose function and so there is no need to address all of them. Another solution to addressing registers using few bits is to separate address registers and data registers, which means that to execute an operation fewer bits are required to select the specific address register.

To address more registers using fewer bits words, the CPU incorporates a bank register. The bank register is a set of register file folders. The folder is represented by a number of pages, and each page by a register file. The CPU can only read and write on a single page at a time. However, instructions have the flexibility to move into the bank register, and therefore the data can move around the pages. The input and output between pages is through shared buses; access to the buses is controlled through tristate buffers, and only one line must have access to the bus at once. Figure 1.4 below is a representation of a register bank.

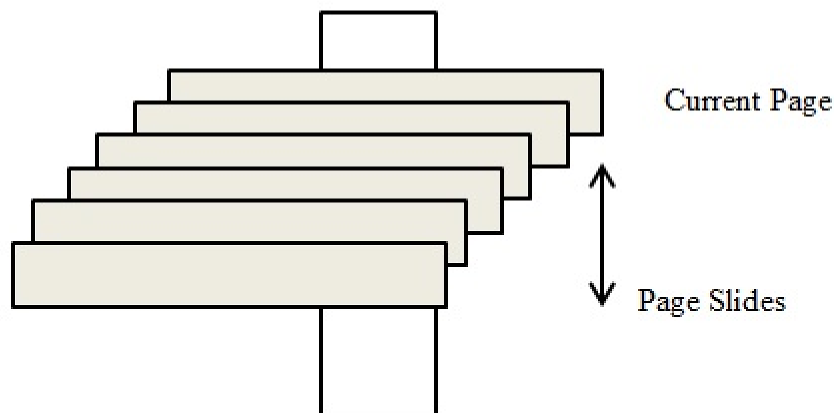


FIGURE 1.4: The register back consists of a multiple register file connected together. They are also called pages. This technique is used to expand the register file and store more data. The grey boxes represent the register files or pages. The register bank enables just one page per time and can go up and down to the arrow [26].

It is noticeable that there are different pages that could be accessed one at a time. The equation that demonstrates how many registers can be addressed is expressed by n=log2 (N) and m=log2 (M), where N is the total number of registers in the entire bank register and M is the number of registers per page. Assuming N>M, the register could be addressed using 2 variables n and m, where n and m are the numbers of bits to map the N and M registers [25]

The Register file has been a very useful technique to process parallelism. However, the growing number of registers inside the chip cause the energy consumption to increase, and simpler operations suffer due to this extra complexity [20]. For example, the register file in the Motorola M-core processor uses 16 % of the total energy consumption and 42 % of the total data path energy [70]. What makes the register file poor in energy efficiency is that it typically has 2 inputs to write the data and one output to read the result. Also, as mentioned above, the register file has a control line to control the connection of the single register. In case of simultaneous accesses, the register file gets multiple lines in a high state and so it generates a hot spot [69]. In order to activate all registers each time and not corrupt data, the architecture has to drive a significant amount of current, which should increase with the increase

of the number of active states per time. In operation process the register file has many inactive registers per clock cycle. As a result, the sum of these registers leads to an increase in static power [20]. Likewise, the greater the number of transistors inside the chip, the slower the supply voltage scales [31]. The register file leaks a lot of power, which particularly affects. Today, embedded systems which are widely used nowadays. Therefore, This power dissipation is the greatest challenge in the electronic market. Consequently many studies explore novel techniques to reduce, improve or replace register files in the CPU architecture [39].

The Modern CPU incorporates pipeline techniques to enable fast data execution. This Pipeline process works as an industrial assembly line where more inputs are accepted before the first input appears as output.

The most important factor in the pipeline technique is time. These timing problems are called hazards. A hazard occurs when a stage cannot be performed in the succeeding clock. Thus the timing problems are grouped into Structural hazards, Data hazards and Control hazards.

Today hazards are not the only problem. Another matter of con- concerned is that the pipeline has to be run at full clock frequency during the process, otherwise the performance will be significantly less efficient.. It is true that the pipeline reduces the number of logic levels between stages to increase the operating frequency, but to run the pipeline at full frequency and meet the time so far CPU engineers scale down the supply voltage. However, today the voltage cannot be scaled down more and as a result the pipeline technique suffers from power dissipation and hotspots [23].

1.5 The History of Stack Architecture

In 1946 Alan M. Turing anticipated the Stack machine. He coined the terms BURY and UNBURY to refer to calling up and returning routines (pop), (push) [65]. In 1957 Klaus Samelson and Federich L. Bauer patented the concept. They were working on a compiler and they implemented the push-pop algorithm theory [5]. In the same year Charles Leonard Hamblin also developed the idea independently [24].

Barton invented the first Stack machine called B5000 [13]. It was the first machine to adopt hardware/software trade-offs. The machine was capable of running with a high level language “ALGOL”. It used the Master Control Program (MCP) operative system [43] and for the first time a machine supported virtual memory. The second generation start with the B6500 in 1968 [24] which had the feature of the hardware-managed activation record. Later versions were the B6900 and B7700 [13]

In 1972 the HP 3000 series arrived. It was basically an evolution of Burroughs, with the advantage of supporting real-time processing [64]

In 1974 the ICL 2900 series arrived. It was developed by International Computer Ltd. It was based on Burroughs’ concept but with a difference: it was based on an accumulator, an idea derived from the Manchester MU5 [13]

In the second generation of Stack computing, Philip J. Koopman Jr. produced some of the best theoretical research. His book Stack Computer, The New Wave remains the best reference.

To describe the second generation of Stack, I refer to the thesis “Second-Generation Stack Computer Architecture,” by Charles Eric LaForest [38], which synthesises the most important architecture and describes in detail the main aspect of the new features in reordering operands in the stack memory of Stack Architecture.

In 1983 Chuck More designed the NC4016, a 16-bit processor for the Forth programming language. The NC4016 implements an unencoded instruction to control concurrently the ALU, the memory and the stack. It is a very small chip, about 4000 gates in about 16000 transistors, and the low number of gates suggests a small architecture with low power consumption. The Stack is separated from the main memory using an external buffer [36]

In the early 1980s an emergent idea was to create an architecture that could interconnect many CPUs to work together and accomplish several tasks at once, and so the Transputer (transistor and computer) was the first architecture designed to perform the tasks concurrently [32] [53]

The Transputer was leading the market in the 1980s. It interconnected in a single memory and a serial communication. The main purpose was parallelism computing. In fact, the IMS T414 introduced in 1985 with this parallelism was applied in different products, such as digital signal processing, robot control and image synthesis. The last version, the IMS T800-30, could perform two and half million floating point operations per second. The IMS 800 was integrated in the Harp project and so into parallel computing [32]. The main idea was to develop a low cost but faster supercomputer based on reconfigurable nodes of Transputers.

In the early 1980s an emergent idea was to create an architecture that could interconnect many CPUs to work together and accomplish several tasks at once, the Transputer (transistor and computer) was the first architecture designed to perform the tasks concurrently [32].

The RTX-2000, based on the NC4016, added new features to the architecture, such as a multiplier accelerator 16X16. The Stack is built on a chip that can be portioned to support fast task switching, an interrupt controller, a byte-swapped memory access, and counter/timers. [51]

In 1995, the Minimal Instruction Set Computer (MISC) called MuP21, had two processors, with the secondary processor acting as an accelerator and therefore controlled by the primary. The master had 5-bit instruction in a 20-bitword memory. The ALU and the stacks were instantiated in 21-bit to support arithmetic carry. The data Stack was six cells deep and the return Stack four cells deep. It incorporated an address register for memory access and the storage of temporary values. The secondary one, which was a video processor, had priority over the memory bus, the instruction set of 5 bit as the primary and could generate a 16-colour NTCS signal output. The entire architecture used just 7000 transistors and dissipated only 50mW.

The c18 designed by Moore in 2001 had two address registers for on chip communication, a watchdog timer, a few hundred words of RAM implemented on the chip, and could dissipate just 20mW [44].

A Low-power Microprocessor based on Stack Architecture, Lund University, has been designed using 65nm node technology. It consumes 20uW/MHz and uses just 0.16 square mm of area [60]

The NOMAD CPU processor architecture is an experimental CPU from the University of York. In this thesis it will be used to compare software vs hardware, see Chapter 5. Instruction must be issued in order and one per clock cycle. However, it uses an optimization compiler and so makes it out of order completion architecture.

That is to say, by observing all architectures three main advantages come out; First and most important all the architectures consume a small amount of power, secondly the architectures are very tiny:, the MuP21 uses just 7000 transistor while the Lund CPU use just 0.16mm square, and finally using a stack process that executes the instructions in order facilitates the interconnection of parallel microprocessors.

All these attributes are fundamental in designing modern CPUs and alleviate the Dark-Silicon problem.

1.5.1 Stack Architecture, a simple architecture

Stack machines adopt the reverse Polish notation instruction set [36]. Basically, the algorithm uses a parenthesis-free notation. To clarify, instead of reordering and giving priority to parenthesis, it evaluates the equation directly from left to right. A typical example could be (3+3)/2 = 3. If it is considered without parenthesis the result is 4.5. Figure 1.5 shows how the reverse Polish notation works. For instance, in reverse Polish notation also known as postfix notation, every operator follows its operands [89]. This technique avoids the need for parentheses. For example, in the equation 1.1 the operator "/ & +" follow 3, 3 and 2 as a result it has "3+3=6", "6/2=3". It becomes very clear how in a Stack memory the operands and operators work to complete the equation without errors.

(3+3)/2

/

/+3

/+33

/6

/62

3

FIGURE 1.5: implemented in reverse polish notation. It executes the funtion without considering brackets. The operators follow the operand. To implement the function above, it starts from "/" then "/+3" after "/+33" at this stage sum "3+3" and then "/6" finally "/62" result 3.

The Stack Architecture uses a Stack memory. The data and the operations are stored in the same Stack. By default, the operands are saved in a specific location. The main characteristic of Stack is that it typically performs the operation that is placed at the top of the Stack; this technique enables the ALU to call the data directly from the top. Figure 1.6 shows an example of Stack Architecture

The Top of Stack and Next On Stack hold the two operands to be executed from ALU. The ALU, after executing the operation, gives the results back to Stack. At the same time the next value of the Stack shifts up. Following this process in a sequential chain, the top of Stack holds the input operand, the next to the Stack the other input operand necessary for the operation, the ALU performs the operation and stores the result back into the top of the Stack and the Next to the Stack will be updated. If the result needs to be stored in the main memory, it first gets into the stack and then by store operation the value will be saved in the memory.

To avoid stack overflow the stack is usually attached to the main memory. That is to say, it is the poorest technique to use to expand the Stack contents. As is clear,

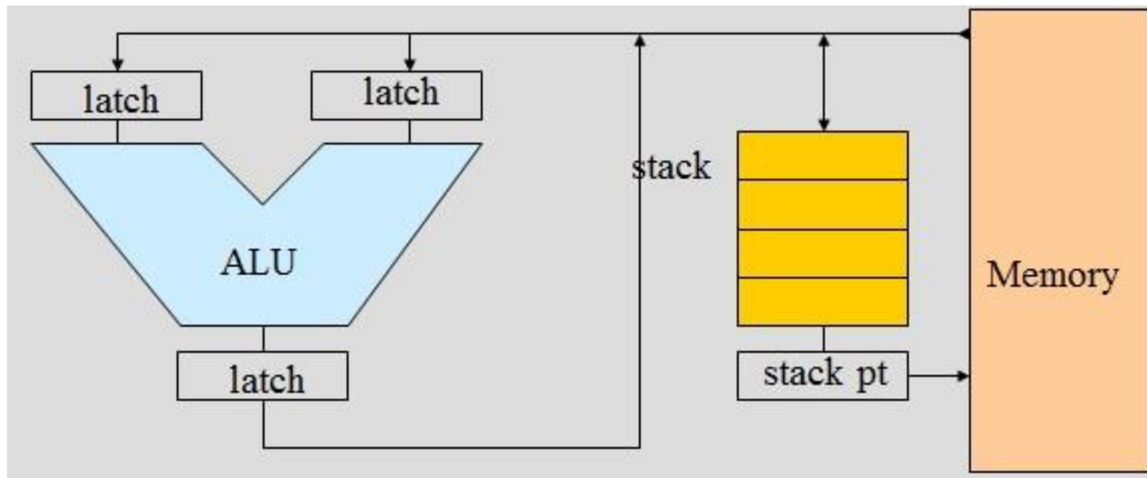


FIGURE 1.6: Stack processor computational model.

communication between the main memory and Stack memory is very slow, so it detracts from fast data execution.

Dr Bailey in his thesis [3] explained that to perform a computation, Stack Architecture needs just three or four variables. This means that only few values need to be on the Stack. However, a better configuration to expand the Stack could be a Multi-Level Cache or Split Cache schemes, in this way it improves the time of communication as shown in Figure 1.7 [4]

One of the advantages in stack architecture is that the memory is controlled without register mapping, because the operand is always placed on the top of the stack. Koopman explained in his book [36] how this characteristic could lead to the generation smaller decoders. Bailey [3] went deeper and so represented in a graph, Figure 1.8, the comparison between instruction set density and operand addressability for CISC, RISC, Stack, GPU and DSP.

In general purpose processors the Stack Architecture has the lower complexity of instruction and the implicit operand addressability because of the way stack memory works. For those attributes, Stack requires smaller program size and a simpler decoder [38], but also generates fast procedure calls and interrupts, all characteristics suitable for embedded systems [35]

Because Stack Architecture requires small program size, it reduces power consumption and also reduces the magnitude of the memory. As a consequence of this small memory a rapid memory chip can be used and thus a better performance in a virtual memory environment is achieved [4].

Koopman in his book [36] described the efficiency of The Stack Architecture to support better Interrupt response latency. Stack machine has very quick interrupt response latency, because most of the Stack machine instruction could be performed in a clock cycle and more due performing always the value on the top of the stack, it needs just save the next instruction that will be needed when back from the interrupt. On the other hand thinking at RISC architecture to perform an interrupt first has to save the procedure and to do this; it must go through pipeline and then can execute an interrupt. In the same way when returns back from the interrupt to avoid losing information it has to restore the procedure before carry on with the process.

He also described how the apparent disadvantage of using the single top element of the Stack is actually an advantage: stacks have compact instructions because they

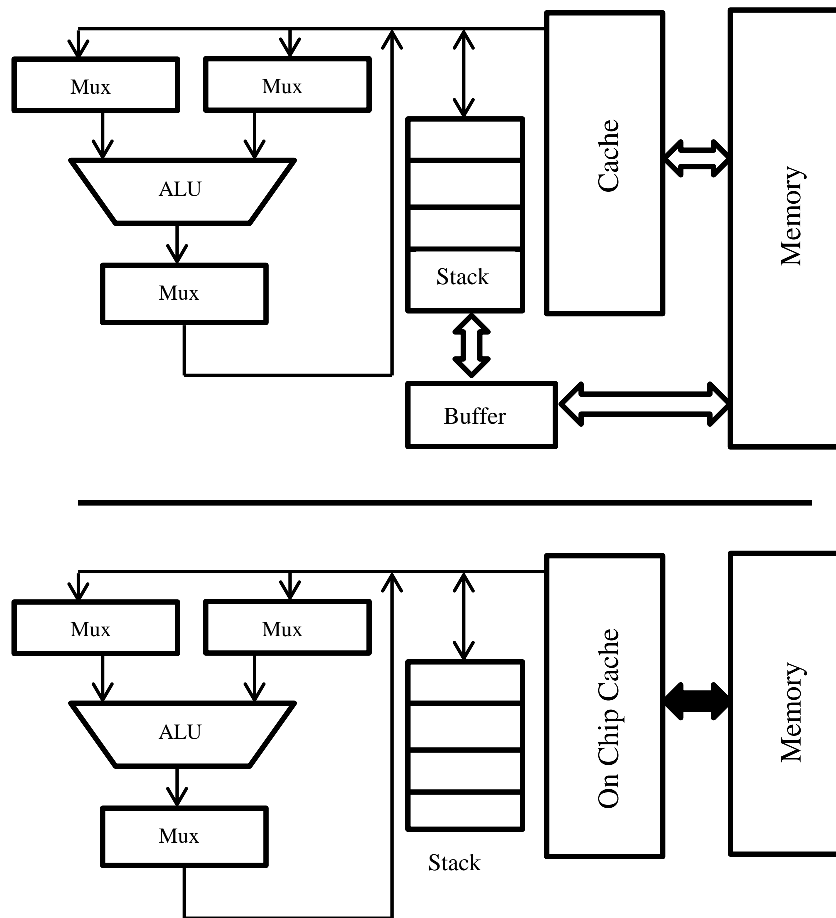


FIGURE 1.7: Split Cache schemes & Multi-Level Cache. These two different techniques help to avoid stack overflow and at the same time improve the time of communication from the stack to the main memory and vice versa.

do not need to identify the source of an operand; Stack memory needs just a single port because just a variable needs to be pushed or popped from the Stack. And more, the reductions of memory makes reduces the complexity of the decoder to manage it and so use fewer transistors to implement it. The simplicity of the Stack involves easier compiler, not just due to the simplicity of the architecture but also because in many compilers there is a Stack intermediate level before the mapping of the register machine.

To conclude all these advantages target the Stack architecture as an optimal solution to improve the power consumption and so ameliorate Dark silicon.

Indeed, Chapter2 will explain that using CoDA architecture and cutting off all complex logic units in a classical architecture will improve the power consumption and so ameliorate the Dark Silicon problem. On the other hand, the Stack architecture is a simple and energy efficient architecture as list above that already incorporates all of these features as shown above. Therefore, it becomes an excellent candidate to be selected as a host CPU in CoDA architecture to reduce power consumption and improve Dark Silicon.

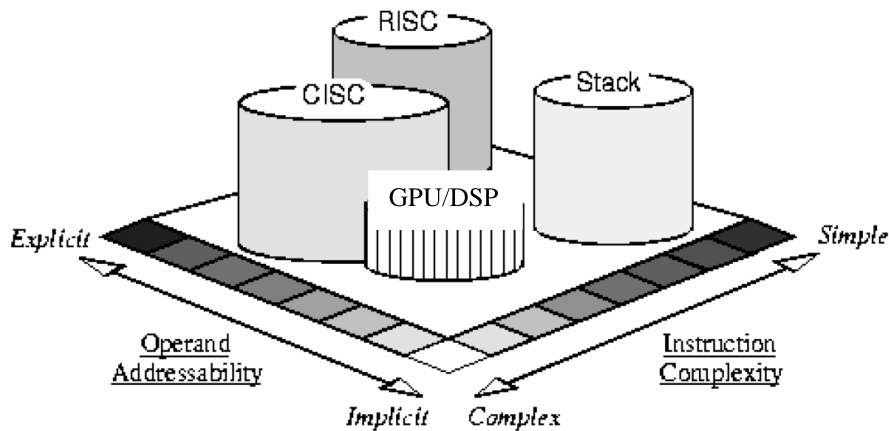


FIGURE 1.8: Classification of processor models. The implicit model shows the complexity of instruction.Stack architecture has the simpler instruction complexity.[36].

1.6 Dark Silicon

Today almost every device has a processor inside. Microcontrollers have become an integral part of modern equipment, because transistors are cheap. Shrinking transistors enables processors to become faster, cheaper and to use less power. However, while scaling, a number of factors constrain an improvement in performance. Today the most challenging issue is how to address energy consump- tion.

The problem of power is not going away. However, it can be improved. In the 1990s the power issue was predicted. Basically it has arisen because the voltage is scaling down more slowly than the frequency is scaling up. Figure 1.9 shows the trend over the years, and that the frequency is increasing.

1.6.1 The Key approaches to improving dark silicon

In post-Dennard scaling, progress is expressed in power consumption [46].The problem of power dissipation has forced a transfer from a processor with one core to one with many cores. Therefore, a large fraction of the chip is in fact dark or in dim silicon. It must be clear that using a multicore architecture does not address the problem but is just an industry solution to bypass the issue.

Figure 1.10 shows a typical example of how the industry controls power using cores.

To stabilise the power, the architecture can add more cores and leave the frequency constant or run the logic faster and leave the core constant.

To deal with dark silicon and decrease every femto-joule, I list three promising approaches [63]

1. The Shrinking Horseman: It is logical to consider building a smaller chip, that is reducing the number of transistors per node size, instead of having dark silicon in the architecture. However, it has been proved that using this technique will not increase the performance of the chip; in fact, it is the most unhelpful way to approach the problem. It is true that small chips are cheaper and have lower leakage compared to power gate efficiency. However, the IC market behaves in the opposite way. Moore's law predicts the success of the transistors

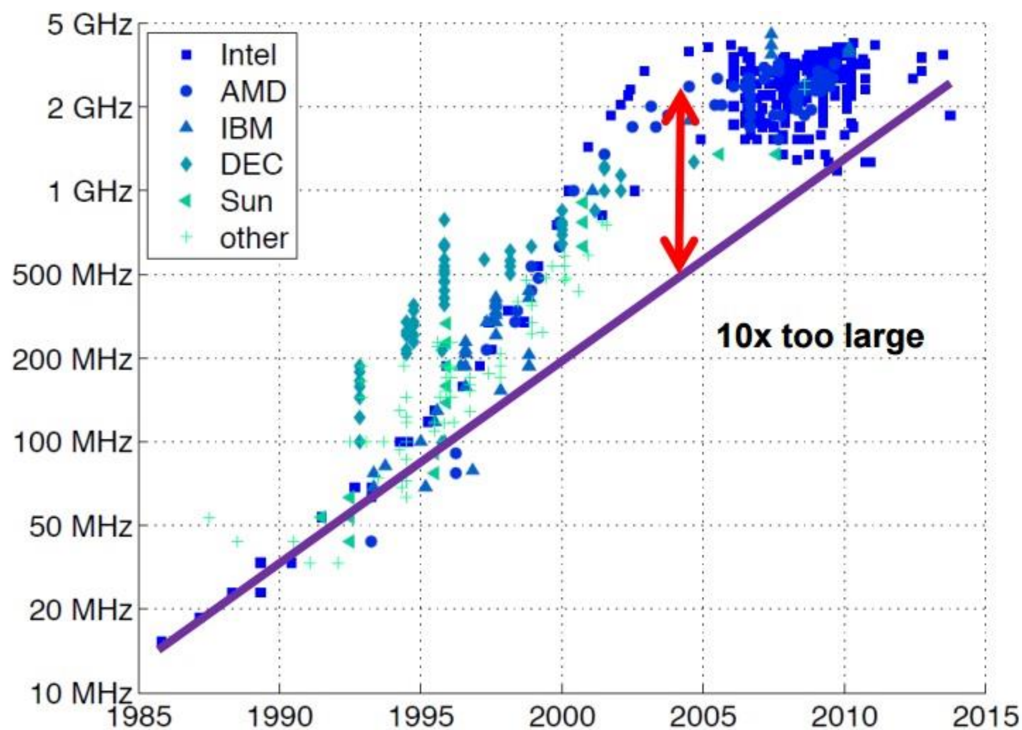


FIGURE 1.9: Trend between years and frequency. The trend shows an increasing of frequency up to 10x [23].

because shrinking transistors produce an increase in chip size, and therefore package-board co-design and design cost will be amortised by the increasing of chip size per generation and that push to invest in the next generation. It must be clear that shrinking chips will invert the IC market. In reality it will require more expensive chips compared to performance, which will lead to the semiconductor industry becoming less and less attractive [63]. Another aspect of the shrinking chip is the power density. It has been demonstrated that shrinking the chip increases the power density. The peak hotspot rise can be expressed as: Tmax = TDP * (Rcon v + k/A). Tmax is the rise in temperature; TDP is the target thermal design power of the chip, Rcon v is the heatsink thermal convention resistance, k is the property that incorporates many-core design, and finally A is the area of the chip. It becomes clear that by decreasing A the second term becomes much larger and the temperature rises [68].

2. The Dim Horsemen: It refers here to dim silicon, the part of the chip which is not run at full frequency. Dim silicon is a consolidated technique already applied in many CPUs to control the power. It has been explained that the power can be controlled by scaling down the frequency or implementing a logic unit such as big cache, which uses less time per clock cycle [23]. A promising technique to improve the power is the Near-Threshold Voltage. It basically works for processors that are used in parallel. It has been proved that this technique can bring about improvements of up to 8x [63]. Also sub-threshold logic seems a promising technique to enable certain applications to reduce power. To operate on sub-threshold logic transistors switches behind threshold voltage. It has

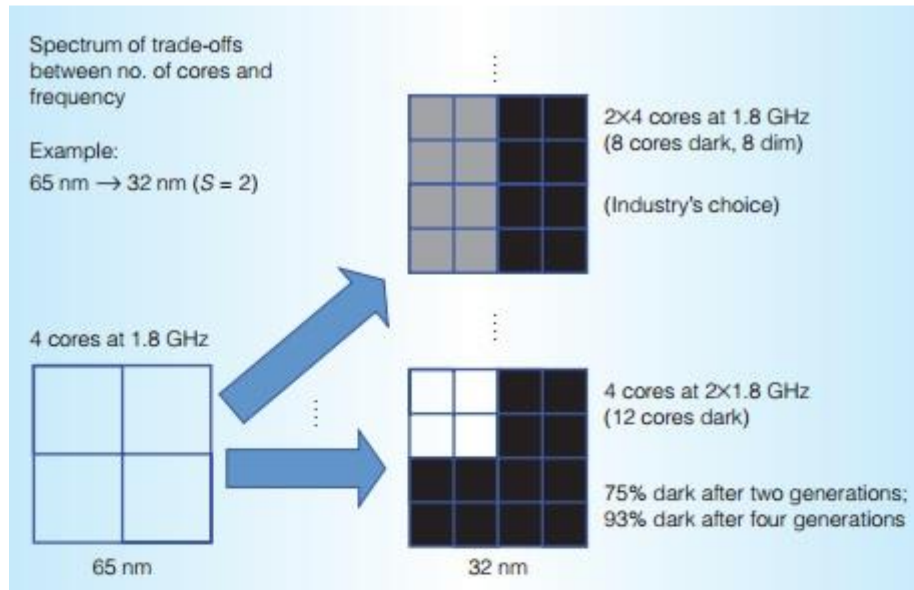


FIGURE 1.10: Multicore scaling leads to a large amount of dark silicon. It shows the two possible solution that industries use to deal with Dark Silicon [62].

the advantage of being easily implemented and stable at low frequency [58].

3. The Specialised Horseman: Co-processor Dominant Architecture (CoDA) seems a more realistic architecture to improve power consumption. It is based on accelerators; therefore, the market will require processors that integrate more independent accelerators. These accelerators are focused on specific tasks and they are implemented inside the CPU as independent peripherals. Therefore, working while the CPU is off or idle, it can achieve faster execution at lower clock speed, thus using less power and at the same time having less software complexity. Figure 1.11 shows the trade-off between the power energy efficiency of different processor architectures. It is clear that specialised cores improve the energy by up to 1000x. This new set of architecture raises a new set of questions. Adding more independent cores to a CPU means generating software with more performance and complexity. A typical example could be CUDA. It is a parallel computing platform and application interface model created by Nvidia, which is not portable between similar architectures [12]. Also, specialised cores can generate processors that are less applicable in a general purpose environment, perhaps they become too specific. It also changes the way to program the chip to obtain maximum performance. Another important point is to find the right functions inside the program that allow for saving the maximum amount of energy and, last but not least, to reduce the gap between hardware and software implementation engineers.

1.7 Summary:

Threshold voltage has stopped scaling down; for this reason the percentage of active transistors in a CMOS technology decrease with the decreasing of size of the

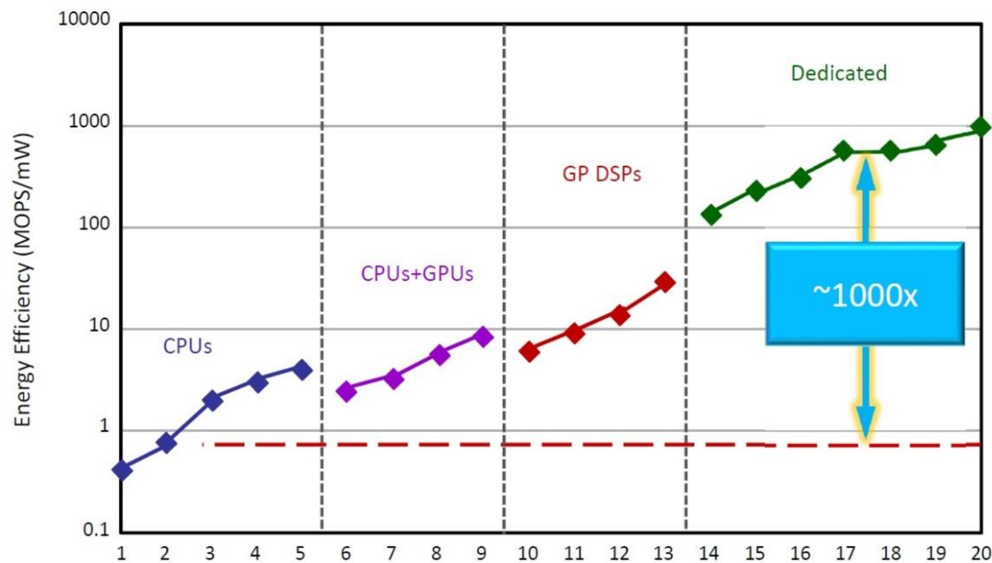


FIGURE 1.11: Energy efficiency of different types of CPU configuration. Dedicate circuit are the most energy efficient [50].

chip. Dark silicon that is the percentage of inactive transistor will cover 90 % of the chip area soon. The Register file and pipeline appear not to be the best techniques to reduce power consumption. Therefore to improve the power efficiency of the architecture much research is focused on CoDAs processor as a mean to move software into hardware. While many approaches tend to use RISC or VLIW architecture as a host, this work has the intention to demonstrate the feasibility to generate coprocessor from software function to be interconnected to Stack CoDA architecture model. Stack machine consumes a small amount of power and can be built using a smaller number of transistors. These two attribute make it a good candidate to design low power CPU. To demonstrate this hypothesis, at first This work will build a full automated translator tool. The translator tool translates stack machine code into two topologies VHDL hardware IPs, called Composite and Wave-core to be analysed in terms of area, power and timing. And then, it observes the performance of cores when compared with a Stack CPU processor

Chapter 2

Accelerator cores and Translator tool

The widening gap between rapidly increasing transistor budgets, inefficient power dissipation and thermal capacity has led to an increase in the dark silicon area inside the chip [63]. In order to make the best use of the available silicon area and improve the power dissipation, researchers are exploring new architectural solutions.

An emerging technique that improves power consumption is the Co-processor Dominant Architecture (CoDA). One notable example of this is the GreenDroid project [61].

The key idea of GreenDroid is to implement a new CPU architecture to supply the mobile phone market of the future. To reduce power consumption, it implements many software functions of Google's Android OS into hardware specialised accelerators. GreenDroid calls its accelerators "Quasi Specific Cores" because they carry out several general purpose computations inside the system [67].

Figure 2.1 shows a typical tile of CoDA architecture. Coprocessors use the same memory model of the host CPU. They have a circuit switching to enable just one core to work at a time working, and a shared data cache L1. Notably, the shared memory helps the stub function to direct the data to the host CPU or accelerator as more appropriate. The switching system enables just one core per time to save energy [71]

A set of tiles is used to run multi workload. They are interconnected by point to point L2 cache, Figure 2.2 shows an example.

To improve the design cycle and the power efficiency of the chip, GreenDroid has developed a toolchain that looks for the most energy demanding software function to implement into hardware, and then synchronises the CPU to the co-processor to execute the job. The tool is also able to synthesise and map a 45nm node ASIC chip size. This new architecture improves the power dissipation up to 13.5 % when compared with a traditional architecture. To achieve this greater improvement, the translator tool is able to receive a program and use a program dependence graph [18] to extract the most power consumption functions to be translated into hardware [67] [21].

GreenDroid calls the co-processors Conservation-cores or Quasi-specific Cores because they are fragments of code that could perform multiple functions. The Qs-Cores are implemented using a depipelining and a cachelets technique [67].

In data-path composite operations, many functions are subset of a function. In other words, the output of one function becomes the input of the next function. During the execution a lot of data tends to replicate, but memory and lower operations data have different requirements. Depipelining linked these two requests; therefore

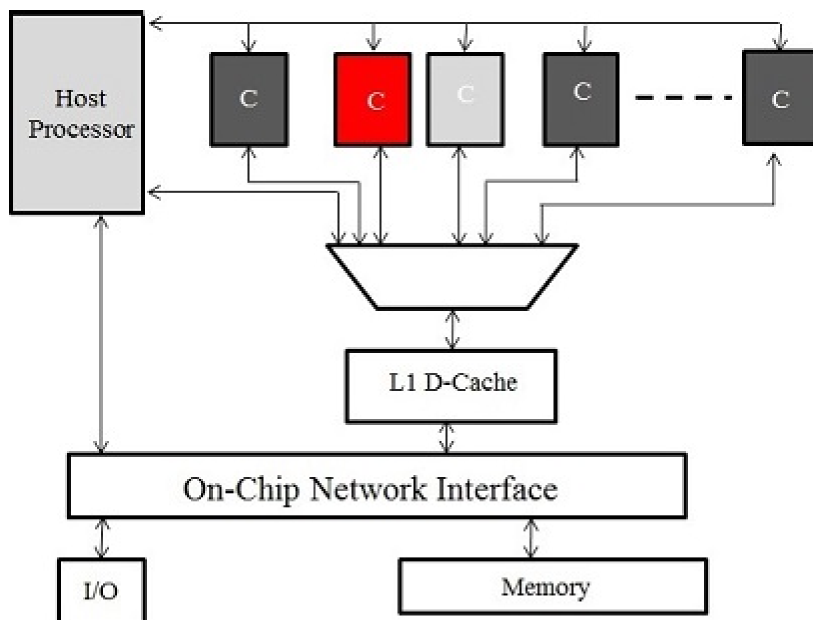


FIGURE 2.1: In CoDA architecture, coprocessors share the memory system with the host CPU and use a switching architecture to enable just once core at time working.

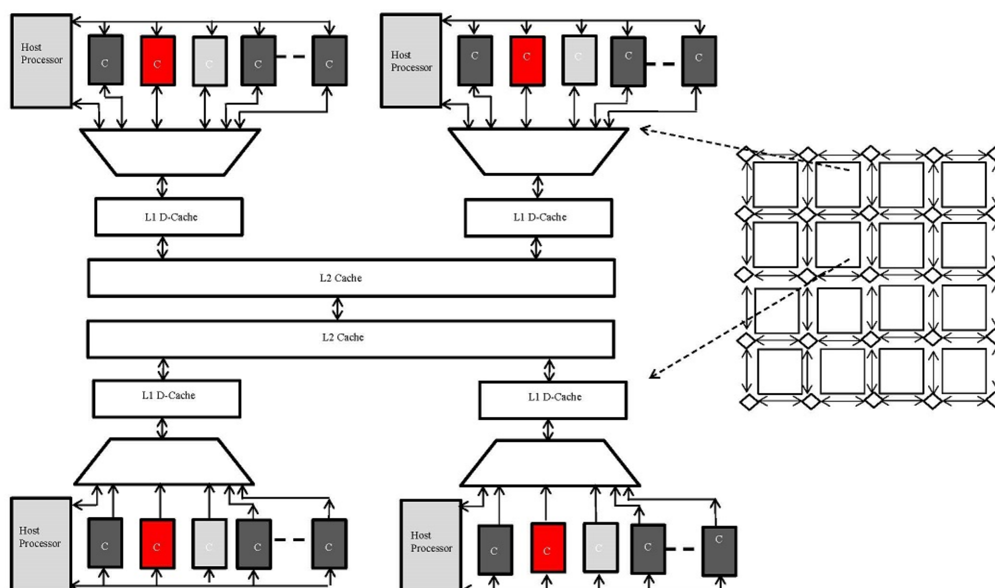


FIGURE 2.2: A typical CoDAs architecture, cluster, shows how the tiles are interconnected. Tiles are linked by L2 cache memory. Each tile contains a host CPU and many accelerators sharing the data by L1 cache memory. In the side it shows a cluster CoDAs architecture.

memory is run at higher frequency to meet the memory interface time, while datapath recording the replicated data in small L0 caches runs at lower frequency to save energy [21].

The translator tool converts software into hardware is based on PICO: Automatically Designing Custom Computer [61]. Figure 2.3.

The PICO system design is formed on a CPU, a template co-processor and a memory interface. The template co-processor is an empty unit already interfaced to the memory but without logics. This helps the engineer to design a customized logic and wrap it inside. The memory interface helps to synchronize the design and improve the life cycle of the project.

Another significant CPU model based on CoDA architecture is the Microchip 8-bit PIC core independent architecture family. It is concentrated on independent accelerators to achieve faster data execution, low power and less complex software. To reduce the power, it idles the CPU while allowing the peripherals to work at full frequency. Using this technique, it decreases the computational time by up to 150x and dissipates just 0.09mW/MHz [37].

Similar but with a different architecture, University of Wisconsin-Madison proposes Dynamically Specialized Execution cores. Based on a RISC architecture. The main idea is to reduce the hardware by catting off; fetching, decoding, and registering access. The cores are interconnected in a switch circuit and each core, which implements just the logic, can receive, transmit or just pass data [54].

The Politecnico di Milano looks inside the problem at memory level. It has created an accelerator store. It, in contrast with traditional cores, instead waiting for data to be processed, looks inside the memory and decides which data to process. In this way performing the most power consuming workload in hardware improves the power efficiency of the chip [8].

On the other hands, Processor based on co-processor architecture have also few weakness. Firstly an architecture with a higher number of accelerators increases the leakage power. Because almost of the time they are sat in idle mode, therefore using a higher number of accelerator the architecture might consume more power than a general purpose architecture. Then, the architecture can suffer for energy consumed by memory. In fact in a idle mode the memory cache consumes around 100mW/MB considering 32 or 45 nm node technology [12]. A coDAs architecture that tends to implement larger and many memory cache to store and direct data to different coprocessors. In design where the memory systems is not well implemented can waste more energy than a traditional architecture. [30].

2.1 Accelerators to reduce power consumption

While dramatic improvement of overall circuit performance has been achieved as a result of shrinking transistors, and Moore's law has thus far proved to be sustainable, power density has continued to be a concern for future scalability.

A key issue is that circuits have reduced in size but not in power density. Figure 2.4 shows the trend between size and power density, and indeed peak power density has increased. This leads to the problem of the silicon area being underutilised due to thermal constraints, and the corresponding desire to find new ways to use the silicon area by working around that problem while delivering good levels of performance on work per square um. Consequently, while power consumption is of great

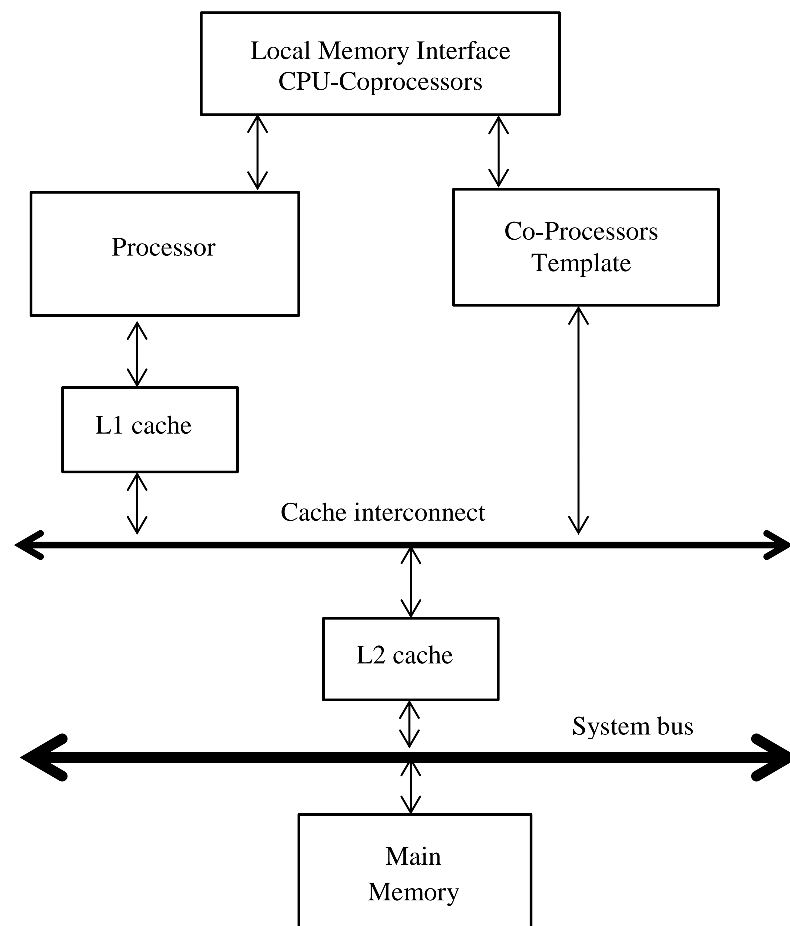


FIGURE 2.3: The PICO design is formed using a processor, a template co-processor, and a memory interface. The template co-processor has just a memory interface to permit the engineer to customize and wrap the logic.

concern, simply minimising power consumption as a performance goal is a potentially misleading path. Such an approach does not necessarily address the fine-grain variation across the chip in respect of temperature and localised power density.

An emerging technique that allows for a reduction in power in a general purpose computing architecture is the use of heterogeneous co-processors, which are effectively specific or general purpose hardware cores spread around the chip.

Based on multi-accelerators architecture, the key point of this project is to investigate a new CPU model to improve energy efficiency. It has as main techniques the compiler and the accelerators, while cutting off all hardware scheduling such as register file and pipeline from the chip and implementing only a Stack register and a memory cache. The compiler has to optimise the hardware at the high level (see Chapter 4). Figure 2.5 gives an ideal example of optimisation of compiler hardware synthesis. It shows a generic hardware multiplier to perform (4x2=8). It executes the operation in a 4 clock cycle using a shifter register, an adder and a multiplexer interconnected among each other. While on the other side it shows the compiler hardware optimisation. To execute the same operation, it uses just a shift register and performs the task in just one clock cycle. At this stage it becomes clear how the circuit has improved in timing (just a clock cycle), in power (less logic operation), and in memory (it uses just one operand).
"However, it is important to emphasise that this thesis is centred on accelerators and not on compiler optimisation."

The accelerator or co-processor is a computational resource to perform the job of the primary processor (CPU). By extracting, in most cases, the most intensive tasks from the main processor, the co-processor can improve the speed and power consumption of the overall system by between 100 and 1,000x [41]. Put simply, accelerators allow the design engineer to build a customised architecture inside a general purpose processor with all advantages of specific purpose architectures.

The accelerator performs just a specific function every time it is required by the CPU. And so to improve the power consumption of the architecture, while the co-processor is in operation mode, the main CPU goes into idle mode.

Nowadays the market is orientated toward chips with integrated FPGAs to facilitate the implementation of accelerators and achieve fast processors with low power consumption. Intel has announced a new Xeon Chip [33], Xilinx, which is already on the market and has an ARM core in its FPGAs.

2.2 Low Power Concept

Today, power is the first issue to consider when designing a CPU. Power consumption should be considered at the early stages of the project. A good approach is to make a trade-off between power consumption, memory, cost, speed and complexity.

Power dissipation in CMOS technology is grouped into dynamic power and static power. We refer to dynamic power when the transistors are in operation mode, while static power is when the transistors are just powered on. As we have seen, the equation for dynamic power is: V^ 2*f*C

Controlling the frequency of each gate can improve the dynamic power. For instance, a more accurate equation is expressed as follows: Pdynamic = α CV^ 2f

The frequency of gates is expressed by the activity factor α. In easy words, in a built gate, node, it represents the possibility that it switches from a lower state to higher [34] [23]. Observing the equation in more detail, the most important factor is

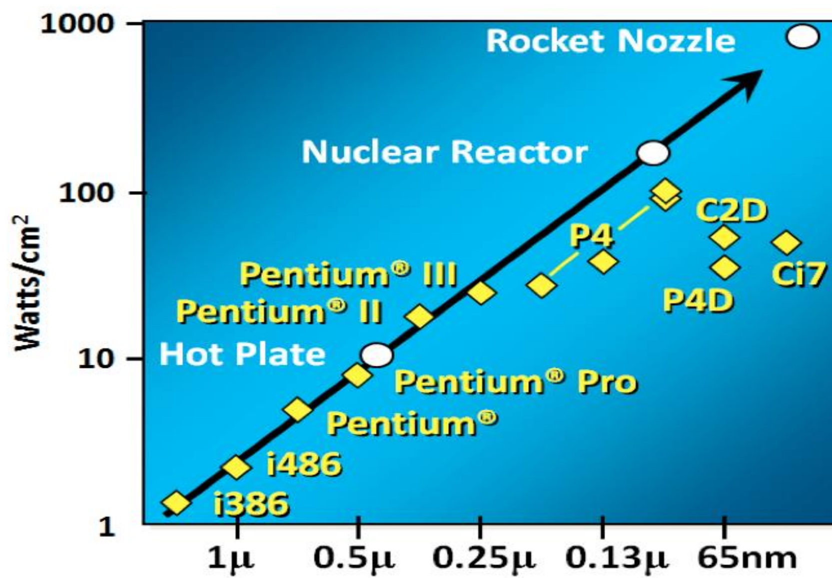


FIGURE 2.4: Power density trends. It shows the increasing of power on decreasing the node side. Modern architecture at 65nm consumes around 1000Watt/cm ˆ 2[41]

.

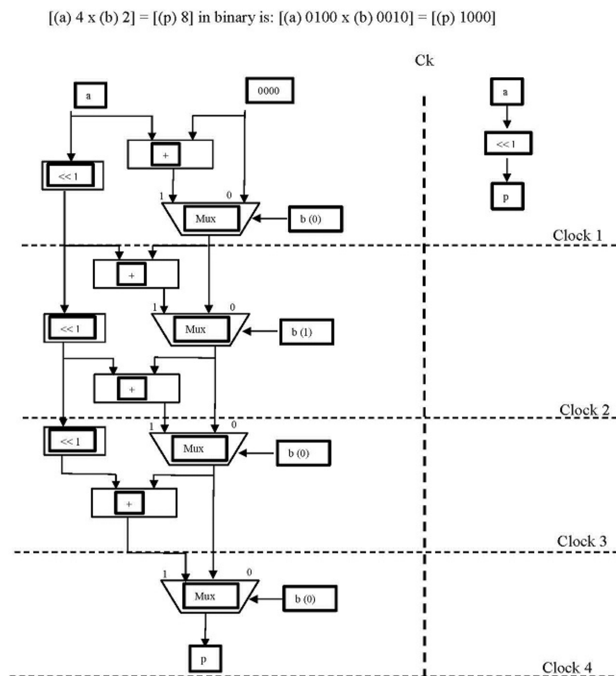


FIGURE 2.5: Hardware synthesis and optimisation hardware synthesis. It show two technique to implement a multiply. The first architecture uses 4 clock cycle, while the optimize one uses just a clock cycle

the voltage; reducing the voltage will significantly reduce the power consumption of the chip. In spite of this, controlling the frequency can also have a relevant impact on energy generation. According to the equation, decreasing the frequency, by clock gating, or powering off a complete area of the chip, by using power gating technique can improve the energy efficiency of the architecture. The power gating technique ensures that at any point in time only the needed logics are on and the others off "in idle state, or completely off". For instance these techniques give a high energy efficiency if applied to CoDa architecture, because it can enable On/Off or power On/Off each co-processor.

For a better understanding about dynamic power, refer to Figure 2.6. The dynamic power is generated from two factors: charging and discharging load capacitances as a gate switch, RC characteristic and switching transition, in the process which sees pMOS switching ON and nMOS switching OFF, for a short time both are ON. In more detail, when the voltage of the of the pMOS and nMOS is switched from 1 to 0, for a short time the current flow from VDD to GND will be biased in the linear region and generate capacitance on the output bus for both transistor configurations.

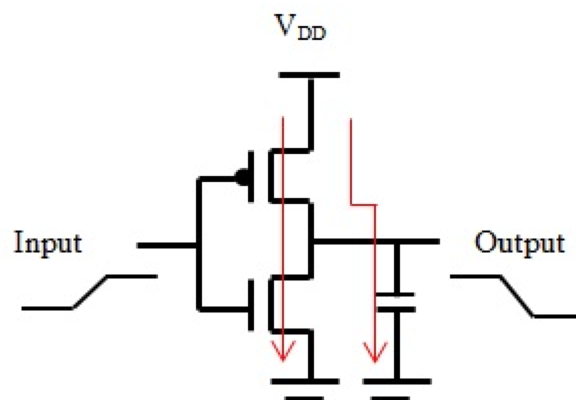


FIGURE 2.6: It shows in the stage of switching between zero to one for a short time PMOS and NMOS are in saturated region so the current flow from VDD to GND. But also, on the output there is a charging and discharging capacitance [34].

When the logic is not on the operative mode the CMOS technology is affected by static power. Basically it is generated by three elements: gate leakage, drain junction leakage and sub-threshold current Figure 2.7

- Sub-threshold current happens because there is conduction between the source and drain. In other words, they are too close to each other and so with the scaling down of the transistor the gap becomes narrow, and as a consequence the leakage increase [34].
- Gate leakage arises when voltages are applied on the gate and basically depend on the dielectric thickness. In more detail, the electron could be found outside its orbit, therefore it could be situated on the wrong side of the oxide [34].

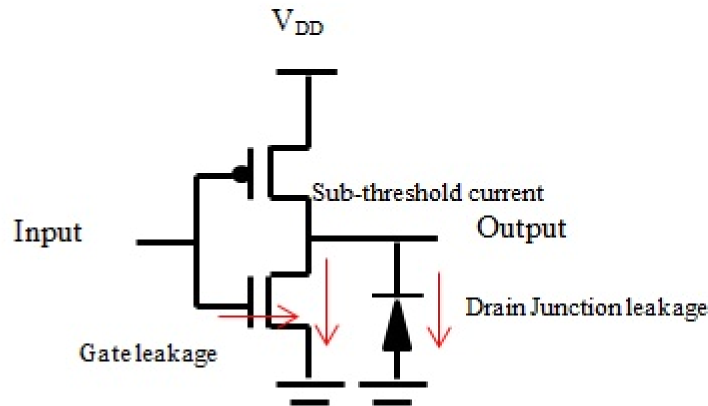


FIGURE 2.7: CMOS static power is generated: gate leakage, it means when the voltage is applied on the gate, the electron could jump and land on the wrong side of the oxide. Drain junction leakage is formed when the source or the drain is at different potential and finally the sub-threshold current is generate from a connection between the source and the drain [34].

- Drain junction leakage arises when either the source or drain are at different potential from the substrate [34] .

Understanding how power consumption acts inside a chip helps in the design design of a valuable low-power system. Figure 2.8 shows how the transistor acts in the different phases of operations. It is now clear, that when the transistor is in switch off mode current still flowing.

To measure the power consumption inside the microcontroller, we can sub-divide the circuit in many subsystems and analyse the power of each system. In this way we can understand and optimise each unit Figure 2.9 shows an example of power consumption inside the Sun's 8-core 84 W Niagra2.

2.3 Design Flow

So far dynamic and static power has been explained and it has demonstrated which unit of the CPU architecture consumes more power. This consideration has helped to identify and break down into different stages some processes for improving power consumption. Before going into detail, it must be clear that the goal of this research is focused on generating accelerators for a Stack CPU Architecture. To deal with Dark silicon power consumption must be considered as a priority. Also considerable attention must be paid to performance and advancements for the design cycle as a valuable variable that enables the design to be competitive and applicable on the market. The procedure below explains the methodology for dealing with Dark Silicon with reference to these variables. **Procedure:**

- **Translation methodology:** The aim of this project is to convert software into hardware. In order to investigate how power, area and timing behave, a tool has been built in ANSI C to convert software into hardware VHDL. The tool gen- erates the VHDL code into two different architectures called "Composite

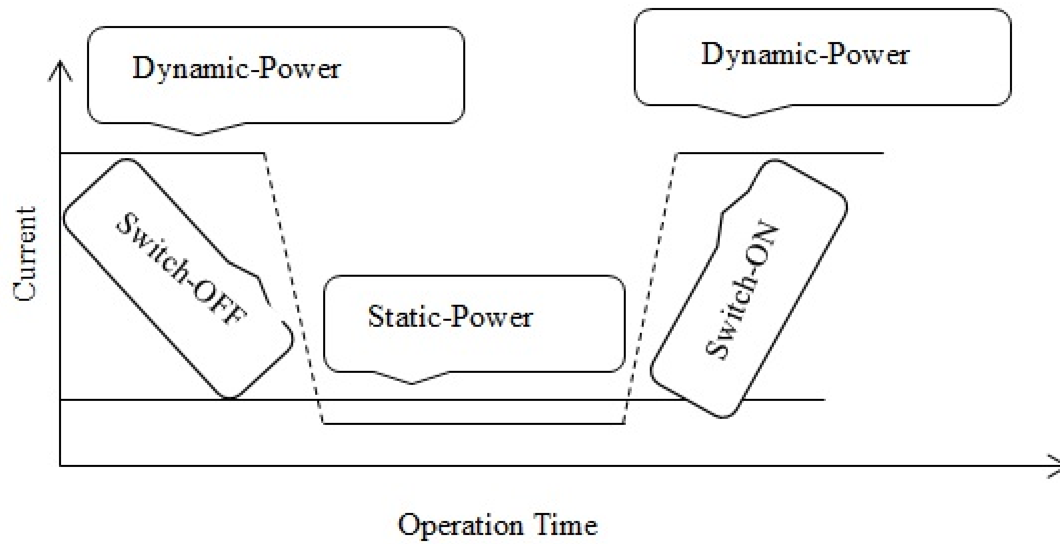


FIGURE 2.8: Power consumption phases. It shows the different stage of power. When the transistor is in operative mode uses dynamic power, the current arises. When the transistor is off, we call static power, the current decreases but do not become zero. It means that in this state, the transistor still uses current.

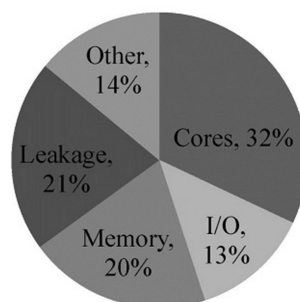


FIGURE 2.9: Power consumption in Niagra 2. Most of energy is used by Cores, Leakage and Memory [23] .

and Wave-core". Composite architecture, is implemented using a more traditional way to implement a finite state machine;, it tries to optimize the area at cost of increasing power density. In contrast Wave-core architecture, an evolution of composite architecture, tries to expand the area to improve the power density. Also it has a better control of granularity giving the opportunity to improve static power. (See section 2.6 for more details). To investigate both composite and wave-core architectures the translator tool is capable of automatically producing thousands of hardware functions, co-processors, from the software function which can then be analysed.

- **Function validation:** To observe how the architectures behave in term of power, area and timing, a set of source codes, software benchmarks, is selected and converted by the translator tool into the two different hardware architectures, above mentioned. Also, it will introduce possible VLSI solutions to improve the performance of the architecture.

- **Instruction level parallelism:** One of the key ideas of this architecture is to generate optimised hardware directly at the middle level of the compiler, . This chapter focuses on static parallelism and observes how implemented compiler optimisation techniques, such as scheduling, impact the hardware function. (See Chapter 5 for more details).

2.4 Accelerator generation; design flow

As with all hardware designs, it is important to consider the trade-offs between dynamic and static power consumption, timing and area. In order to observe in detail thousands and thousands of hardware units, a translator tool was developed in (ANSI C) to map machine code fragments (typically basic-blocks) into hardware structures. This involved translating machine code generated by a Stack machine C++ compiler [56] into VHDL, and subsequently synthesising to FPGA and ASIC targets. The translation tool is able to generate two architectures, called Composite and Wave-core.

The translator tool was developed with the idea of translating into hardware the most energy demanding functions and helps the design engineers to estimate the best trade-offs according to power, area and timing, and then select the most efficient hardware units to be interconnected to the architecture. While this methodology may be applied in a future CPU design, at this stage the tool is simply used to translate software into hardware and generate a high amount of data to be analysed. Figure 2.10 describes the top level of the process methodology. It explores all stages of the design from the machine code until the IP core is generated.

One of the advantages of this method is that the engineer can easily translate and observe thousands of units in terms of area, power and time and then select the best unit to be interfaced to the CPU.

Returning to Figure 2.10, the closed loop jumps into each stage from software function to IP hardware core, before the process is completed. The methodology gives the opportunity to translate the software into hardware; it generates two types or architectures to be considered, also it generates a vector test bench to test both architectures and a VCD file to record the logic analysis. Finally it generates an excel file with the power, timing and area data to be analysed. Another point to elucidate is that the methodology is fully automated. At first the translator tool imports a file

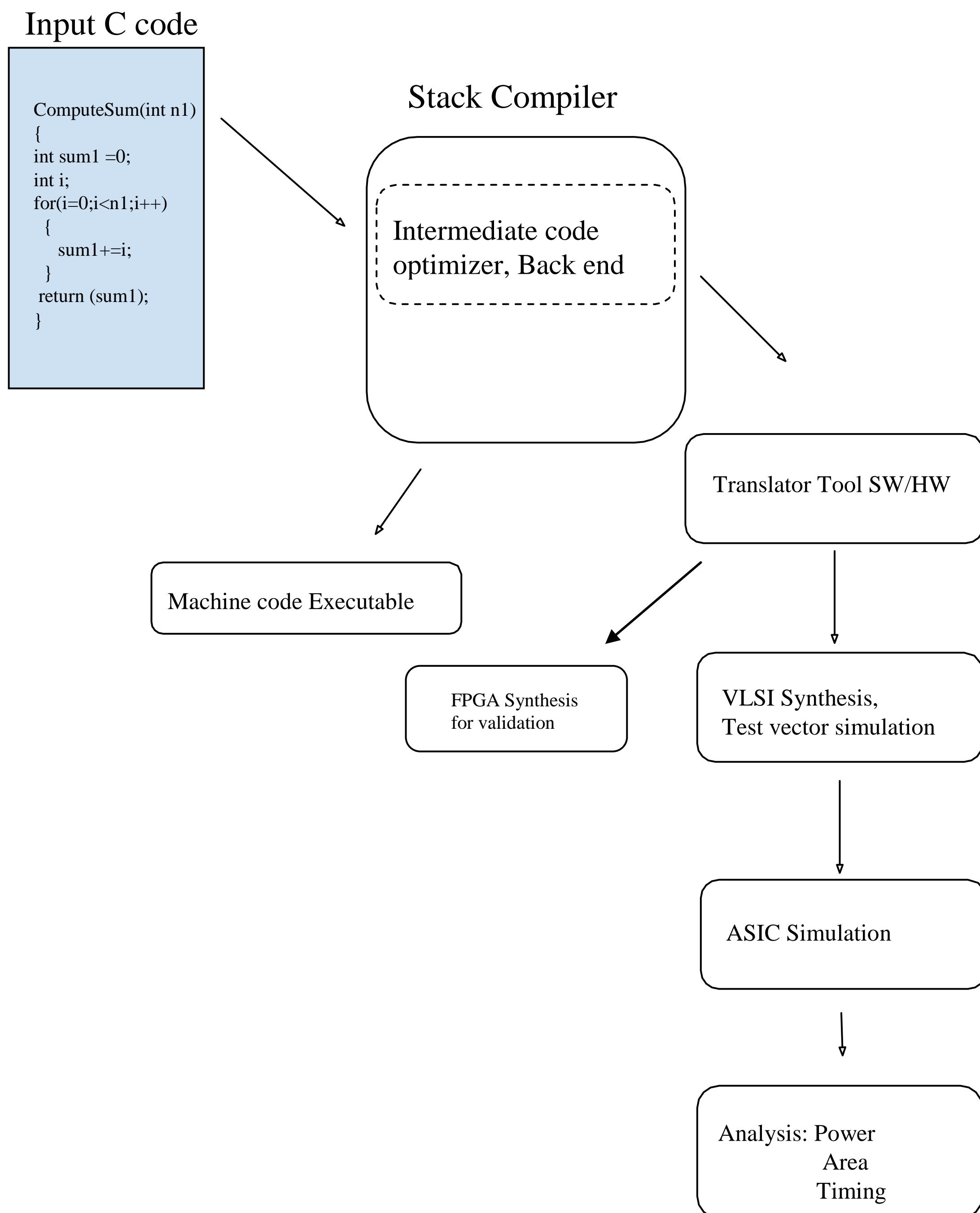


FIGURE 2.10: Top level of the tool chain methodology. Starting from the intermediate machine code, the translator tool generate to topologies of VHDL. Then the IP is ready to be synthesised in FPGA and VLSI to analyse the timing, area and power.

and translates it into 2 VHDL topologies of architectures. Similarly, it generates a Verilog vector test bench for each VHDL file, because just the Verilog generates the VCD file. The translator tool also generates a csv file for each of the IP cores that reports the numbers of states, the numbers of inputs/outputs, the instruction per cycle (IPC), and the cycle of instruction (CPI). After that a TCL script imports each VHDL file into Cadence ICTM Suite for ASIC 65nm UMCTM; the script adds the Verilog test bench. files, a memory vector to simulate a real communication between logic and memory, and then synthesises each core. At the end of the synthesis, cadence generates the RTL schematic, and VCD file which includes the timing, power and area report. At this stage the script built in Python reads each report, extracts the value of power, area and timing and then creates a csv file to be read by Microsoft Excel. At this point the IPs unit can be observed
Figure 2.10 in more detail:

- **Input C code, Stack Compiler:** The methodology starts with a file already converted into software Stack assembly architecture using a Stack Compiler [56]. The assembly file is ready to be translated into VHDL.
- **SW/HW translator VHDL:** At this stage the tool translates the software into hardware. It generates two topologies of architectures, called Composite and Wave-core in VHD, a Verilog test bench file and a csv file that update the hardware characteristic: Input/Output, IPC, CPI, number of states in the state machine. (See section 2.6 for more details)
- **FPGA validation:** To test the VHDL generated from the translator tool, the methodology synthesises the IP first in FPGA and then if no error occurs in VLSI
- **Logic Synthesis, Test vector simulator:**To understand if the hardware core performs the right solution. The IP generated from the translator tool is Synthesised in FPGA using Xilinx ISE Design Suit 14.2. The translator tool also generated a test vector to test the architectures.
- **ASIC Simulation:**To observe how the hardware cores behave when they are implemented in ASIC platform, the cores are synthesized using Cadence Suite on a 65nm node CMOS device; the tool is able to generates a Register-Transfer Level and a txt file that reports the Timing, Power and Area (see appendix D, E, F for the report).
- **Analysis: Power, Timing and Area:**Finally a script in Python extracts the value of timing, power and area, from the txt file and generates a csv file. At this stage it has all the characteristics of the core that enable us to understand how it behaves and if it could be a possible candidate to be interconnected to the host CPU and improve the power of the overall device.

2.5 SW/HW Function translation

The translation tool was developed in (ANSI C) to map machine code fragments (typically basic-blocks) into hardware structures. This involved translating machine code generated by a Stack machine C++ compiler [35] into VHDL, and subsequently synthesising to FPGA and ASIC targets. The translation tool is able to generate two

types of finite state machine here called Composite and Wave-core (see section 2.8 & 2.9 for more details). It is notable that many compilers use a Stack based intermediate compilation model prior to registering mapping, and therefore our approach is essentially very similar to taking such intermediate representations and translating them into hardware [52]. Figure 2.11 shows an example of a high-level data-flow structure of the core generated from a low-level machine code sequence. In more detail on the left there is a code sequence generated by a Stack machine compiler [35], in the middle there is a segmented version of the same code, and to the right the graphical data-flow representation.

The segmentation of the code is achieved by identifying load and store operations (@loc and !loc tokens in this case) as the data-flow 'checkpoints'. m1, m2 and m3 represent memory accesses. One important characteristic of such Stack architecture is that it acts destructively on the data at the top of the Stack: operands are popped and results pushed by successive actions. This avoids any need to use operand mapping (e.g. to intermediate register identities) in the translation process. However, one challenge introduced by this approach is that, taking a block in isolation, one cannot immediately know what inputs and outputs there are to this block as a unit. To deal with this problem reiterative backtracking is used, whereby the Stack content is tracked instruction by instruction, and the number of initial inputs (values present on the Stack before execution) and outputs (values remaining after execution), can then be determined.

Backtracking operates as follows: Each instruction has a known Stack effect, it means each command requires a specific number of variables to push or pop from the stuck such as; "ADD requires two value from the stack and gives back just one value, load from memory "@LOC" needs just one value from stack without returning a value, Figure 2.12. The program code sequence is analysed and the local Stack depth is calculated, initially assuming it to be zero (i.e. zero inputs to the code block). This is illustrated in 2.13. At some point the Stack depth tracking either reaches the end of the sequence with a positive or zero value, or it encounters a negative Stack depth during the tracking traversal. When the Stack depth is negative, the initial Stack depth is increased and the analysis restarts. Eventually, the Stack depth will be zero or positive on the nth reiteration, and the backtracking is complete. The required initial Stack depth represents the input operands to the core and the final Stack depth represents the results available after core execution is completed. In a practical implementation, these inputs and outputs must be transferred to/from the core by the host CPU, most likely via a bus of communication.

Once the code block has been analysed using the backtracking algorithm, the block can be generated such that a known number of input operands are clocked into the block from the host CPU before beginning the core operation. Once the core reaches completion, it will generate a known number of results to be returned to the host CPU.

Formal translation of the core is relatively straightforward. In this implementation, each instruction in the sequence has an associated VHDL statement or statements that define the required behaviour. This could equally be a Verilog coding if desired. For example, the XOR instruction causes the top two Stack operands to be logically combined and an equivalent VHDL statement would be as follows: Figure 2.14

Note that in most Stack models, this is a destructive operation, so the remaining underlying Stack operands pop by one position, as shown. The movement of Stack contents of the form Sx := Sy corresponds to a redundant operation in a hardware structure – it is simply a point to point wire. This means that the apparent effort

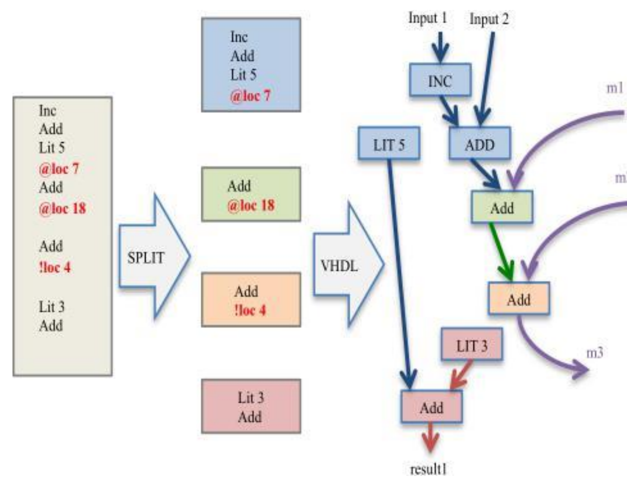


FIGURE 2.11: High-level data-flow relationship of the core. Starting from left there is a middle level machine code based generated from the compiler [35]. In the middle there is a subdivision version of the code to recognize any memory load and store and finally on the right there is the graphical data flow representation..

Command	PUSH	POP
Lit 1	0	0
@loc	0	1
Add	2	1
Sub	2	1
!loc	1	0
Nop	0	0

FIGURE 2.12: Stack effect of instruction relevant example. In more detail, Lit means literal. It is the value itself, @Loc means load the value from location, Add means add two value, Sub means subtract two value, !loc means store the value in memory and Nop means do nothing.

Command	Push	Pop	Pass1 (depth=0)	Pass2 (depth=1)	Pass3 (depth=2)
Nop	0	0	0	1	2
Add	2	1	-2	-1	1
Lit5	0	1	restart	restart	2
@Loc7	0	1			3
Nop	0	0			3
@Loc	0	1			4
Add	2	1			3
Nop	0	0			3
Nop	0	0			3
@	1	1			3
Add	2	1			2(done)

FIGURE 2.13: Backtracking algorithm example. It shows how the tool apply the backtracking algorithm to count the Input/Output of the core

S0 := S0 XOR S1;

S1 := S2;

S2 := S3;

....

Sn-1 := Sn;

FIGURE 2.14: It is a destructive operation. In most stack models operation becomes redundant so the hardware synthesis destructs redundant operation.

involved in maintaining the Stack order is actually purely notional and not computational. The translation tool has been designed to generate code that is portable between XILINX FPGA toolsets and CADENCE VLSI toolsets. The tool has been tested successfully with both platforms and therefore permits both an FPGA and VLSI approach to targeting of the cores to actual hardware. The translation tool also auto-generates Verilog test benches for each core, and outputs scripts for performing synthesis and power analysis. (See section 2.5.1, 2.5.2 & 2.6 for more details).

2.5.1 The Translator tool in detail

The translator tool to convert software into hardware uses a Finite State Machine (FSM), Figure 2.15

To build the generic FSM, at first stage the algorithm detects all data symbolised by (@loc and !loc), which in the assembly Stack code represent the load and store operations, and so at every @loc and !loc the tool considers them as a state. After that, it uses a look up table technique to record how many variables needs to push or pop to the stack Figure 2.13 . At this stage it applies the backtracking algorithm to count how many values needs to push or pop from the stack. The push on the stack is associated with the inputs and the pop from the stack with the outputs of the IP hardware unit, as is shown in Figure 2.14. The backtracking algorithm is a technique that methodically looks across all variables to find the solution to the problem. It approaches the problem by grouping all values in vectors (V1, V2 ... Vn) and then jumping inside every value to find the solution. It first initiates an empty vector that fills with partial values, and then goes forward and back until it can produce the result [15]. Finally, with the number of states and the number of inputs/outputs of the entity unit known, the software command simply needs to be translated into a hardware command, which will be done directly by associating the VHDL statement. (See section 2.5.2 for an example of VHDL code). The structure of the VHDL code is implemented with the characteristic to be a push-down Stack. With this in mind, it now becomes easy to understand that the name Stack machine come from the fact that the Stack machine implements a register with Stack. The operands are always on the top of the Stack and the result is stored in the top register of the Stack. It uses a Last-in, First-out to hold short-lived temporary values. Despite this, the logic generated from the tool chain is a pure data-flow structure, therefore neither a Stack nor a register file. In order to understand why the first approach implemented was the core in a Stack machine instead of traditional architecture of such pipeline the following factors need to be considered:

- The Stack machine is a simpler architecture, because it execute just the value on the top of the stack, it is simpler to implement (Note section 2.5)
- Stack can execute the interrupt function directly, as hardware makes sure of the Stack management. In other words because the stack just execute values from the top of the stack, the interrupt function is always placed on the top of the stack.
- The Stack does not need a temporary register, therefore there is no need to store data to memory across procedure calls.
- Compilation processes make it a relevant choice because it becomes easy translate SW/HW. (Note section 2.5).

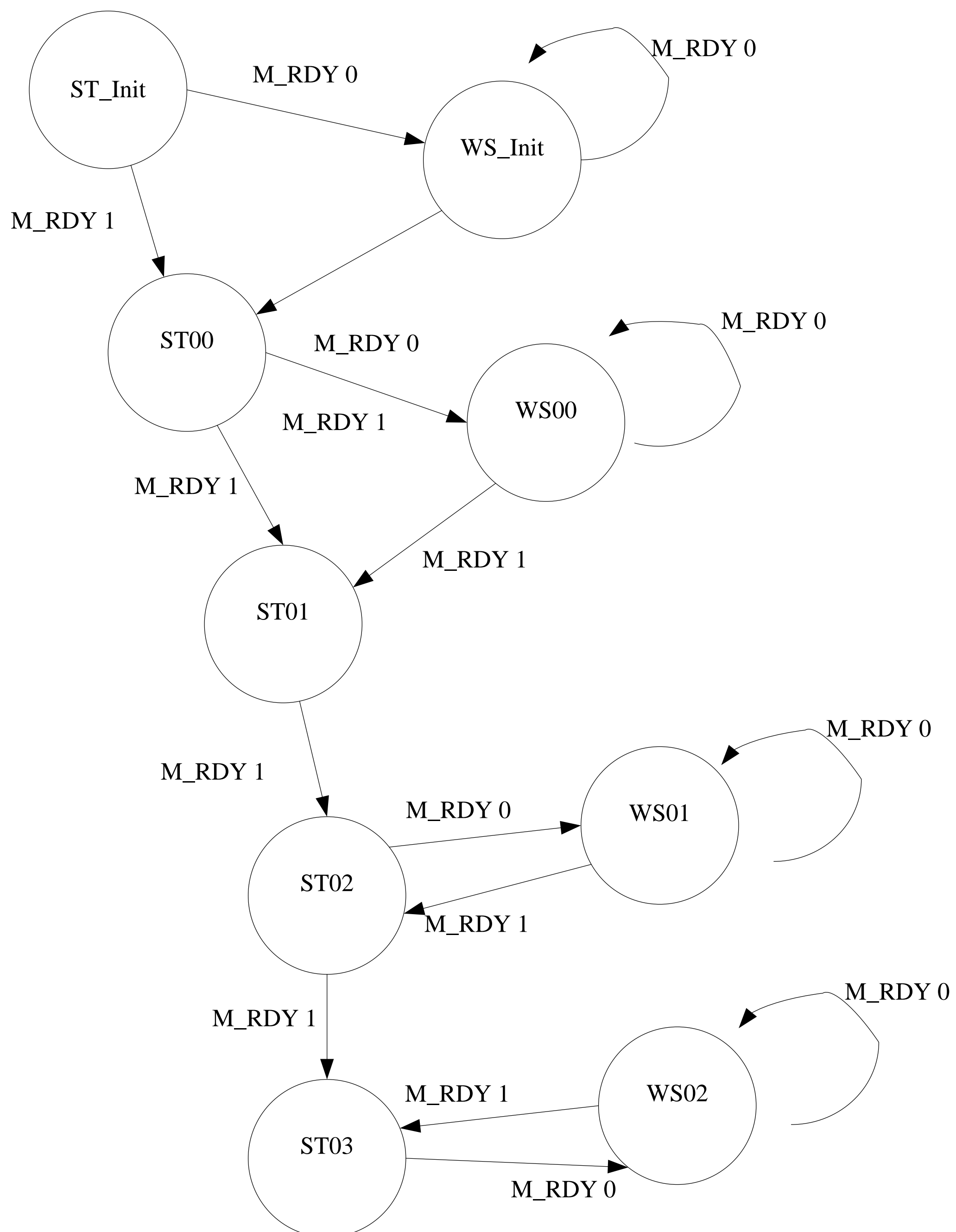


FIGURE 2.15: Generic finite state machines. The FSM starts with ST Int state, it symbolises the initial condition. If the memory ready input is low (M RDY) then it jumps in the dedicate wait memory event (WS Init). It stays there till the memory finish to read the value from the bus or write the value on the bus to serve next state. If the M RDY is high then it jumps directly in the next state.

- The rationale is: if most work is done by special cores, then why a needed of a more complex CPU when a Stack CPU can do the job? It must be remembered; Stack architecture has been ignored for the poor performance in reordering data on the stack which makes it appears to be a slow architecture. However it has had always a good reputation for consuming a small amount of power.

2.5.2 Testing the translator tool

The schematic in Figure 2.16 describes the testing methodology of the tool chain to demonstrate that both architectures do the same work. It explores all stages of the design from machine code until IP core generation.

The technique used is the same as used in the industry today. Starting from the top, there is the design entry, the software function to be translated into hardware. The translator tool generates two VHDL topologies of architectures. They are called Composite and Wave-core. To prove the translator tool creates an executable VHDL file and both Composite and Wave-core architectures perform the same result, the VHDL generated is synthesised using Xilinx ISE Design suit 14.2. If no errors occur then the codes will be tested using a VHDL vector test bench. Alternatively, if an error occurs in both synthesis level and tested level, the error will be detected and recorded to modify the tool generator. After that the same code will be tested using CADENCE VLSI toolsets. If no errors occur then the IP is ready to be implemented. Conversely, the error/s will be recorded to debug the tool generator. At the end of the process CADENCE generates the power, timing and area report to be analysed.

To understand better how the translator tools works and analyse how it translates stack machine assembly code into hardware Figure 2.17 shows a typical example

On the left side an example of assembly code is showed. In more detail the first two operations are a copy of the variables; therefore there is a stack reordering. Note, the variable temp stores at first instance the value in the process. Then the add operation sums the two values on the top of the stack memory, following a reordering of the stack memory using a rotate command. RSU3 rotates the three values of the stack. To clarify, the third value of the stack memory becomes the first and as a consequence the first value becomes the second and finally the second value becomes the third. To conclude there is the subtraction computation following the division. To generate IPs core VHDL at first it splits the software function in many states and then it is reordered in a stack manner. Finally it directly translates the software instructions into VHDL instructions. Before explaining the VHDL code, the two architecture are here explained in more details.

The translator tool generates two types of architecture called Composite and Wave-core. Figure 2.18 shows an example of Composite architecture.

To understand in more detail how the translator tool generates the Composite architecture and to highlight the differences between the Composite and the Wave-core the piece of assembly machine code on the left side of Figure 2.18 will be analysed. The different commands have already been explained but in this example a memory access is added, @loc7,. Basically the memory access symbolizes the start of the new state as explained in section 2.5.1 and it is implemented in a single VHDL process.

Figure 2.19 Figure 2.20 Figure 2.21 show the VHDL code for the Composite architecture. Attention should be paid to two points: firstly how at every access of memory, @loc 7, the translator tool generates a new state and secondly how the state

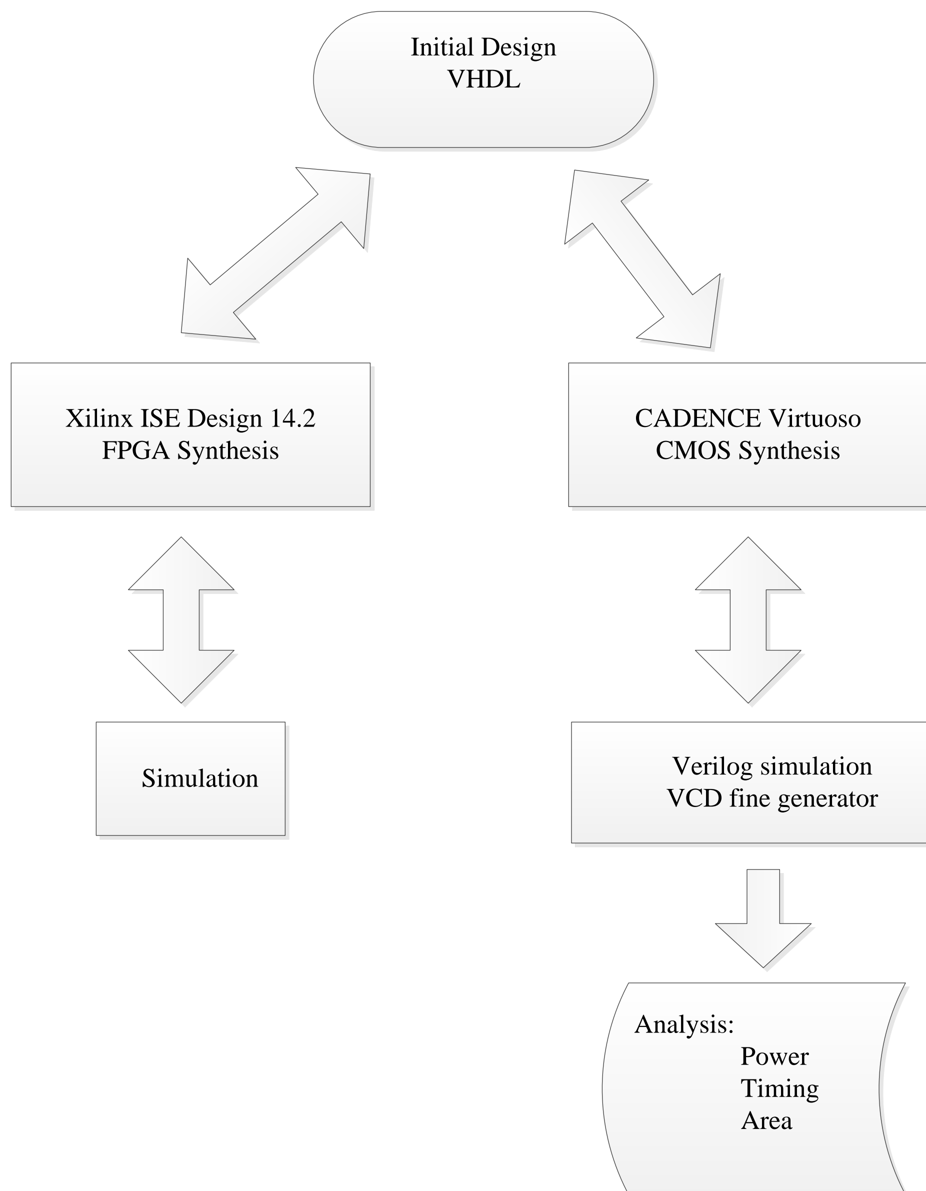


FIGURE 2.16: Top level of testing methodology. It starts from the initial design, VHDL code, generated from the translator tool to be synthesised using Xilinx ISE design 14.2. If the code generated from the translator tool has not syntax error, then the code is simulated and tested using a VDHL test bench. Otherwise, it backs in the initial design and wait for the new source, VHDL code. After the code has been tested successfully in Xilinx, it will be synthesis and tested in Cadence. At this stage the test bench is built in Verilog, because it generates the VCD file that report the behavioural of transistors in term of power, area and timing.

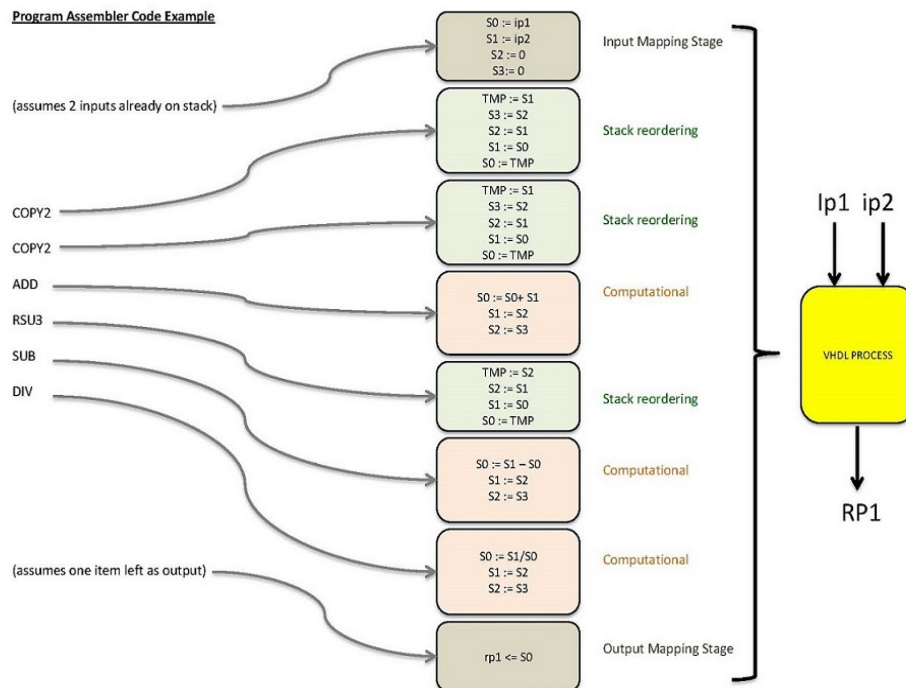


FIGURE 2.17: program assembly code example. On the left side there is an example of assembly function code. In the middle assuming two inputs on the top of stack, follow to copy instructions then the function compute and add operation, rotate the values to be reordered and perform a subtraction and a division. Finally, the code is directly translated in VHDL syntax to generate the hardware IP.

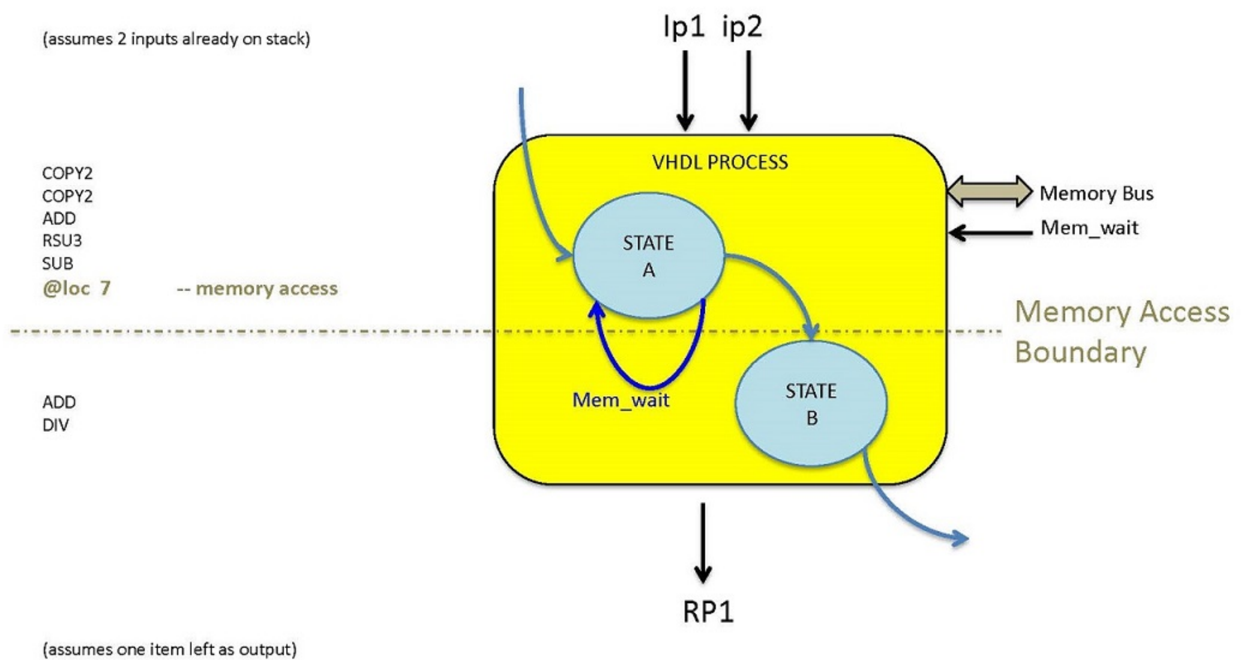


FIGURE 2.18: shows a composite architecture, on the left side there is an assembly stack machine code. Inside the function there is a memory access by @loc7. The code is directly portable in VHDL code using a state machine implementation generate the hardware IP.

Composite.vhd **Sun Oct 16 17:23:43 2016**

```
1    LIBRARY ieee;
2    USE ieee.std_logic_1164.all;
3    USE ieee.numeric_std.all;
4    USE ieee.std_logic_unsigned.all;
5
6    ENTITY darc1 IS
7     PORT (
8            i00 : IN std_logic_vector(31 DOWNTO 0);
9            i01 : IN std_logic_vector(31 DOWNTO 0);
10           i02 : IN std_logic_vector(31 DOWNTO 0);
11
12           r00 : OUT std_logic_vector(31 DOWNTO 0);
13
14   FP : IN std_logic_vector(31 DOWNTO 0);
15   FPout : OUT std_logic_vector(31 DOWNTO 0);
16           M_ADDR :   OUT std_logic_vector(31 DOWNTO 0);
17           M_DATA : INOUT std_logic_vector(31 DOWNTO 0);
18           M_RD  : INOUT std_logic;
19           M_WR  : INOUT std_logic;
20           M_RDY : IN std_logic;
21           reset : IN std_logic;
22           CLK   : IN std_logic
23         );
24   END ENTITY;
25
26   ARCHITECTURE yy OF darc1 IS
27     TYPE States IS (ST_INIT,WS_INIT,ST_RESET,ST00,WS00,ST01,WS01,ST_END);
28     SIGNAL Mstate : States;
29   BEGIN
30
31
32   -- CONTROL PROCESS --------
33    PROCESS(clk,reset)
34    BEGIN
35    IF reset='1' THEN
36       Mstate <= ST_RESET;
37      ELSIF(rising_edge(clk)) THEN
38    CASE Mstate IS
39      WHEN ST_RESET => Mstate <= ST_INIT;
40      WHEN ST_INIT =>      IF M_RDY='1' THEN Mstate <= ST00; ELSE Mstate <= WS_INIT; END
     IF;
41
42      WHEN WS_INIT =>      IF M_RDY='1' THEN Mstate <= ST00; END IF;
43
44         WHEN ST00 =>  IF M_RDY='1' THEN Mstate <= ST01; ELSE Mstate <= WS00; END IF;
45
46         WHEN WS00 =>  IF M_RDY='1' THEN Mstate <= ST01; END IF;
47
48      WHEN ST01 | WS01| ST_END =>   WHEN OTHERS =>
49
50    END CASE;
51   END IF;
52    END PROCESS;
53
54
55   -- EXECUTE COMP PROCESS --------
56    PROCESS(clk,reset)
```

Page 1

FIGURE 2.19

Composite.vhd **Sun Oct 16 17:23:43 2016**

```
 57      VARIABLE T,s0,s1,s2,s3,s4,s5,s6,s7,fpi :std_logic_vector(31 DOWNTO 0);
 58     BEGIN
 59       IF(reset='1') THEN
 60        -- reset any internal states --
 61
 62        s0 := (OTHERS=>'0');
 63        s1 := (OTHERS=>'0');
 64        s2 := (OTHERS=>'0');
 65        s3 := (OTHERS=>'0');
 66        s4 := (OTHERS=>'0');
 67        s5 := (OTHERS=>'0');
 68        s6 := (OTHERS=>'0');
 69        s7 := (OTHERS=>'0');
 70     fpi:=(OTHERS=>'0');    -- ref E --
 71     M_ADDR <= (OTHERS=>'Z');
 72     M_DATA <= (OTHERS=>'Z');
 73     M_RD <= 'Z';
 74     M_WR <= 'Z';
 75       r00 <=(OTHERS=>'0');
 76
 77       ELSIF(rising_edge(clk)) THEN
 78        M_DATA <="ZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZ";
 79     CASE Mstate IS
 80        WHEN ST_INIT =>
 81       -- COMP. connect 3 input params here --
 82         s0 := i00;
 83         s1 := i01;
 84         s2 := i02;
 85       fpi := FP;
 86     --copy2 ;
 87      T:=s1 ;
 88          s7 := s6;
 89          s6 := s5;
 90          s5 := s4;
 91          s4 := s3;
 92          s3 := s2;
 93          s2 := s1;
 94          s1 := s0;
 95      s0:=T ;
 96     --add ;
 97      s0:=s0+s1 ;
 98          s1 := s2;
 99          s2 := s3;
100          s3 := s4;
101          s4 := s5;
102          s5 := s6;
103          s6 := s7;
104     --rsu3 ;
105       T:=s0 ;
106      s0:=s1 ;
107      s1:=s2 ;
108      s2:=T ;
109     --sub ;
110      s0:=s0-s1 ;
111          s1 := s2;
112          s2 := s3;
113          s3 := s4;
```

Page 2

FIGURE 2.20

```
Composite.vhd                                        Sun Oct 16 17:23:43 2016
114         s4 := s5;
115         s5 := s6;
116         s6 := s7;
117
118       --M> @loc 7
119         M_ADDR <= std_logic_vector(to_unsigned(7,32))+fpi;
120         M_RD <='1';
121         M_WR <='Z';
122
123       WHEN ST00 =>
124         s7 := s6;
125         s6 := s5;
126         s5 := s4;
127         s4 := s3;
128         s3 := s2;
129         s2 := s1;
130         s1 := s0;
131
132
133          s0 := M_DATA;
134          M_RD <='Z';
135          M_WR <='Z';
136
137    --add ;
138     s0:=s0+s1 ;
139         s1 := s2;
140         s2 := s3;
141         s3 := s4;
142         s4 := s5;
143         s5 := s6;
144         s6 := s7;
145    --div ;
146     s0:=s1/s0 ;
147         s1 := s2;
148         s2 := s3;
149         s3 := s4;
150         s4 := s5;
151         s5 := s6;
152         s6 := s7;
153      -- recover 1 results here --
154        r00 <= s0;
155      FPout <= fpi;
156
157       WHEN OTHERS => s0 := s0;
158
159      END CASE;
160      END IF;
161
162    END PROCESS;
163
164   END ARCHITECTURE;
165
166

                                  Page 3
```

FIGURE 2.21: shows the VHDL for the composite architecture. Note at command @loc 7, the translator tool generated a new state with a communication of memory.

machine is implemented using a single process. In this way it optimizes the area at the cost of increasing the power density and increases the possibility of generating a hotspot as will be explained in the next chapters.

On the other hand there is the Wave-core architecture. It has evolved from the Composite, in a way to be implemented using a multi processes very similar to a systolic fashion. It means: every state is implemented using dedicated hardware or is an easy description using a dedicated process. Each process is interconnected using signals Figure 2.22

To understand better the VHDL code for the Wave-core, it analyses the Figure 2.23, Figure 2.24, Figure 2.25, Figure 2.26 and Figure 2.27

The main differences between the Composite and the Wave-core could be explained in the VHDL code Figure 2.23, Figure 2.24, Figure 2.25, Figure 2.26, Figure 2.27 show how the Wave-core is implemented and two significant pieces of evidence emerge; first any state is implemented using a dedicated process, it acts as a dedicated hardware. Second each process is interconnected using a set of signals and so in a systolic model. In this way the architecture expands the area and as a benefit improves the power density and reduces the possibility of generating hot spot as will be explained in the next chapters. At every state there is a memory access boundary. This helps to isolate each process and by applying power gating technique improves the static power, as will be demonstrated.

Finally, the Wave-core architecture appears to increase the cost of chip manufacture because it increases the area. However, the main priority in design CPU today is its power. Therefore the increasing of cost in manufacture will be absorbed by generating more power efficient CPU architecture. A similar effect has been observed in the manufacture of faster CPU architecture. At each new chip generation the increase of cost in manufacture was absorbed by CPU speed performance [2] [63].

2.6 The Verilog test bench

The translator tool is capable of auto-generating the Verilog test bench for each core. (See Appendix C for an example of Verilog code.) The advantage of using Verilog instead of VHDL is that it can generate the Value change dump (VCD) file. The VCD file holds information about any value changes on the selected variables; therefore, it is an invaluable instrument to monitor, record and observe how hardware behaves in respect of timing, area and power.

Before beginning to build the test bench we must understand the switching power. We have explained the CMOS transistor and its bipolar configuration (NPN, PNP), when a signal changes its state from 0 logics to 1 and from 1 to 0 logics the output capacitance changes from charging to discharging and vice versa. The energy on transition could be described as: see Figure 2.28

According to Figure 2.28 a test bench that could group the least and most changing switching transistor to obtain an occurrence estimation is needed.

The first issue to build a test bench was to describe 4294967296 = 2^{32} combinations using just a few patterns. In other words, how to select the most and the least changing grouping patters from 4294967296. The solution to this problem was to consider that 2^4 goes into 2^{32} . Therefore, considering 2^4 the most changing switching transistors could be described in 16 test patterns. Figure 2.29

The test bench was built with the idea of monitoring the switching of transistors. Starting from the top of Figure 2.29; firstly, the transistors are always off, so no power is on; secondly, they start being off and on; thirdly, the situation is inverted

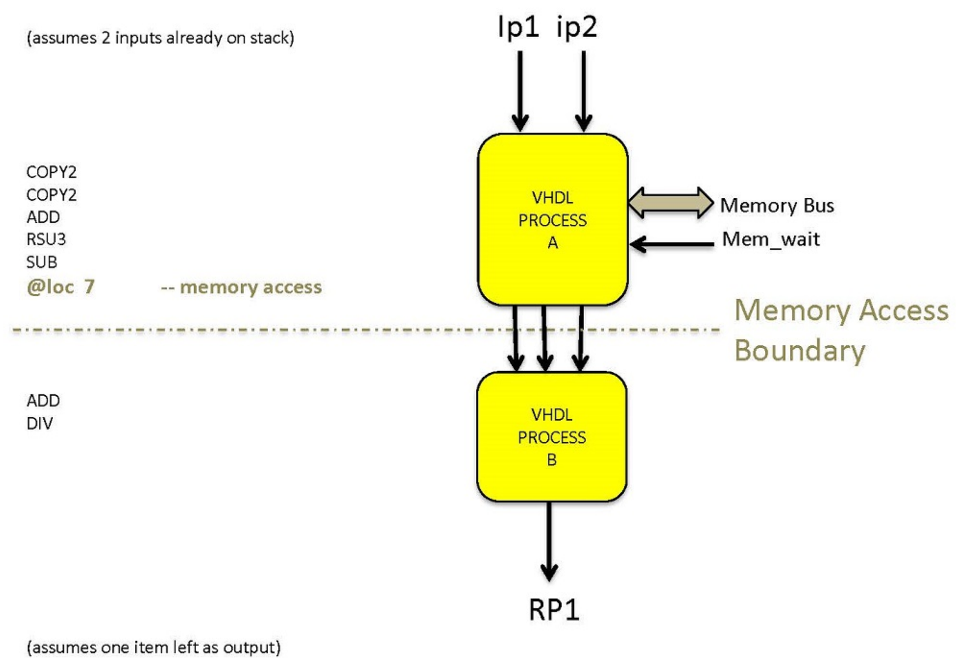


FIGURE 2.22: An example of Wave-core architecture. It is implemented using a multi processes approach in a systolic manner.

Wave-core.vhd **Sun Oct 16 17:59:00 2016**

```
1   LIBRARY ieee;
2   USE ieee.std_logic_1164.all;
3   USE ieee.numeric_std.all;
4   USE ieee.std_logic_unsigned.all;
5
6   ENTITY darc1 IS
7    PORT (
8           i00 : IN std_logic_vector(31 DOWNTO 0);
9           i01 : IN std_logic_vector(31 DOWNTO 0);
10          i02 : IN std_logic_vector(31 DOWNTO 0);
11
12          r00 : OUT std_logic_vector(31 DOWNTO 0);
13
14  FP : IN std_logic_vector(31 DOWNTO 0);
15  FPout : OUT std_logic_vector(31 DOWNTO 0);
16          M_ADDR :   OUT std_logic_vector(31 DOWNTO 0);
17          M_DATA : INOUT std_logic_vector(31 DOWNTO 0);
18          M_RD   : INOUT std_logic;
19          M_WR   : INOUT std_logic;
20          M_RDY : IN std_logic;
21          reset : IN std_logic;
22          CLK   : IN std_logic
23        );
24  END ENTITY;
25
26  ARCHITECTURE yy OF darc1 IS
27    TYPE States IS (ST_INIT,WS_INIT,ST_RESET,ST00,WS00,ST01,WS01,ST_END);
28    SIGNAL Mstate : States;
29  SIGNAL u0s0, u0s1, u0s2, u0s3, u0s4, u0s5, u0s6, u0s7, FPi0  : STD_LOGIC_VECTOR(31
    DOWNTO 0);
30
31  BEGIN
32
33
34  -- CONTROL PROCESS --------
35   PROCESS(clk,reset)
36   BEGIN
37   IF reset='1' THEN
38      Mstate <= ST_RESET;
39     ELSIF(rising_edge(clk)) THEN
40   CASE Mstate IS
41     WHEN ST_RESET => Mstate <= ST_INIT;
42     WHEN ST_INIT =>      IF M_RDY='1' THEN Mstate <= ST00; ELSE Mstate <= WS_INIT; END
    IF;
43
44     WHEN WS_INIT =>      IF M_RDY='1' THEN Mstate <= ST00; END IF;
45
46        WHEN ST00 =>  IF M_RDY='1' THEN Mstate <= ST01; ELSE Mstate <= WS00; END IF;
47
48        WHEN WS00 =>  IF M_RDY='1' THEN Mstate <= ST01; END IF;
49
50     WHEN ST01 | WS01| ST_END =>   WHEN OTHERS =>
51
52   END CASE;
53  END IF;
54   END PROCESS;
55
```

Page 1

FIGURE 2.23

_Wave-core.vhd Sun Oct 16 17:59:00 2016

```
 56
 57   -- WAVE EXECUTE PROCESS --------
 58
 59    proc_P00  :  PROCESS(clk,reset)
 60     VARIABLE T,s0,s1,s2,s3,s4,s5,s6,s7, fpi :std_logic_vector(31 DOWNTO 0);
 61    BEGIN
 62   IF(reset='1') THEN
 63
 64       -- reset any internal states --
 65
 66    fpi :=(OTHERS=>'0');
 67    FPi0 <=(OTHERS=>'0');  -- ref A --
 68       u0s0 <= (OTHERS=>'0');
 69       u0s1 <= (OTHERS=>'0');
 70       u0s2 <= (OTHERS=>'0');
 71       u0s3 <= (OTHERS=>'0');
 72       u0s4 <= (OTHERS=>'0');
 73       u0s5 <= (OTHERS=>'0');
 74       u0s6 <= (OTHERS=>'0');
 75       u0s7 <= (OTHERS=>'0');
 76       s0 := (OTHERS=>'0');
 77       s1 := (OTHERS=>'0');
 78       s2 := (OTHERS=>'0');
 79       s3 := (OTHERS=>'0');
 80       s4 := (OTHERS=>'0');
 81       s5 := (OTHERS=>'0');
 82       s6 := (OTHERS=>'0');
 83       s7 := (OTHERS=>'0');
 84    M_ADDR <= (OTHERS=>'Z');
 85    M_DATA <= (OTHERS=>'Z');
 86    M_RD <= 'Z';
 87    M_WR <= 'Z';
 88
 89
 90   ELSIF(rising_edge(clk)) THEN
 91
 92   IF(Mstate=ST_INIT) THEN
 93
 94      -- WAVE. connect 3 input params here --
 95        s0 := i00;
 96        s1 := i01;
 97        s2 := i02;
 98   fpi  := FP;
 99    --copy2 ;
100     T:=s1 ;
101         s7 := s6;
102         s6 := s5;
103         s5 := s4;
104         s4 := s3;
105         s3 := s2;
106         s2 := s1;
107         s1 := s0;
108     s0:=T ;
109    --add ;
110     s0:=s0+s1 ;
111         s1 := s2;
112         s2 := s3;
```

Page 2

FIGURE 2.24

```
Wave-core.vhd                                              Sun Oct 16 17:59:01 2016
113         s3 := s4;
114         s4 := s5;
115         s5 := s6;
116         s6 := s7;
117    --rsu3 ;
118      T:=s0 ;
119     s0:=s1 ;
120     s1:=s2 ;
121     s2:=T ;
122    --sub ;
123     s0:=s0-s1 ;
124         s1 := s2;
125         s2 := s3;
126         s3 := s4;
127         s4 := s5;
128         s5 := s6;
129         s6 := s7;
130
131       --M> @loc 7
132         M_ADDR <= std_logic_vector(to_unsigned(7,32))+fpi;
133         M_RD <='1';
134         M_WR <='Z';
135
136   -- generate output signals --
137
138       u0s0 <= s0;
139       u0s1 <= s1;
140       u0s2 <= s2;
141       u0s3 <= s3;
142       u0s4 <= s4;
143       u0s5 <= s5;
144       u0s6 <= s6;
145       u0s7 <= s7;
146   FPi0  <= fpi;
147
148   ELSE
149
150    M_ADDR <= (OTHERS=>'Z');
151    M_DATA <= (OTHERS=>'Z');
152    M_RD <= 'Z';
153    M_WR <= 'Z';
154
155
156
157   END IF;
158
159    END IF;
160
161
162   END PROCESS; -- proc 0
163   -----------------------------------------------
164    proc_P01  :  PROCESS(clk,reset)
165     VARIABLE T,s0,s1,s2,s3,s4,s5,s6,s7, fpi :std_logic_vector(31 DOWNTO 0);
166    BEGIN
167   IF(reset='1') THEN
168
169       -- reset any internal states --

                                   Page 3
```

FIGURE 2.25

```
Wave-core.vhd                                          Sun Oct 16 17:59:01 2016
170
171  -- #### States 1, s = 0
172
173     r00 <=(OTHERS=>'0');
174   fpi :=(OTHERS=>'0');   -- ref C --
175   M_ADDR <=  (OTHERS=>'Z');
176   M_DATA <=  (OTHERS=>'Z');
177   M_RD <= 'Z';
178   M_WR <= 'Z';
179
180
181  ELSIF(rising_edge(clk)) THEN
182
183      IF(Mstate= ST00) THEN
184   -- get input signals --
185
186      s0 := u0s0;
187      s1 := u0s1;
188      s2 := u0s2;
189      s3 := u0s3;
190      s4 := u0s4;
191      s5 := u0s5;
192      s6 := u0s6;
193      s7 := u0s7;
194  fpi  := FPi0;   -- ref D --
195
196
197  -- main dataflow sequence ----
198
199        s7 := s6;
200        s6 := s5;
201        s5 := s4;
202        s4 := s3;
203        s3 := s2;
204        s2 := s1;
205        s1 := s0;
206
207
208         s0 := M_DATA;
209         M_RD <='Z';
210         M_WR <='Z';
211
212   --add ;
213    s0:=s0+s1 ;
214        s1 := s2;
215        s2 := s3;
216        s3 := s4;
217        s4 := s5;
218        s5 := s6;
219        s6 := s7;
220   --div ;
221    s0:=s1/s0 ;
222        s1 := s2;
223        s2 := s3;
224        s3 := s4;
225        s4 := s5;
226        s5 := s6;

                                Page 4
```

FIGURE 2.26

Wave-core.vhd Sun Oct 16 17:59:01 2016

```
227         s6 := s7;
228
229
230    -- WAVE. generate final output signals --
231
232        r00 <= s0;
233    FPout <= fpi;
234
235    END IF;
236
237     END IF;
238
239
240    END PROCESS; -- proc 1
241    -----------------------------------------------
242
243    END ARCHITECTURE;
244
245
```

Page 5

FIGURE 2.27: shows a VHDL code of Wave-core architecture. The main characteristic is that he Wave-core is implemented using multiple process and are interconnected by signals

Energy/transition= $C*V^2$ Therefore:

Power=Energy/transition*f, it could be described us:

Power=$C*V^2*f$

FIGURE 2.28: It describes the equation of energy transition

0	0	0	0
0	0	0	1
0	0	1	0
0	0	1	1
0	1	0	0
0	1	0	1
0	1	1	0
0	1	1	1
1	0	0	0
1	0	0	1
1	0	1	0
1	0	1	1
1	1	0	0
1	1	0	1
1	1	1	0
1	1	1	1

FIGURE 2.29: It describes how select the most switching transistor in a 2^4 using 2^2 combination

and so now they are on and off; and finally they are always on. In this way the testing patterns show all variation of transistor power.

The second issue was to find an algorithm that could build fifteen test cases. Starting from the best solution, all switches are powered off. In contrast, as a last solution all switches are powered on. In the middle the switches should have the most alternating states.

To deal with the problem, the test case at first set up the transistors switching at zero. Then, it set up the last test case at one in order to describe the opposite solution. In the middle, to have the maximum switching system, it used the four-bit binary hypercube for finding Hamming distance. Hamming distance is usually used to detect errors; however, its characteristic of changing one bit per time could be used to described the best switching grouping patterns. See Appendix C for an example.

2.7 The Composite Cores Architecture

The tool chain generates two styles of architectures, called Composite and wave-core.

The 'Composite core' model is implemented using a classical Mealy-State Machine model. It uses a shared logic structure to process a sequence of instruction steps in hardware. It means that the synthesis tool tries to optimize the logic and reuse them as soon as they become available. In a sense this is analogous to a vastly simplified CPU, in that a shared computational structure is used in successive steps to apply a sequence of operations Figure 2.30 shows a typical abstract Composite core model and an example of state by state operation. The core has a distinct number of states, which correspond to the number of subdivisions of a code sequence that are identified by the translator tool. Each division is bounded by a memory reference (a load or store) since it is clear that the core must synchronise to memory contents in order to maintain coherency with the rest of the program execution history. More often than not these loads and stores refer to local variables in the cases examined based on a set of C++ benchmarks. The hardware core stalls when memory wait cycles are needed. Ideally, during such stall events the cores will only consume static power related to the retention of the current state of the core. Before conducting our experiments, we envisaged that Composite cores would be likely to be relatively compact structures, and as a result of the repeated use of shared logic, the power density of these cores would be high compared to alternatives.

2.8 The Wave-core Architecture

The Wave-core model treats each state in a computational flow as a separate hardware structure. Division into states is identical to that of the Composite core model, relating to load or store operations. However, the combinational computation structure for each state is implemented as a separate circuit and these are activated in turn, hence a 'wave' of activity rolling through a 'daisy chained' structure stage by stage. This is illustrated in Figure 2.31

Progression between stages is controlled by memory wait events. This means that as little as one core-stage is active during a given period, and with appropriate power gating techniques, proceeding or successive cores could be powered off altogether and eliminate redundant leakage power sinks. Wave-core architecture is implemented in a way that could be expected to be larger in a silicon area, because it

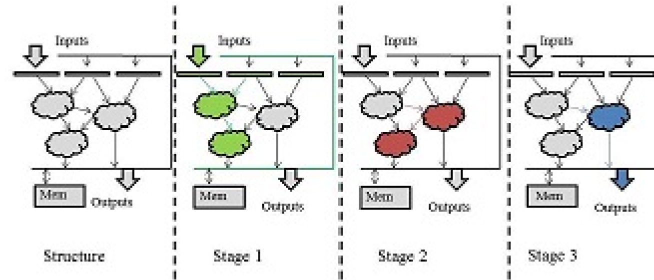


FIGURE 2.30: The Composite architecture uses a share logic design. Starting from the left, the structure shows a set of logic generate to perform a specific function that is executed in three stages. Stage 1 shows two possible logics candidate to perform the first step. Stage 2 shows the other two possible logics candidate to perform the second step and finally stage 3 a possible candidate to perform the last step. Note in stage 1 and stage 2 the last logic in green and red in used in both stage as happen the same in stage 2 and stage 3 logic in red and in blue are used to perform the operation. It means that the architecture reuse the logic as soon become available. In this way the architecture optimise the area at coast of increasing power density

increases the number of logics in the chip. But that power consumption as a whole may be reduced, and even if not, the power density would be better, since the same work is being done by a larger number of logics.

2.8.1 The Wave-core architecture: An idea to reduce power

Moore's law and shrinking transistors have changed the evolution of technology. This improvement in innovative chips has given a substantial profit in the form of sophisticated integration density, greater performance and low cost. Nowadays these three achievements are reducing while the average power density continues to increase, as is demonstrated in the International Technology Roadmap for Semiconductor (ITRS) [22]

The power density is expressed as: PD = W/cm ˆ 2. One possible solution to ameliorate the power density is to expand the area of the logic circuit. One possible solution to ameliorate the power density is to expand the area of the logic circuit. Therefore, it is important to understand about the increase in area and, as a consequence, the wire length of the logic circuit which kind of response the chip has regardless of power, and timing and also coast of fabrication.

The main concern of expanding the area of the chip are; the increase in leakage current, that increases with the decreasing of transistor's size and the cost of manufacture

The architecture described here, called Wave-core, is a sequential logic circuit, which uses a series of independent logic stages. Each hardware stage corresponds to a set of instructions. Each stage is controlled by a memory wait event. Observing the Wave-core architecture from another angle, it has some similarity with the pipeline in that it reduces the amount of logic per stage. Conversely, the pipeline reduces the logic per stage at the cost of increasing the number of registers. In order to run efficiently, this sequential parallelism architecture "pipeline" needs to work all the time as soon a process starts. Otherwise, its performance is declined by pipeline

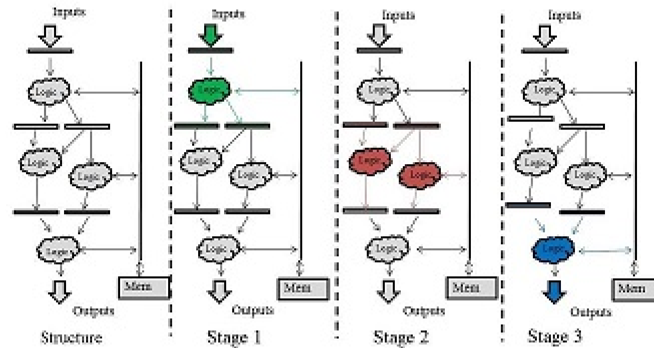


FIGURE 2.31: Wave-core architecture uses a split logic design. Starting from the left, the structure shows a set of logic generate to perform a specific function that is executed in three stages. Stage 1 shows the logics uses to perform the first stage. Stage two shows a new set of logics to perform the second step and finally the stage 3 shows a different logic to perform the last step. Note in every stage new logic/s are implemented to perform the function. It means that the architecture tries to implement new logic per stage in this way the architecture expand the area and decreases the power density.

hazards [28]. This characteristic requires the pipeline works all the time and so entails a lot of power consumption, and more thus amount of logics are regrouped in a small area of the chip generating hotspots. Observing from another angle, the Wave-core has also some similarity with multicore architecture in that it jumps from one stage to another or from one core to another core. In spite of this, the multicore architecture is surrounded by complex hardware design, such as register file and network on chip. It increases power dissipation. The Wave-core can be described as shift logic architecture. It implements the FSM using a dedicated set of logics at each state; during the process of execution it shifts from logic to logic. It means, it transfers data from a set of dedicate logic to memory at each state of the data flow. This behaviour appears as a wave form where the pic of power is only on the active stage during the sequential process of code.

To understand and compare the Wave-core architecture with the Composite, a set of experiments will be explained in the next chapters.

2.9 Summary:

The internet of Things and mobile phone require long battery life. To extend battery life, many CPU architects have begun considering CoDA architecture, and therefore converting software functions into hardware co-processors. Today many investigations are centred on cut off hardware scheduling and move complex software function into hardware IPs core, as explained in section 2. This work aims to use a translator tool in Stack structure to generate simple and faster hardware IPs design from the software level as explained in section 2.4 & 2.5 with the advantage of optimising them at compiler level and cutting off all hardware scheduling. (Note chapter 4 for more details on compiler optimization).

Chapter 3

Standard Core Generation Experiment

Chapter 2 introduces the Composite and the Wave-core architecture. To summarize, the Composite architecture is implemented using a classic implementation of Finite State Machine (FSM), therefore optimizing the area of logic at cost the of increasing the power density, as will be demonstrated. Figure 3.1 shows the diagram block of the Composite architecture.

The Composite architecture is a synchronous system controlled by a clock rising edge , so at each clock rising edge the state changes value. A finite state machine is generally formed by a set of registers all having the same clock signal. In this case the Composite architecture also has a stack memory interface to read the input data from the top of the stack and to return the value of the data to the top of the stack. Returning to the block diagram Figure 3.1, the FSM is formed by state memory and a shared logic process. All past states data is stored in the state memory . This helps to jump into the current state and into the subsequent state. The size of the state memory is finite and that why it is called a finite state machine. Basically, the amount of memory allocated is designed by a synthesizing tool during the process of implementation and follows the rule of x memory elements which can address up to 2x states. Each state is connected by D type flip flop.

The shared logic process is the execution part. It is formed by logics which compute the state. One of the main differences with the Wave-core architecture, as will be explained soon, is that; in the Composite architecture the sharing logic process groups all logics necessary to execute all states.

On the other hand the Wave-core Figure 3.2 is an improvement of the architecture design than the Composite architecture; it has a similar architecture, but is implemented using a multi processes technique or separate logic process. It means that each logic process has just the amount of logic to compute a specific state. This helps to expand the area of logics so decreasing the power density. It is formed by state memory to memorize the states, as seen in the Composite architecture, and a dedicated circuitry that actively computes the states.

The Wave-core claims that this wave of activating transistors across the section of the states helps to distribute the heat and therefore dissipate the power more effectively. Correspondingly these expansions of logic potentially reduce the power density of Wave-core when compared with the Composite architecture.

To demonstrate the efficiency of the Wave-core, but also to better understand the differences between the Composite and the Wave-core. This thesis analyses the area, timing and power for each IP core generated . To make these comparisons, this work implements and compares both architectures. As the ultimate goal, it gives a clear picture of both architectures and identifies how both architectures could be applied to improve the power efficiency of the processor.

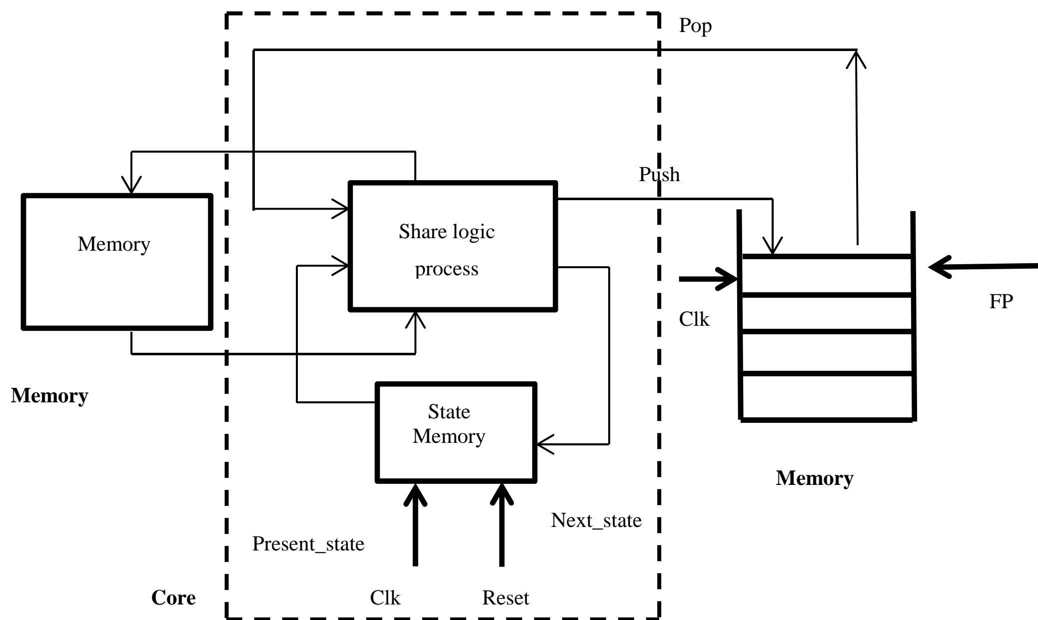


FIGURE 3.1: Block diagram of the Composite architecture. It is a synchronous architecture, therefore at each rising clock edge the state changes. It optimize the area of the sharing logic at cost of increase the power density.

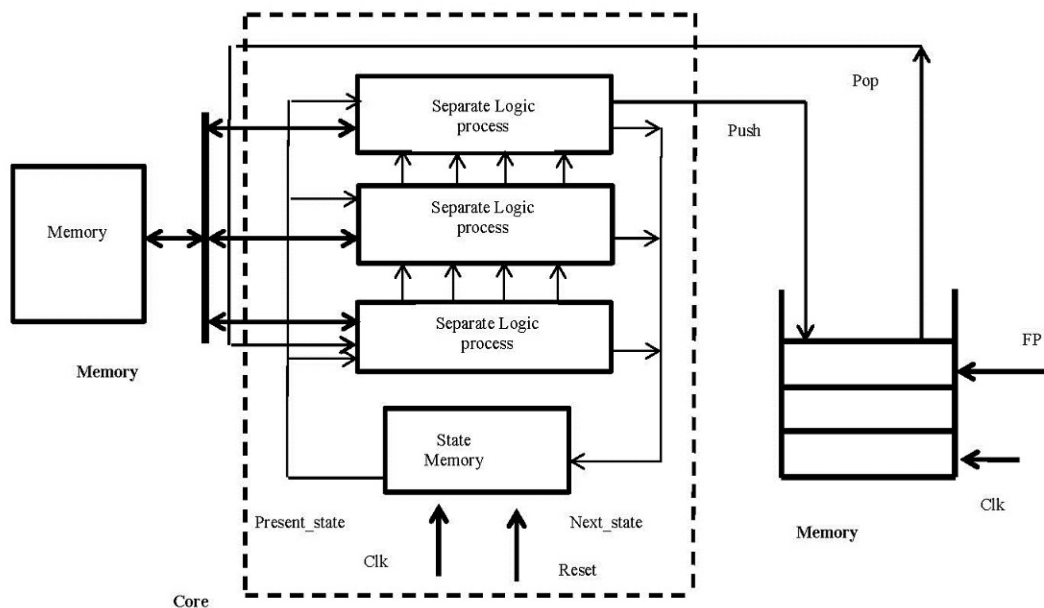


FIGURE 3.2: Synchronous architecture with a dedicate logic architecture per state. It uses a separate logic process and each other is interconnected by signals. This technique expands the area and improve power the power density.

The method of comparison starts with a standard version of different benchmarks called “baseline”. The baseline benchmarks are based on the standard compilation model. The benchmarks considered are: Binary-trees.baseline [6], Core.baseline [11], Fannkuch.baseline [17], Fasta.baseline [1], N-body.baseline [47], Scimark.baseline [14]], and Spectral.baseline [59].

Remember; such benchmarks were processed using the translator tool to generate a set of synthesisable VHDL models (see Chapter 2.5.1). Then, these cores were synthesised using the CADENCE VLSI suit 65nm UMC CMOS FARADAY standard cell library.

To deeply understand the IP cores, at first the hardware units for the seven benchmarks in relation of the number of Inputs/Outputs are studied. Note that the number of inputs/outputs represents the number of variables to be popped and pushed from/on the stack memory. They describe the size of the stack memory. Then for each IP core the number of the states in the state machine is examined. The number of states signify the number of clock cycles taken to execute the function, and finally the number of instructions per clock cycle (IPC) is observed to understand how many operations could be executed at each clock cycle.

3.1 The Baseline core bench marks Input Comparison

The first study to understand the characteristic of the IP cores for both Composite and Wave-core architecture is to observe the number of inputs/outputs of each IP core. It explains how many variables are needed as inputs to execute a function and how many variables return as results, outputs. This helps to formulate which kind of connection the core needs to communicate with the stack memory. But it also describes the size of the stack memory. Figure 3.3 and Figure 3.4 describe for each benchmark the number of inputs as a percentage.

To analyse the number of input, the seven benchmarks were synthesised using the aforementioned translator tool. The translator tool during the translation process, assembly to VHDL, generates a csv file. It records the number of Input/Output, number of states in the state machine and the Instruction per clock cycle IPC.

3.1.1 Input operands per core (Smaller benchmarks, baseline case)

The cores in the Binary-trees benchmark Figure 3.3 a) have a minimum of zero to a maximum of four inputs. It shows that the majority of cores around 57 % have zero input. Then about 22 % of cores have two inputs. About 19 % of the cores have just one input and only around 1.7 % of cores have four inputs.

The cores in the Funnkuch benchmark Figure 3.3 b) have a number of inputs between zero and two. In this case the percentage of cores with zero input increases to around 64 %. The number of core with one input has also increased to circa 27 %. The other ˜ 9.5 % of cores have two inputs.

The majority of cores in the n-body benchmark Figure 3.3 d) have ˜ 58.3 % zero input and roughly 25 % have one input; about 10.4 % have two inputs. The others circa 2.1 % have four, five and seven.

The Spectral-norm benchmark Figure 3.4 f) has similar characteristics to others and so most cores around 51 % have zero input, and the least nearly 1.6 % have four, five and six; in the region of 20 % have just one input while about 15 % have two. Cores with three inputs make up about 6.6 %, seven circa 3.3 % of the total.

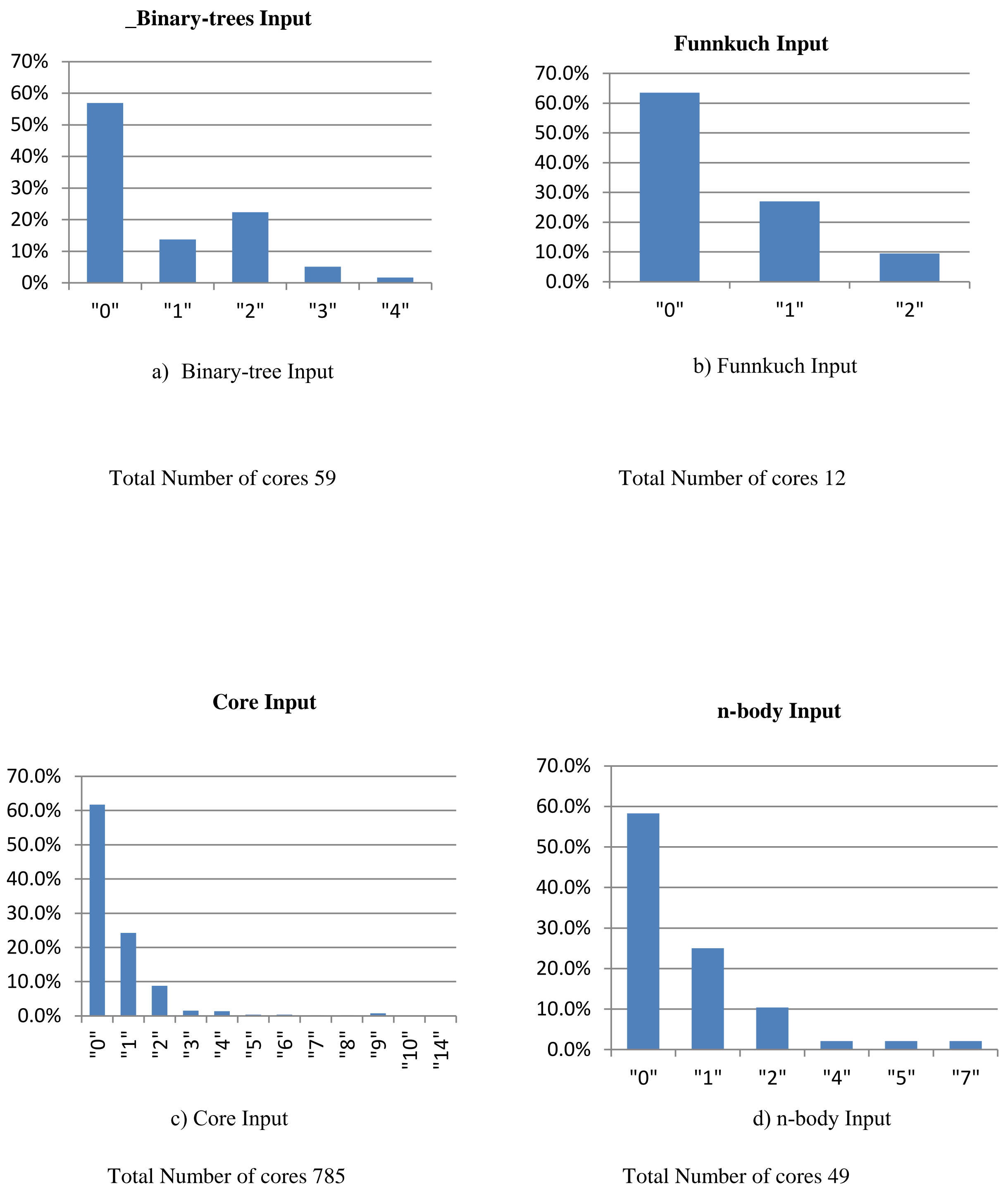
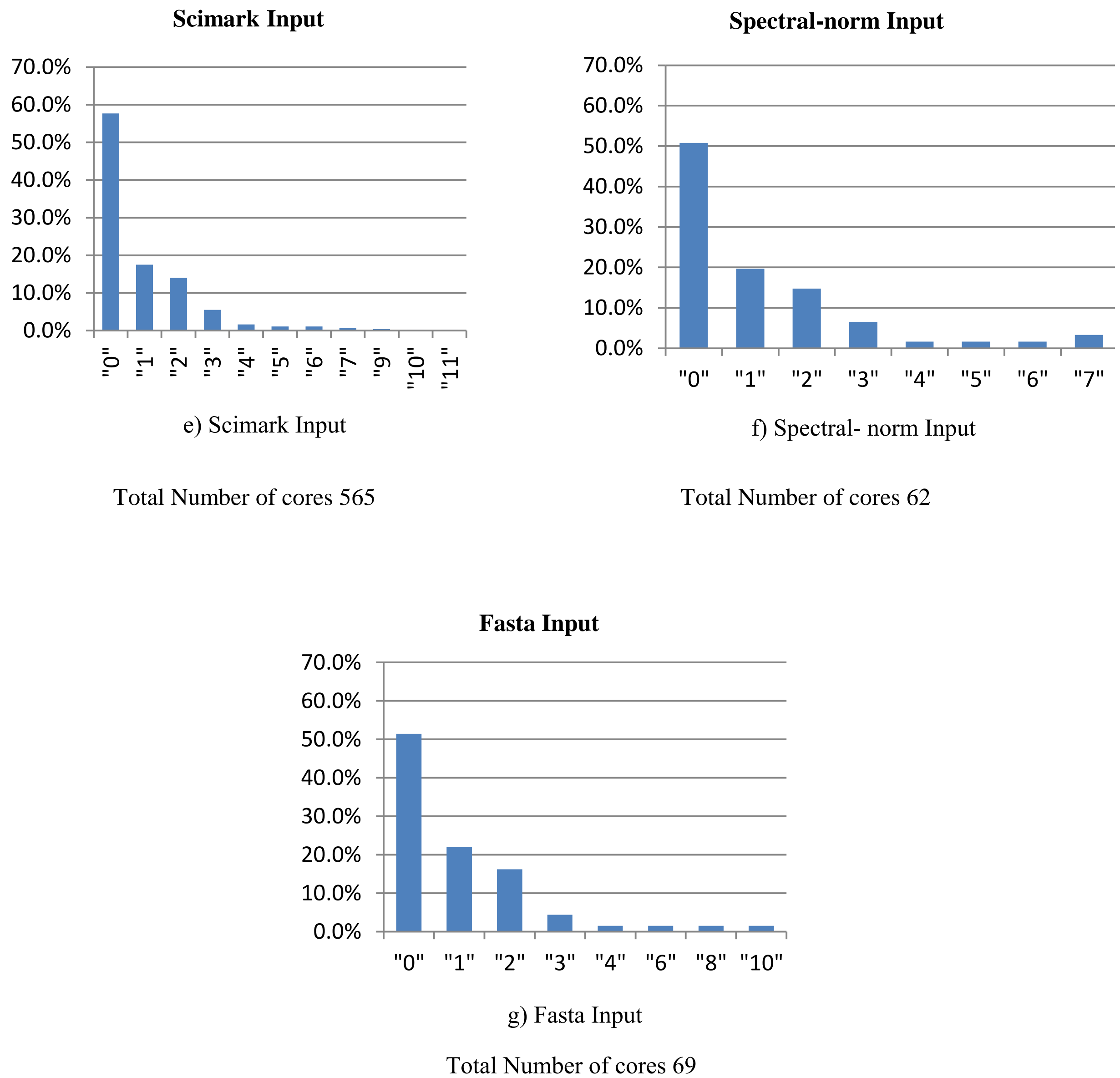


FIGURE 3.3: N. of Input per core per benchmarks. The higher percentage of cores has a low number of Inpute



e) Scimark Input

Total Number of cores 565

f) Spectral- norm Input

Total Number of cores 62

g) Fasta Input

Total Number of cores 69

FIGURE 3.4: N. of Input per core per benchmarks. The higher percentage of cores has a low number of Inpute

In the Figure 3.4 g) Fasta benchmark about 1.5 % of the cores have ten, eight, six and four inputs, but the most common remain zero input cores, at about 52 %. It has around 22 % with just one input and about 15 % with only two.

3.1.2 Input operands per core (Larger benchmarks, baseline case)

The Core benchmark Figure 3.3 c) has a different scenario. The number of inputs ranges between zero and fourteen. The majority of cores, approximately 62 %, have zero input, followed by approximately 24 % with one input and around 8.8 % with two. Slightly more than circa 1 % have inputs of three and four, and all others (very nearly 0 %) have a higher number of inputs.

The Scimark benchmark Figure 3.4 e) has a spread of inputs that vary between zero and eleven. The majority, at around 58 %, have zero input, while roughly 17.6 % have just one input. Also about 14 % of cores have two inputs and in the region of 5.5 % have three. The minority of cores have five, six, seven, nine, ten and eleven.

3.1.3 Input operands per core (All benchmarks, baseline case)

In conclusion, the number of inputs for each IP cores per benchmark have been analysed. The cores with zero input were found to be the most common because for most of the time the core acts just as a transient. Therefore it passes information from a core to another core. Then the percentage decreases, but remains consistent for the number of cores that have two, three and four inputs. The data also show that only a few IP cores had more than four inputs, such as the Core benchmark with just 0.1 % of cores having fourteen inputs.

Again as before mentioned; the number of inputs is the number of operands needed to pop off of the CPU operand stack before core invocation. This leads to the inescapable conclusion that to execute function using stack architecture it needs a communication bus to perform up to four variables from the top of the stack to the core being used. The total number of input for all benchmarks is shown in Figure 3.5

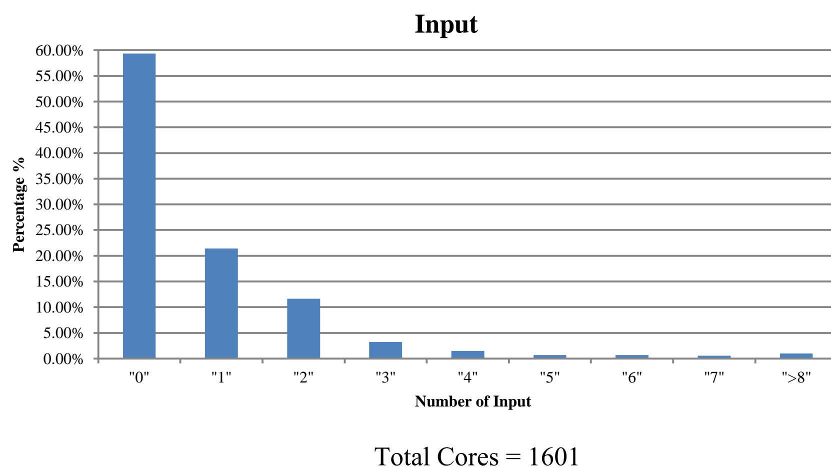


FIGURE 3.5: Total number of inputs in the seven benchmarks (The majority of cores have between zero and two inputs)

3.2 The Baseline core benchmarks Output Comparison

Understanding the characteristics of the benchmark helps us to fully figure out how both architectures might work. This section investigates the number of output results each function in each benchmark generate. The output indicates how many parameters need to be pushed back on the stack after the core completes its work. Adding the number of inputs from the previous section, it gives a complete picture on the size of stack needed and also the type of communication needed between memory and IP cores.

Figure 3.6 and Figure 3.7 shows the percentage of the number of outputs each core requires.

3.2.1 Output operands per core (Smaller benchmarks, baseline case)

The Fannkuch benchmark Figure 3.6 a) shows that most of the cores (about 75 %) have zero or one output. The maximum number of outputs is three; however, it is the minority, with just around 1 %. The other 2 % have 2 outputs.

The n-body benchmark Figure 3.6 c), has the number of outputs between zero and three. The majority of cores have zero output close to 79.2 % followed by around 10.4 % with just one. Also, circa 6.3 % have two outputs and finally nearly 4.2 % have three outputs.

The Binary-Trees Figure 3.7 e) has in the range of 45 % of cores with zeros output. About 40 % have one, a further 10 % have two and finally about 1.7 % have four outputs.

In Spectral Figure 3.7 f). The greater part of cores roughly 74 % have zero outputs, and circa 15 % have just one. Thereafter, about 6.6 % have three outputs and about 5 % have 2.

Ultimately, the Fasta benchmarkFigure 3.7 g) has its outputs grouped between zero and five. In the range of 70 % have zero output. Cores with one output make up 9 %; nearly 7.5 % have two outputs, about 6 % have three, circa 4.5 % have four and finally roughly 3 % have five outputs.

3.2.2 Output operands per core (Larger benchmarks, baseline case)

The scenario for the Core benchmark Figure 3.6 b) is slightly different. The output ranges between zero and six. But, most of the cores have between zero and one output: around 76 % have zero and around circa 15 % just one. The minority of cores have between two roughly (5.4 %) and three about (2.6 %) outputs. Furthermore, circa 0.9 % of cores have four outputs, followed by nearly 0.4 % with six and about 0.3 % with five.

The Scimark Figure 3.6 d) spreads output requirements are between zero to five. slightly less than 70 % (the majority) are grouped at zero input, and circa 22 % have one output. Around 5 % of cores have two outputs, roughly 2.5 % three and about 1.6 % four outputs.

3.2.3 Output operands per core (All benchmarks, baseline case)

According to the data, the cores have a common attribute. The majority of cores have zero output, and the next largest percentage have one. For instance, the Fannkuch has around 75 % of total cores with zero and one outputs. Other benchmarks have a consistent proportion extended up to four outputs. The number of outputs are

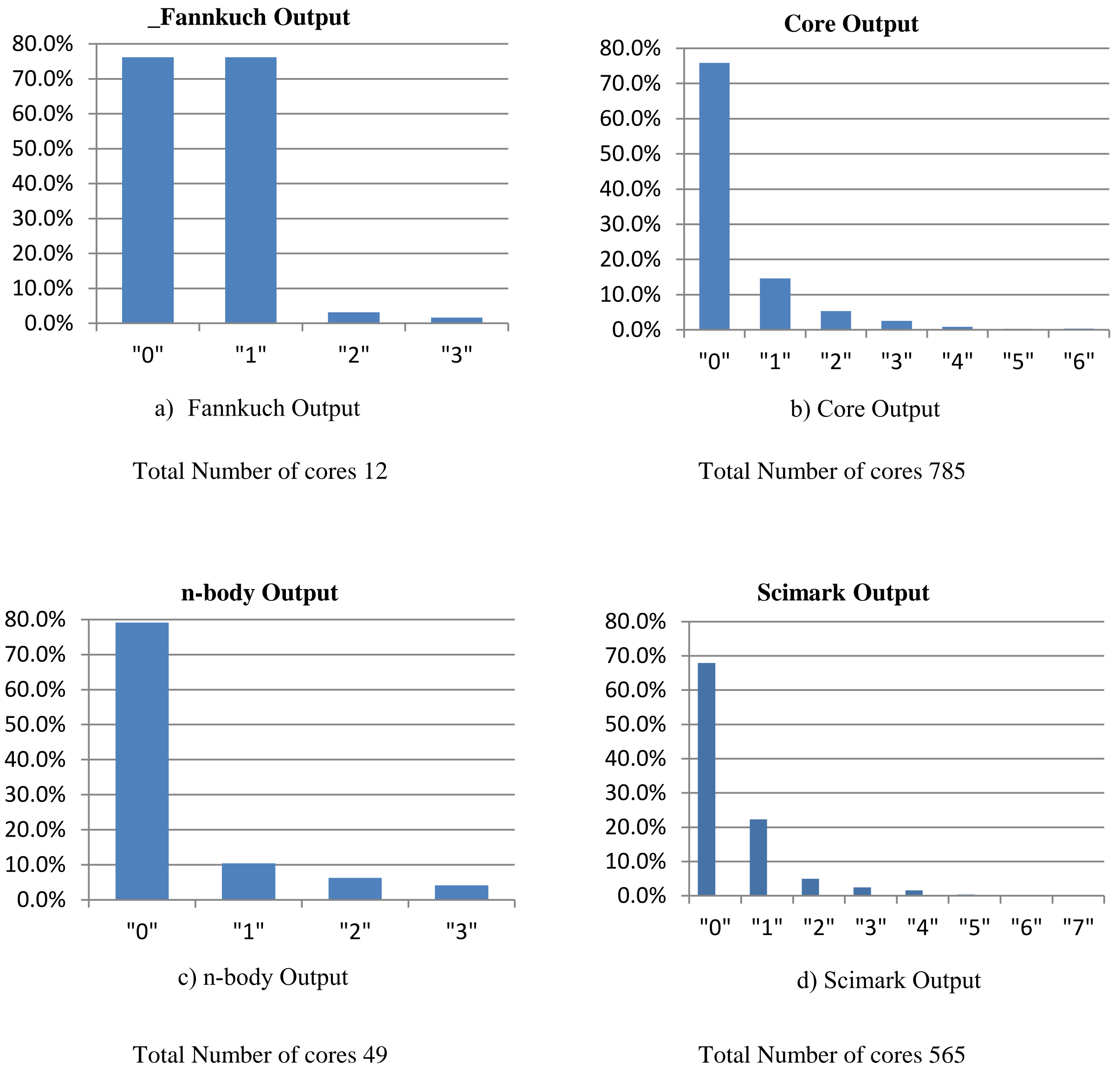


FIGURE 3.6: N. of Onput per core per benchmarks. The higher percentage of cores has a low number of Output

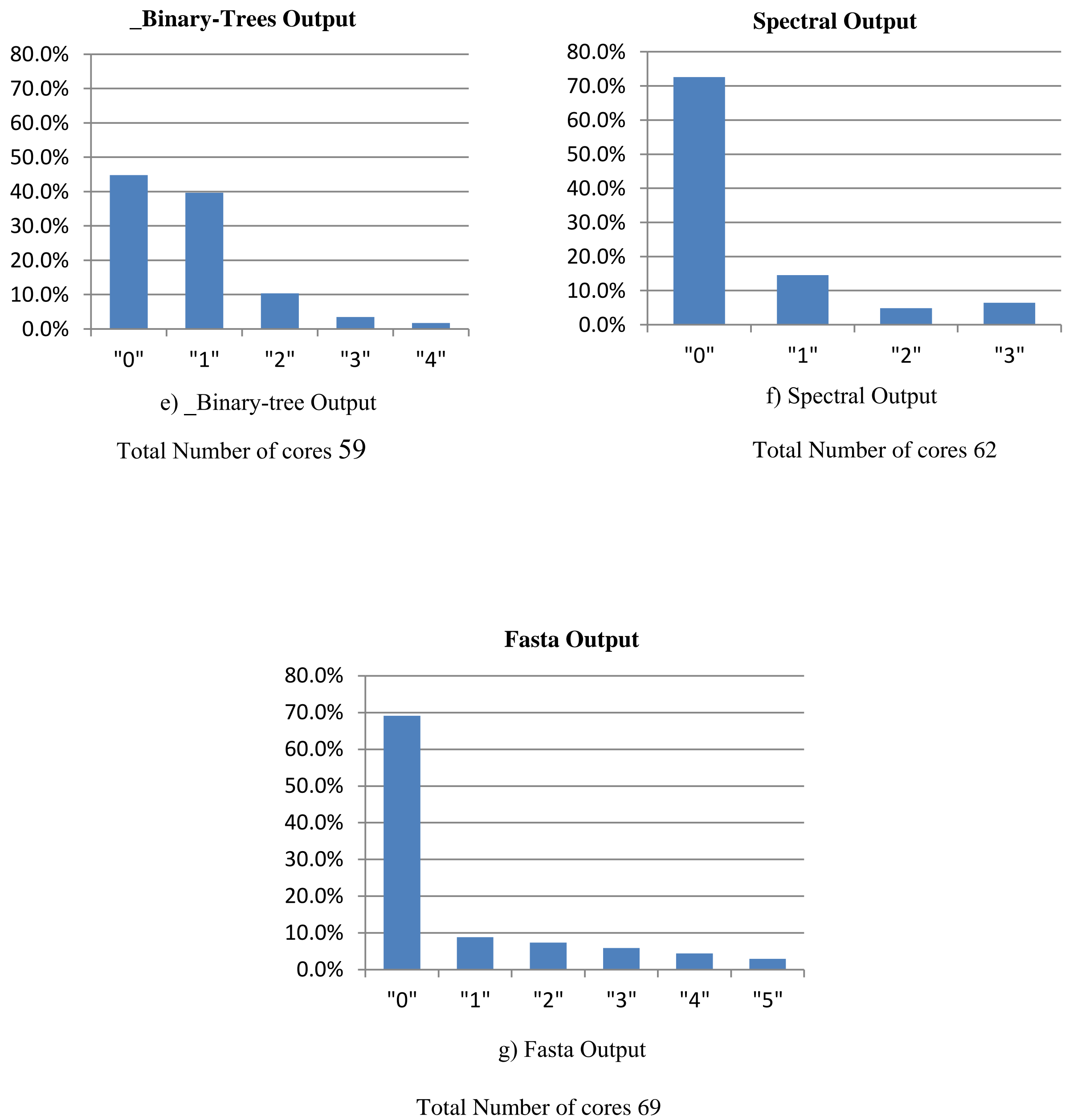


FIGURE 3.7: N. of Output per core per benchmarks. The higher percentage of cores has a low number of Output

the number of results to be pushed back onto the stack after the core operation, and therefore it describes how much communication overhead is required. It is very important to understand the amount of memory needed, memory level stack, and type of communication, memory to IP core.

To conclude, the analysis demonstrates the majority of function request just four level stack memory and so the system require a small stack memory. It also demonstrates that all these data could be moved around using a simple two variables bus communication.

Figure 3.8 shows the total number of outputs in the combined seven benchmarks as an overall model.

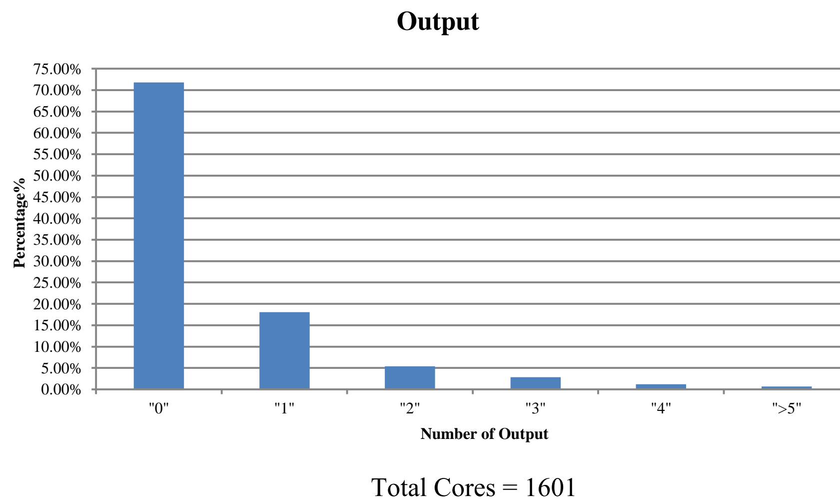


FIGURE 3.8: Total number of output in the combined seven benchmarks (The majority of cores have an output of between zero and two

3.3 The number of states in the Finite State Machine

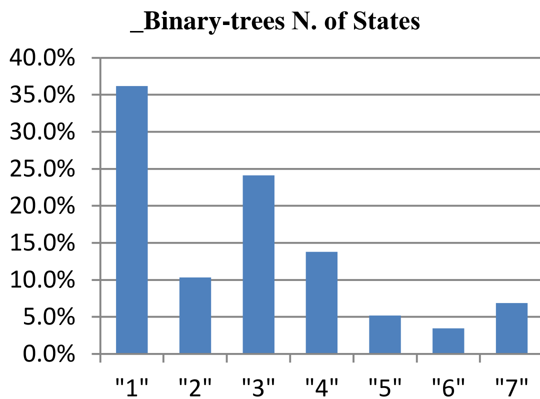
This section analyses the number of states in each IP core generated. The general idea is to understand better the benchmarks and analyse how many states the architectures, Composite and Wave-core, need to perform a function. This will help especially in Chapter 5 to observe how the performance, power, area and timing, change when the architectures increase or decrease the number of states. Figure 3.9 and Figure 3.10 show the number of states in percentage for each function.

3.3.1 Number of states per core (Smaller benchmarks, baseline case)

Binary-tree benchmark Figure 3.9 a) shows that the states vary between one and seven. The data demonstrates that the majority of IP core has from one to four states. The first and highest percentage is represented by one state, in the range of 36 %, followed by three states with around 24 %.

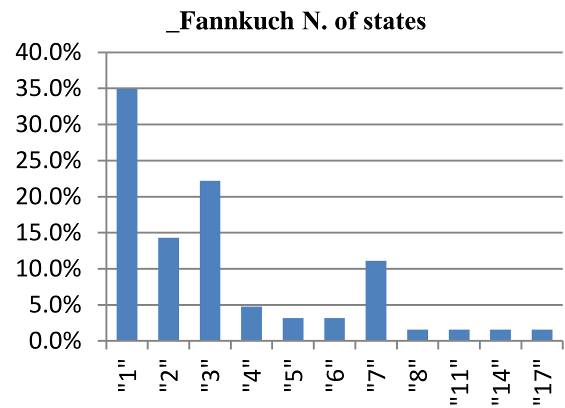
A roughly further 14 % of cores have four states and just in the range of 10 % have two states. Finally, between circa 7 % and around 5 % have seven, six and five states.

The scenario is similar for the Fannkuch benchmark Figure 3.9 b). It shows that the majority of IP cores have from one to three states. However, the spread increases



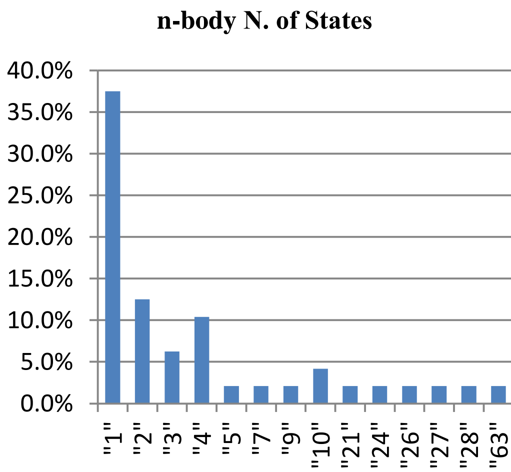
a) _Binary-trees N. of States

Total number of cores 59



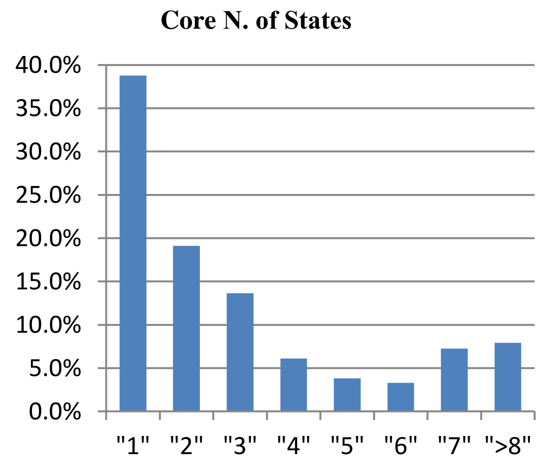
b) _Fannkuch N. of states

Total number of cores 12



c) n-body N. of States

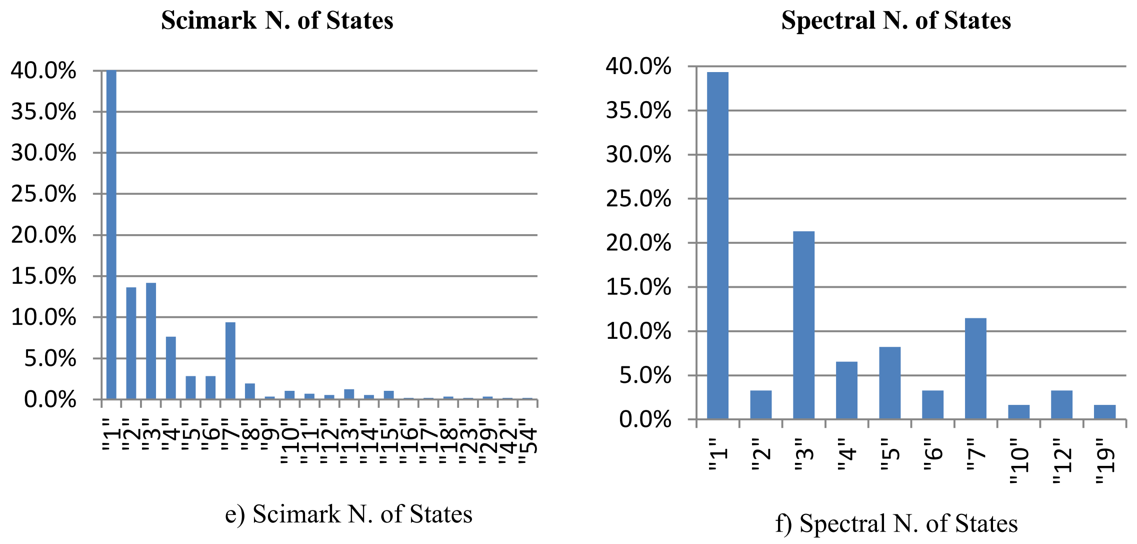
Total number of cores 49



d) Core N. of States

Total number of cores 785

FIGURE 3.9: Number of state for benchmark, The majority of cores execute the function in 3 states

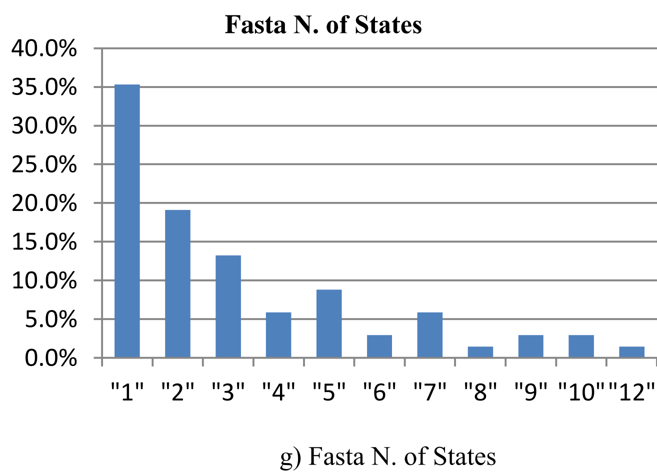


e) Scimark N. of States

f) Spectral N. of States

Total number of cores 565

Total number of cores 62



g) Fasta N. of States

Total number of cores 69

FIGURE 3.10: Number of state for benchmark, The majority of cores execute the function in 3 states

and reaches seventeen states. What makes this benchmark very interesting is that it has around 11 % of cores with eleven states, and is therefore a suitable candidate to analyse Wave-core architecture and compare it with Composite, because it can observe if increasing the number of states improves the performance in power consumption in the Wave-core.

The n-body benchmark Figure 3.9 c)has a huge range of state per core. It varies between one, which it is the most populated, to sixty-three. The majority of cores perform the function within four states. However, it has a few cores with five and six states, which could be used to observe how Wave-core architecture behaves.

The Spectral benchmark Figure 3.10 f) is very compact with the number of states grouped between one and nineteen states. The most populated core performs the function in one state. However, around 23 % and circa 12 % execute the function in three and seven states. It tells us that stack producOn the other hand the Wave-core (Figure 3.2) is an improvement of the architecture design than the Composite architecture;e very compact code and tends to implement a function in few clock cycles.

A last benchmark to observe is the Fasta Figure 3.10 g). Most cores complete the function in just three states, therefore it is not the best candidate to observe how Wave-core behaves. However, it can help to understand Composite architecture better.

3.3.2 Number of states per core (Larger benchmarks, baseline case)

In the core benchmark Figure 3.9 d), the majority of cores with almost 40 % had one state. Then around 20 % two sates and just less than 15 % three states. It also has few cores around 8 % with seven and more than eight states.

In the range of 40 % the Scimark benchmark Figure 3.10 e) has one state. However, a few percentages of cores have up to fifty-four states. Notably, the other around 30 % is spread between two and three states. Then about 15 % of cores have four and seven states. A very small percentage increases the number of state up to 54.

3.3.3 Number of states per core (All benchmarks, baseline case)

This section has investigated the number of states required for cores in each benchmark. The overall data shows that the majority of cores complete the function in just one state and it makes sense when compared with the number of inputs/outputs equal to zero because it demonstrates the majority of IP cores acts just as a transient. However, the range is not negligible for larger state-counts. Figure 3.11 shows the total number of the states in the seven benchmarks. Another point to elucidate is that the number of states in the state machine are directly associate with the power. Decreasing the number of states the architecture decreases the power. Section 4 explains this in details. It also demonstrates how to decrease the number of states in the hardware IPs cores by applying instruction stack scheduling techniques at compiler level.

3.4 The Number of Instructions per Clock Cycle

The number of instructions per clock cycle (IPC) helps us to better understand the aggregate data flow of each benchmark. It shows the average number of instructions executed for each clock cycle equivalent to each state operated by a given core. In

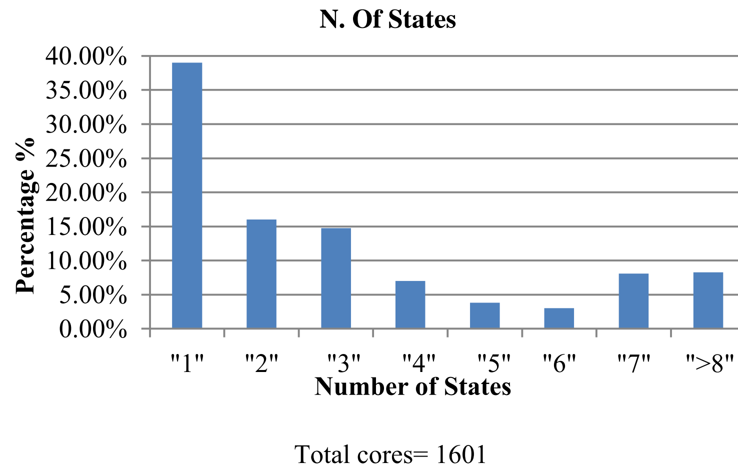


FIGURE 3.11: Total number of states in the seven benchmarks. It shows the majority of cores perform the function into 3 states

other words, the number of CPU instructions that can be fitted into each state. One of the key aspects of this thesis is to analyse how the architecture evolves when it optimises the hardware using static parallelism. Static parallelism means finding parallelism at software level.

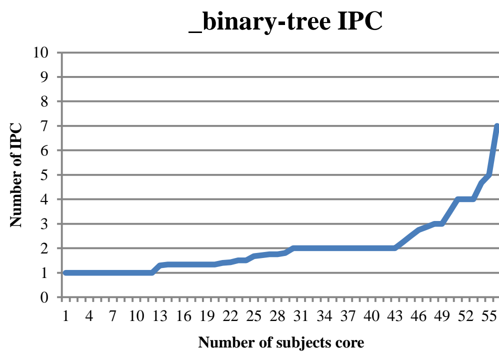
Understanding the IPC is one of the important aspects because it helps to demonstrate whether extracting more software IPC helps to reduce the power and process faster function.

By observing the IPC for each benchmark and comparing with the further optimisation, (see Chapter 4), the way the number of instructions per clock cycle increases can be analysed and how this affects the performance of the two architectures can be shown. Chapter 4 will also explain how increasing the IPC, decreases the number of states. This is an important point because increasing the number of parallelism the Composite and the Wave-core architecture in the way they are designed decrease the number of states. Therefore, they either can perform faster function or improve the power consumption. Figure 3.12 Figure 3.13 show the IPC for each benchmark.

The seven benchmarks have related characteristics. The majority of cores implement few instructions per clock cycle; It means the way the instruction is implemented is very poor, because if only a few instructions per clock cycle are executed the architecture becomes slower. The maximum number of instructions per clock cycle is eight Figure 3.13 b) Core benchmark. The other benchmarks have a maximum of seven instructions per clock cycle, except for the n-body benchmark Figure 3.13 e) that has a maximum number of six instructions per clock cycle.

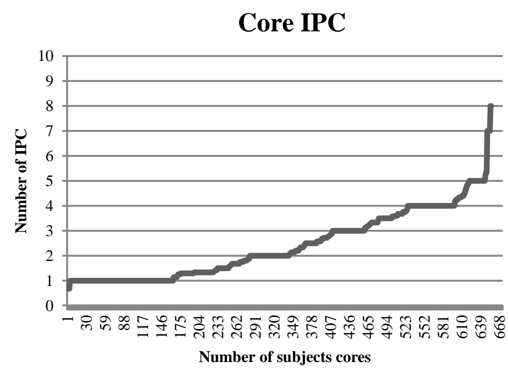
This indicates an interesting point: a significant number of cores have the ability to operate the equivalent of 2, 3, 4, or more instructions per clock cycle. If cores are clocked at a similar rate to the master CPU then the architectures might gain performance in speed as well as in power. This concept will be better understood in Chapter 4.

A further consideration is worth noting, which is to observe that cases in which only a few states might appear are likely to be poor candidates for selection to be implemented, but this also depends on the IPC. A 2 states cores with IPC= 8 is potentially more desirable than a 5 states core with IPC =1, because the architecture could reduce the frequency and so decrease the power consumption. Figure 3.14



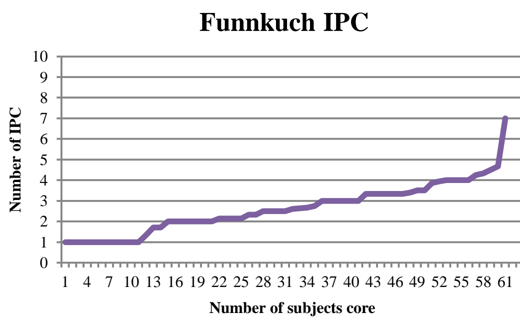
a) _binary-tree IPC

Total Number of cores 59



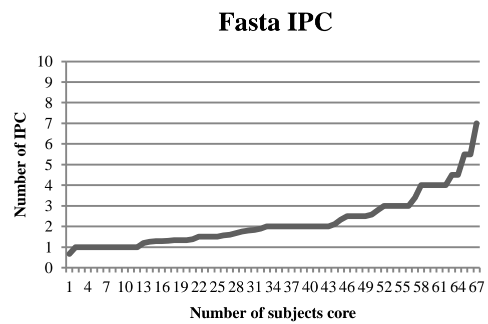
b) Core IPC

Total Number of cores 785



c) Funnkuch IPC

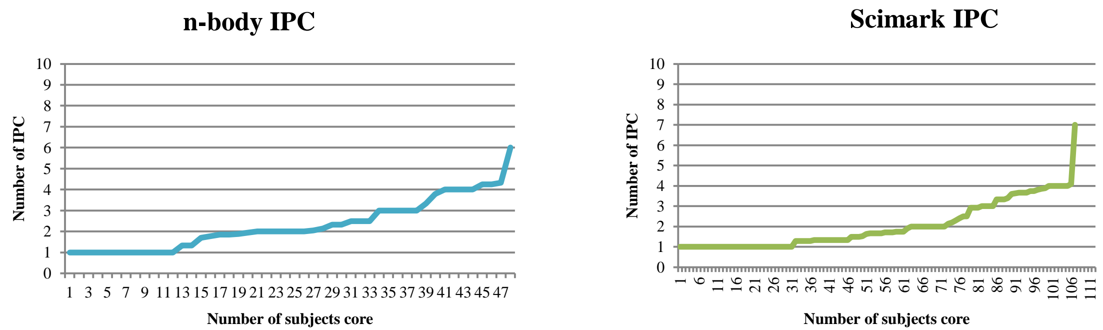
Total Number of cores 12



d) Fasta IPC

Total Number of cores 69

FIGURE 3.12: Number of instructions per clock cycle for benchmark. It explains the number of IPC executed at each state

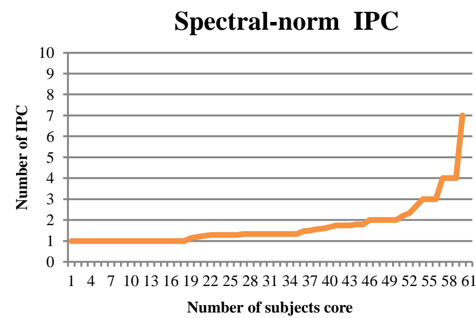


e) n-body IPC

Total Number of cores 49

f) Scimark IPC

Total Number of cores 565



g) Spectral- norm IPC

Total Number of cores 62

FIGURE 3.13: Number of instructions per clock cycle for benchmark. It explains the number of IPC executed at each state

plots the number of state vs IPC. This concept will be further developed in Chapter 4.

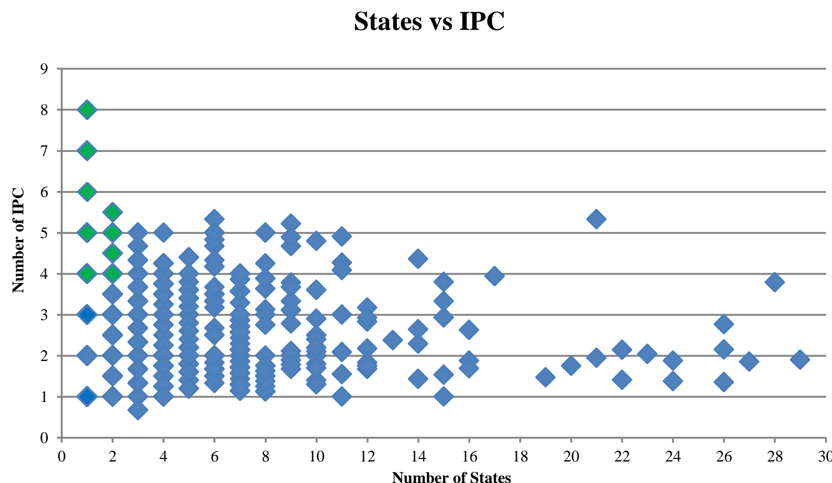


FIGURE 3.14: Number of states vs IPC it shows how many instruction can be performed per clock cycle. It could be very important in optimizing the power, less state, less clock cycle give as a result better power consumption hardware architecture.

Figure 3.14 demonstrates that a higher number of cores could perform with just a few clock cycles. It means smaller number of state implemented per each state machine, less clock cycle to implement the function and lower frequency per clock cycle. All these factors help to decrease the power consumption.

3.5 Composite vs Wave-core Total Area

To understand the characteristics of both Composite and Wave-core architectures and spots the differences and so the advantages and disadvantages, this section starts from the total area. The Composite architecture is described as a single process sequential architecture. It implements the function in a way to save area, squeeze the logic and so decrease the static power and optimizing the timing at cost of increasing power density and increasing the possibility to generate hotspots. In comparison, Wave-core architecture is implemented with the intention to reduce the power density by expanding the area and reducing the logic per area. The comparison between the Composite and the Wave-core start with the total area using Box and Whisker chart [7]. The total area of both architectures are shown in Figure 3.15. The differences quartile is shown in Table 3.1

To understand in general how the architectures behave, at first the median is considered. The median for Composite architecture has a total area of 2009um ˆ 2. Alternatively Wave-core architecture had the median at 3617 um ˆ 2.

Again, comparing the median in Composite architecture and extend to the third quartile, the value observed is below the median of Wave-core architecture; in fact, it is at 2829um ˆ 2, while the median for Wave-core is at 3617um ˆ 2. At this stage becomes clear that Wave-core architecture increases the area of logic because around 75 % of cores in Composite architecture are less than the median of Wave-core architecture.

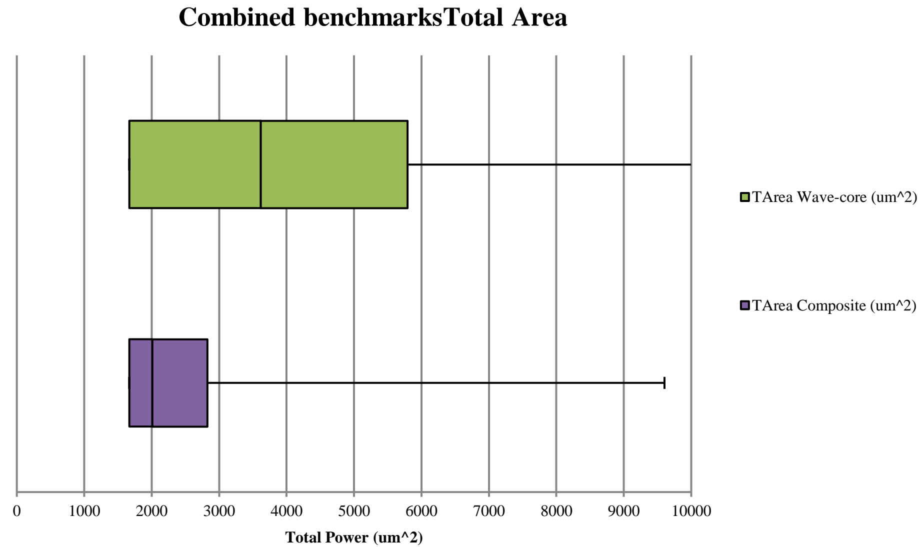


FIGURE 3.15: Describes the total area in both Composite and Wave-core Architecture. Wave-core expands the are

TABLE 3.1: Combined benchmarks Total Area differences quartile

	TotAreaComp (um ˆ 2)	Differences	TotAreaWave (um ˆ 2)	Differences
Min	1668	1668	1668	1668
Q1	1668	0	1668	0
Med	2009	341	3617	1949
Q3	28525.25	816.25	5793.25	2176.25
Max	9603	6777.75	56850	51056.75

Furthermore, another observations is to compare the upper quartile. Composite architecture has the upper quartile at around 2825um ˆ 2, while in contrast Wave-core architecture has its upper quartile at circa 5743um ˆ 2.

Similarly, the minimum and the low quartile are at the same value with both architectures are at around 1700um ˆ 2, because many benchmarks are implemented in one single state. In contrast, the maximum point for Composite is at 10000umˆ 2, while the maximum point for Wave-core is at around 57000umˆ 2.

Ultimately, it has demonstrated as it presented that Wave-core architecture increases its area as compared to Composite. This is fully as expected: Wave-core duplicates resources in a multiple-stage structure, so area would be expected to increase.

To see better how each core behaves for Composite and Wave-core Figure 3.16 compares area for every core function in terms of the two implementation styles. This shows that Wave-core architecture typically expands the area by around 2.3x compared with the identical Composite core function

To conclude, Figure 3.17 shows an example of a Wave-core and Composite in a 65nm layout. The Composite core overlaid on an equivalent Wave-core for size comparison. The example above shows clearly the Composite cover less area when compared with the Wave-core. The synthesis was done using the FARADAY STANDARD cell library with the 65nm target technology.

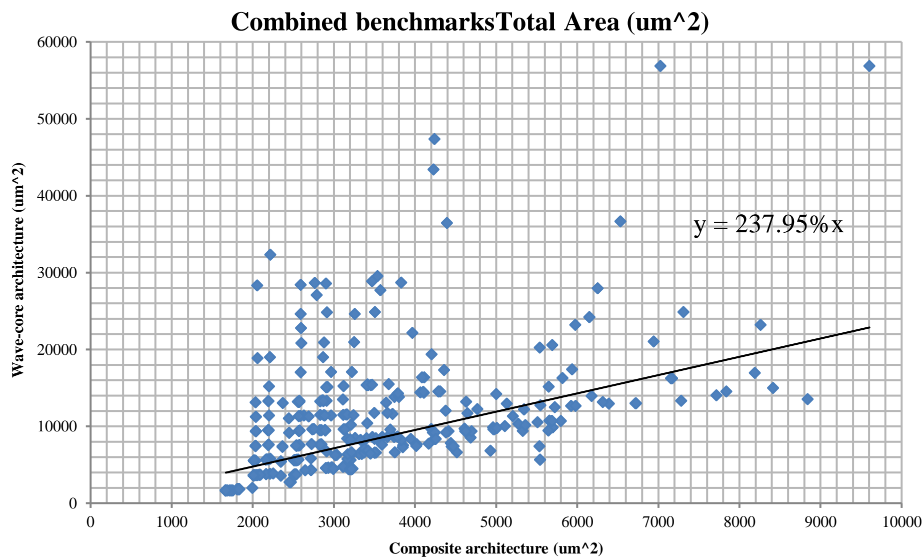


FIGURE 3.16: The Wave-core architecture expands the area about 2.3x

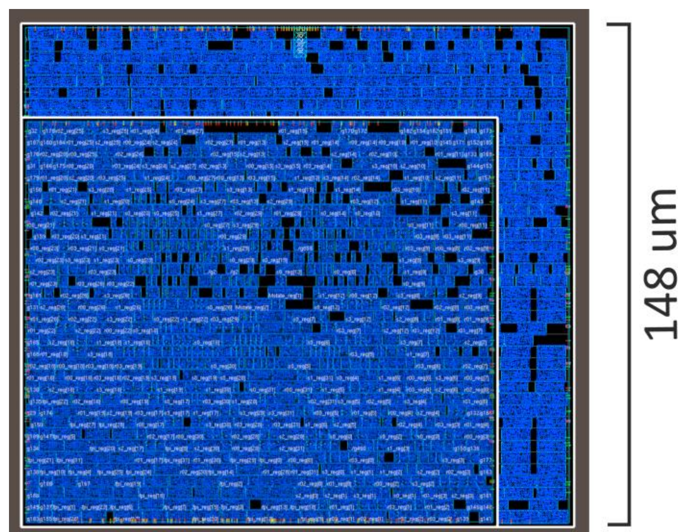


FIGURE 3.17: Composite and Wave-core compared for area. The Wave-core expand the area

3.6 Composite vs Wave-core Timing

The main difference between Composite and Wave-core architecture is; Wave-core implements a function using a novelty state machine, dedicate hardware per state, it increases the number of logics therefore appears to execute the function using more time. On the other side, Composite architecture tries to group the logic per area and so timing should slightly decreases when compared with the Wave-core. Figure 3.18 shows the overall behaviour of both Composite and Wave-core architecture in respect of time. The differences quartile is shown Table 3.2

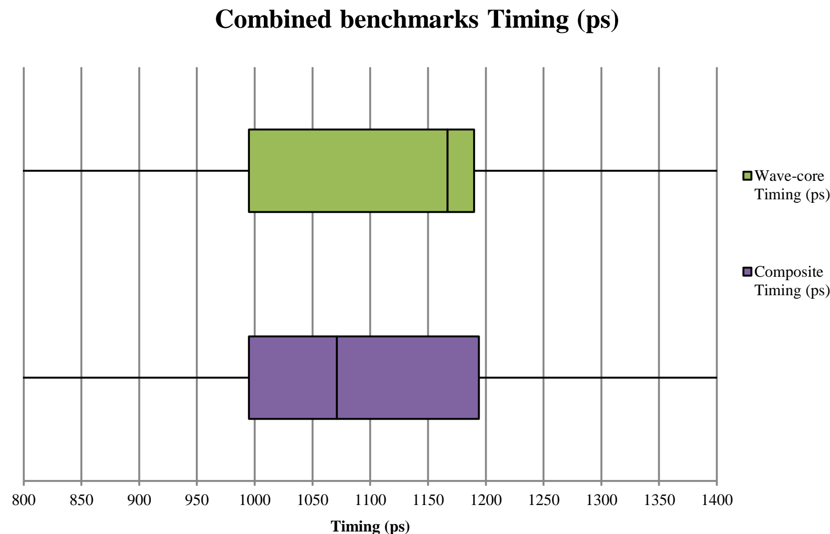


FIGURE 3.18: Combined benchmarks Timing (ps). Non parametric representation of the different quartiles. Both architectures are in the same timing range.

TABLE 3.2: Combined benchmarks Timing differences quartile

	Composite Timing (p)	Differences	Wave-core Timing (p)	Differences
Min	955	955	858	858
Q1	955	40	995	137
Med	1071	76	1167	172
Q3	1194	1233	1190	23
Max	4611212.1	4610018.102	6144225.972	6143035.972

Firstly it compares the median. Wave-core has its median at 1167ps and Composite 1071ps, therefore as we assumed Composite architecture executes using less time, but there is only 10 % between the two core types.

To investigate either further, the lower quartile and the upper quartile is analysed. The data show they match. The lower quartile for Composite is at 995ps while the lower quartile for Wave-core is also 995ps, but also the upper quartile for Composite is at 1194ps and for Wave-core it is at 1190ps. Again, it shows the difference is very little between the two cores.

Finally, the minimum and the maximum is observed for the two architectures. The minimum for Composite is at 995ps and for Wave-core it is at 858ps. Alternatively, the maximum for Composite is at around 47000000ps, 47us, while for Wave-core is at around 6200000ps, 6.2us.

In conclusion. According with whiskers and box graph Figure 3.18, the Composite architecture implements the cores using slightly less time, because its medium quartile is at around 1100ps while the Wave-core is around 1200ps. However, the upper quartile and the lower quartile are situated in the same range and more the differences are very small so it assumes there are no significant disadvantages of Wave-core for timing.

A good point to consider is that both Composite and Wave-core architectures have timing centred between 1000ps and 1200ps, and so converting in frequency it is between 1GHz and 800MHz, which is the same frequency of many modern embedded processors in 65nm [45]

3.7 Composite vs Wave-core Leakage

The multi-stage processes to implement Wave-core architecture enlarge the logic area and therefore raise the leakage. This technique is applied with the idea of decreasing the power density and helping the dynamic power, therefore this sections analyses the general data for both architectures using our standard core generator Figure 3.19 The differences quartile is shown in Table 3.3

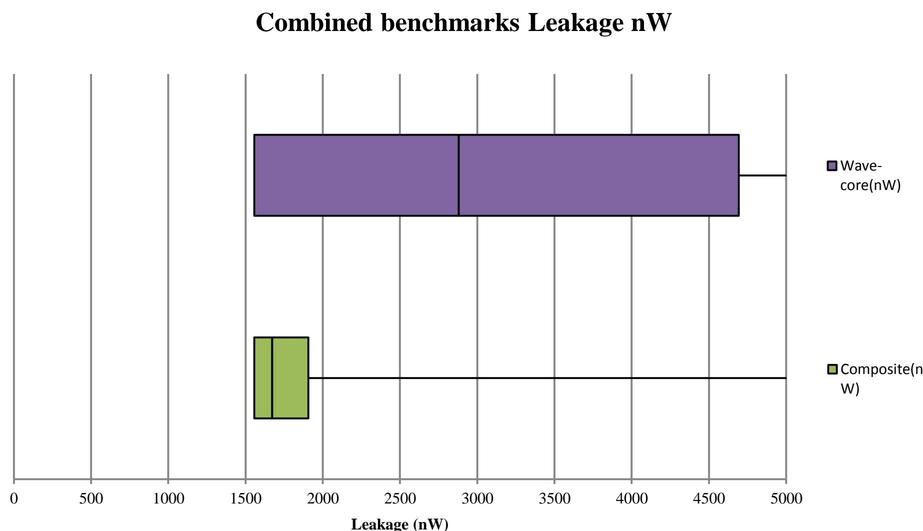


FIGURE 3.19: Combined benchmarks leakage (nW). Non parametric representation of the different quartiles. Composite use less leakage power.

The median in Wave-core architecture is at around 2900nW while Composite has its median at about 1700nW.The result demonstrates as already explained that the Wave-core increases the static power by a notable factor (around 2x) on average.

To investigate in more detail and compare both Composite and Wave-core architecture the upper quartile is observed. Similarly, the upper quartile for Composite architecture is 1900nW while for Wave-core it is around 4500nW. Notably, the upper quartile of Composite core is at 1900nW whereas the median for Wave-core is circa

TABLE 3.3: Combined benchmarks Leakage differences quartile

	C-Leakage(nW)	Differences	W-Leakage(nW)	Differences
Min	1556.871	1556.871	1556.784	1556.784
Q1	1557.027	0.156	1557.027	0.243
Med	1672.159	115.132	2881.255	1324.228
Q3	1906.25	234.091	1190	1811.133
Max	7814.286	5908.036	101115.24	96422.855

2900nW. This suggests that the most common cores in Composite architecture waste less static energy than those in Wave-core.

The data has shown that Wave-core architecture with its multi-stage design expands the area and as a consequence increases the leakage power.

In addition Figure 3.20 shows the leakage data using scatter plot. It also demonstrated that the leakage increases around 2.4

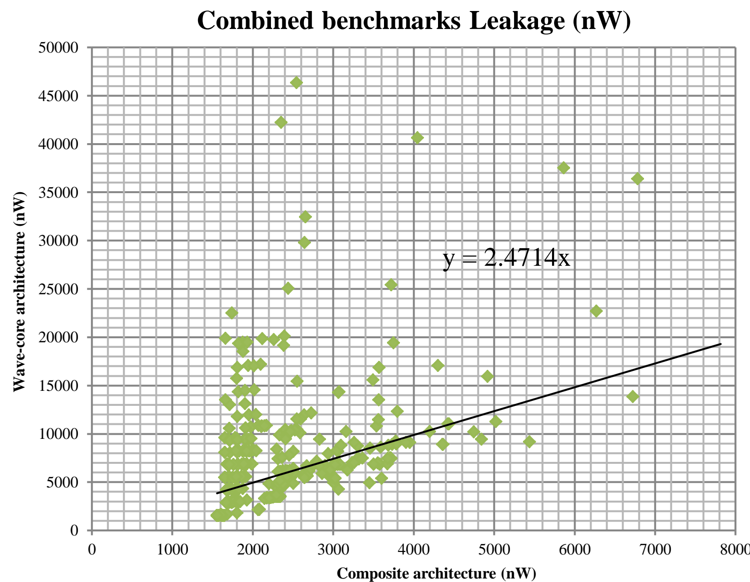


FIGURE 3.20: Combined benchmarks Leakage (nW). It shows that the Wave-core increases the leakage about 2.4x

To understand better the correlation between area and leakage power ; Figure 3.16 shows that the area typically increases by about 2.3x and Figure 3.20 shows that also the leakage increases by about 2.4x. In practice these two graphs show the same trend, in the way that area and static power are linear factors, therefore increasing the area, increases the leakage.

3.8 Composite vs Wave-core Dynamic power

To ameliorate dark silicon, two types of architectures are introduced. Wave-core architecture has been implemented with the aim to reduce the power density, by reducing the logic per state and increasing the total area. The power density is minimised at the cost of increasing static power.

To better understand this effect, this section investigates how the dynamic power is distributed among the cores between the Composite and Wave-core architectures and then by matching them improve the Dark Silicon problem. Figure 3.21 shows the dynamic power for the Composite and Wave-core architectures. The differences quartile for dynamic power is shown in Table 3.4

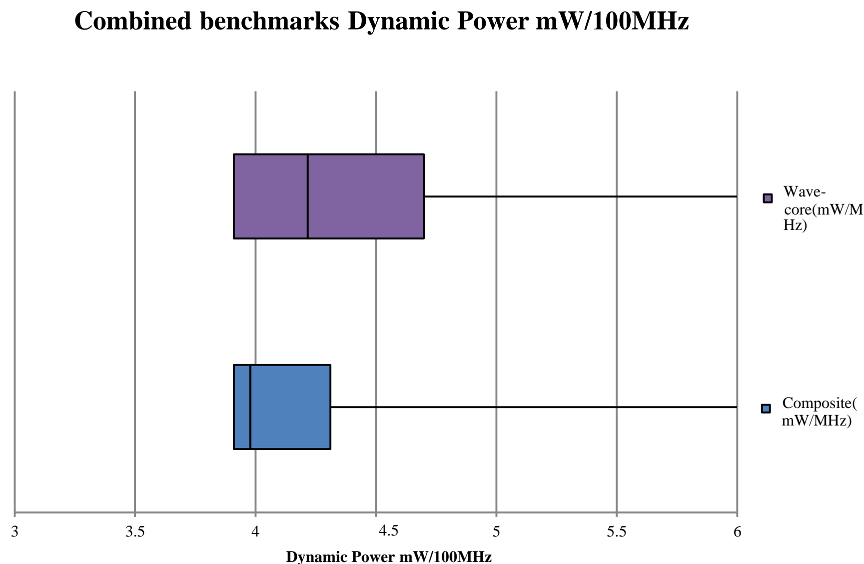


FIGURE 3.21: Combined benchmarks Leakage (nW). It shows that the Wave-core increases the Dynamic Power

TABLE 3.4: Combined benchmarks Dynamic Power

	C-DP(mW/100MHz)	Differences	W-DP(mW/100MHz)	Differences
Min	3.838	3.838	3.91	3.91
Q1	3.91	0.0071	3.91	0
Med	3.979	0.039	4.216	0.305
Q3	4.31	0.331	4.698	0.481
Max	6.291	1.98	17.812	13.114

To begin with, the minimum is observed for Composite architecture. It is at 3.8mW/100MHz. In comparison Wave-core architecture is at 3.9mW/10MHz, therefore both Composite and Wave-core have the same minimum dynamic power. This because many benchmark's cores have only one state and so they are completely identical. The median for Composite architecture is around 3.9mW/100MHz whereas for Wave-core it is almost 4.2mW/50MHz. This suggests that Composite consumes slightly less power than Wave-core. Finally the upper quartile for Composite is at 4.3mW/100MHz. In comparison the median for Wave-core is circa 4.6mW/100MHz. This implies that the most populated core in Composite architecture exceeds the median and so Composite does not perform much better than Wave-core

With respect to Figure 3.21 Composite architecture appears slightly better that Wave-core. To better understand and analyse in detail the amount of increasing power in Wave-core architecture see Figure 3.22.

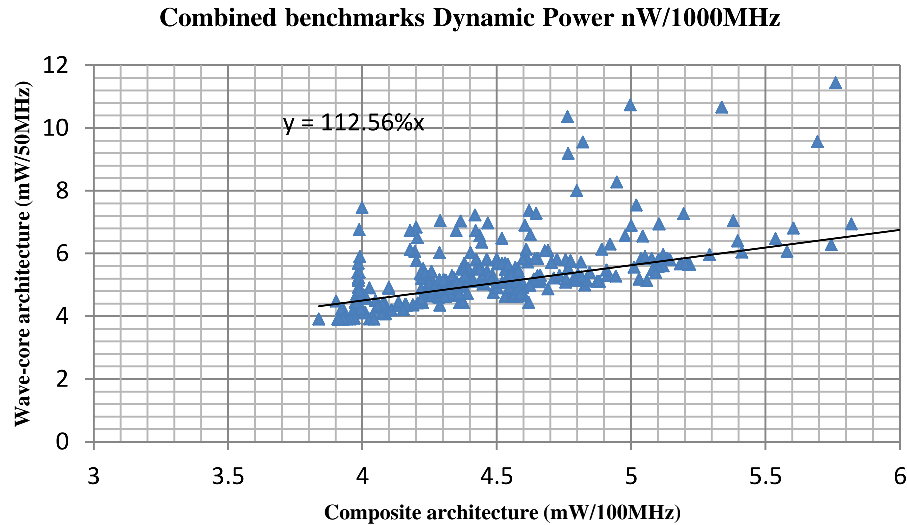


FIGURE 3.22: Combined benchmarks Leakage (nW). It shows that the Wave-core increases the Dynamic Power around 12 %

Figure 3.22 confirms that the difference in dynamic power between Composite and Wave-core is small. So to approximate, while area increases by about 137 % the dynamic power increases by only a small percentage around 12 %.

It suggests that the choice of cores to be implemented is in part influenced by their functionality. Cores with a single state are not likely to be useful unless used excessively by the CPU in alternative software contexts. To be more realistic, the following Section 3.11 uses only cores with three states, those most likely to be selected for implementation in a real system.

3.9 Composite vs Wave-core Total power

To fully understand Wave-core architecture and compare with Composite, this section investigates the total power. To calculate the total power, the dynamic power plus the static power is summed.

The Wave-core architecture has already shown consumes more leakage power and dynamic power compared with Composite architecture, which therefore increases the total power. However, at this stage a comparison between both architectures Composite and Wave-core is done to analyse the differences. Figure 3.23 shows the combined seven benchmarks for the total power. The differences quartile is shown in Table 3.5.

TABLE 3.5: Combined benchmarks Total Power differences quartile

	C-TP(mW/100MHz)	Differences	W-TP(mW/100MHz)	Differences
Min	3.829	3.839	3.911	3.911
Q1	3.911	0.071	3.911	0
Med	3.981	0.069	4.219	0.307
Q3	4.312	0.331	4.703	0.484
Max	6.293	1.98	17.913	13.209

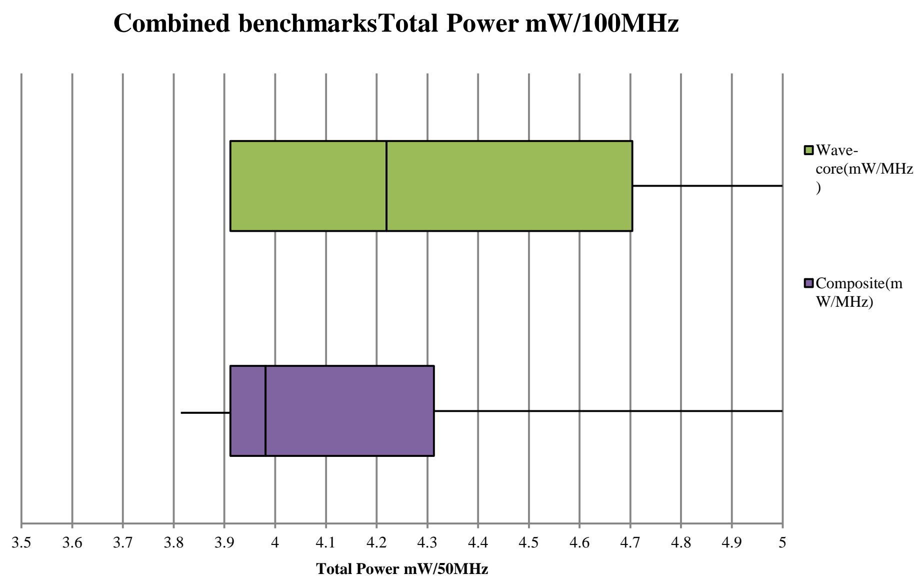


FIGURE 3.23: Combined benchmarks Total power (mW/100MHz). It shows the different quartile to understand which architecture does better. The graph demonstrates that Composite consume less power compared with the Wave-core. But the difference is very minimal.

As noted earlier, the total power is almost identical to dynamic power. In more detail, Composite and Wave-core implement cores consuming about the same energy. With this result It has proved that even though Wave-core increases the leakage with the increasing of logic, it does not affect the total power too much. This is because it increases in area/logic and slightly in active logic per time.

So far, the baseline benchmarks is considered with a high percentage of cores implemented with just one state. However,The Wave-core has been designed with the intention to be applied to the function with a higher number of states, because just with higher number of states the intermittently of transistor helps to decrease the power. For this reason a more accurate reading is considering only the benchmarks that implement the cores with three or more states.(See section 3.10)

3.10 Composite vs Wave-core Power Density

An important aspect of power is power density. The power density has been explained for embedded systems is more important than the power consumption. Ideally Wave-core architecture and its scheduling of active logic spread around the chip should help to decrease the power density.

To understand better how both Composite and Wave-core architectures act in respect of power density. This section compares the combined set of seven benchmarks when the cores are active at 0.5 % of time. Figure 3.24 shows the total benchmarks for the power density. The differences quartile is shown in Table 3.6

To compare the architectures, at first the median is observed. The median for Composite is at around 0.099 W/cmˆ 2, while the median for Wave-core is around 0.005W/cm ˆ 2. This is the first proof that Wave-core performs better in terms of power density.

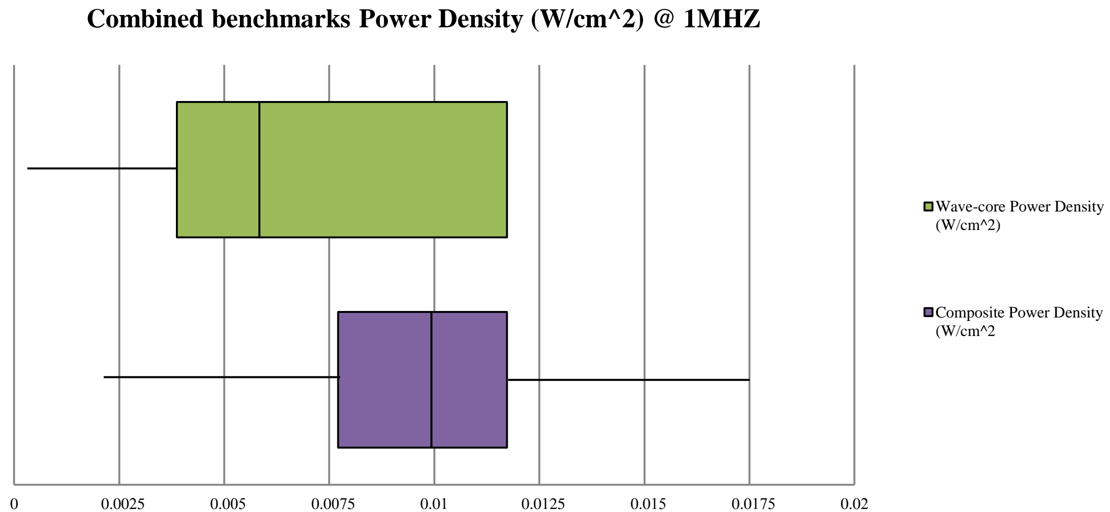


FIGURE 3.24: Combined benchmarks Power Density (W/cm ˆ 2 @1MHz). The Wave-core architecture considering the different quartile decreases the power density about 50 %.

TABLE 3.6: Combined benchmarks Power Density differences quartile

	C-PD(W/cmˆ 2)	Differences	W-PD(W/cmˆ 2	Differences
Min	0.0020	0.0020	0.064	0.006
Q1	0.0077	0.0056	0.003	0.0032
Med	0.0099	0.069	0.0058	0.0019
Q3	0.0117	0.0022	0.011	0.005
Max	0.017	0.062	0.011	0.6E-06

In addition, the first quartile of the cores for Composite is around 0.007W/cmˆ 2, which is above the median for Wave-core. This means that more than 50% of cores waste more energy in Composite architecture

As a final analysis, Composite has more cores above the median than does Wave-core. In fact, Composite cores are centred in the region of 0.007W/cmˆ 2 and 0.011W/cmˆ 2, while the median for the Wave-core is just about 0.005W/cmˆ 2.

Finally, the minimum and maximum for both architectures are observed. As Wave-core is seen still consuming lower power density, as a result the data confirm that the Wave-core helps to improve the power density.

To conclude, Wave-core helps the architecture to decrease the power density without significantly increasing the total power or core cycle time Figure 3.23 and Figure 3.18. To be more precise, considering the quartile, the power density improves by around 50 % and the power consumption declines by about 10 % graphs Figure 3.24 and Figure 3.23.

3.11 Composite vs Wave-core Dynamic power for cores with Multiple States

So far the Composite and Wave-core has been described with the idea to design the Wave-core to improve the power density.

The Wave-core architecture should stabilise the power using a separate logic. The splitting of logic aims to reduce the active logic per time. The Wave-core also uses a memory interface at every stage boundary.

In specific, the characteristic of Wave-core suggests that is more suitable for functions with a higher number of states. Therefore, to better understand how Composite and Wave-core architectures behave, this Section extracts from the data only the cores that are implemented using more than two stages. The dynamic power for the cores with a state greater than two is shown in Figure 3.25. The differences quartile is shown in Table 3.8

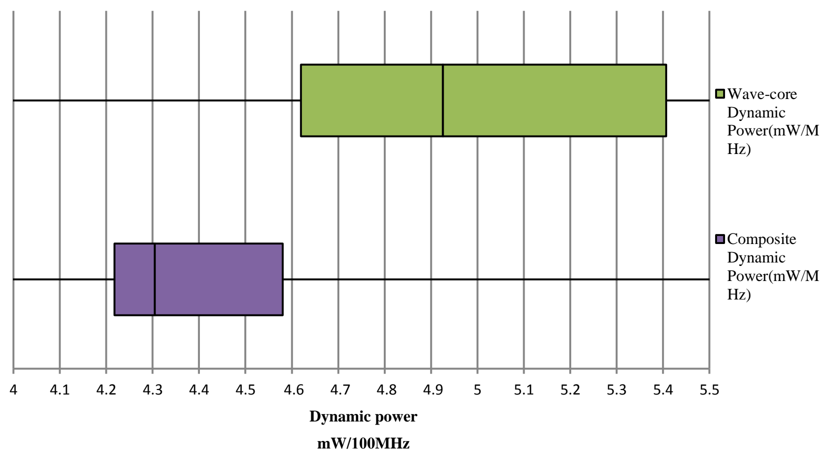


FIGURE 3.25: Composite vs Wave-core dynamic power with more than two states. It appears that the Composite code implements the cores using less dynamic power.

TABLE 3.7: Combined benchmarks Dynamic Power multi states differences quartile

	C-DP(mW/100MHz)	Differences	W-DP(mW/100MHz	Differences
Min	3.902	3.902	3.901	3.901
Q1	4.217	0.314	4.619	0.709
Med	4.304	0.086	4.925	0.305
Q3	4.581	0.276	5.407	0.481
Max	43.587	39.007	66.005	6.06E+01

To understand the architectures, and observes which one does better the whiskies and box graph is used, because it gives soon an overview. The median is the first characteristic to observe because it is the first sign to say which one consume less

power. The median for Composite is at 4.3mW/100MHz and the median for Wave-core is at 4.9mW/100MHz. This implies that Wave-core consumes more energy.

To confirm this the high quartile for Composite and Wave-core is observed. Composite has the high quartile at almost 4.5mW/100MHz, on the other side the median of Wave-core is at 4.9mW/100MHz.

This implies than the majority of cores in Composite architecture consume less dynamic power than Wave-core architecture.

To conclude, the minimum and the maximum for both Composite and Wave-core architectures are compared. Composite has its minimum at 3.9mW/50MHz while Wave-core has its minimum at 3.9mw/100MHz – the same value, despite the maximum for Composite being at 43mW/50MHz and that of Wave-core being at 66mW/100MHz.

In brief, thus far the data clearly proven that Wave-core architecture and its multiple stages do not improve the dynamic power, but in the next section, using optimised benchmarks, the situation appears to change.(See Chapter 4, power section) On the other hand, comparing the power density, the Wave-core decreases the amount of power per area, see Figure 3.26. The differences quartile is shown in Table 3.8

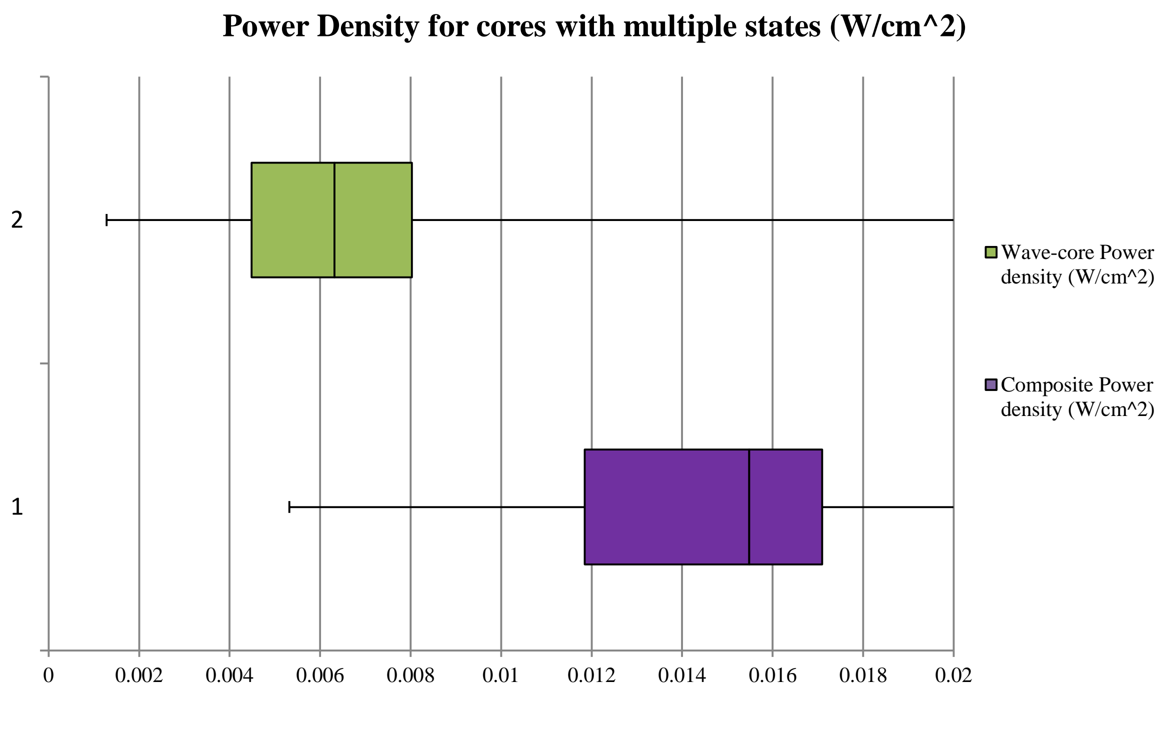


FIGURE 3.26: Power density for cores with multiple states. The Wave-core architecture reduces the power density.

In detail, the Wave-core architecture according with Figure 3.26 decreases the power density around 50 %. The majority of cores in Wave-core are grouped between 0.0044 and 0.008 W/cm ˆ 2. In comparison the low quartile in Composite is at 0.017 W/cm ˆ 2 that is higher than the high quartile Wave-core. This prove that the Wave-core architecture decreases the power density.

Another way to analyse the energy efficiency of the cores, is to measure the amount of power used in a million of instructions per second (MIPS/mW). Figure

TABLE 3.8: Combined benchmarks Power Density multi states differences quartile

	C-P(W/cmˆ 2)	Differences	W-PD(W/cmˆ 2	Differences
Min	0.0053	0.0053	0.0012	0.0012
Q1	0.0118	0.0065	0.0044	0.0032
Med	0.0154	0.0036	0.0063	0.0018
Q3	0.0170	0.0016	0.0080	0.0017
Max	0.0350	0.0179	0.0234	0.0154

3.27 compare the MOPS/mW for the Composite and Wave-core architectures. According to the result the Wave-core core uses less energy per million of operation per second.

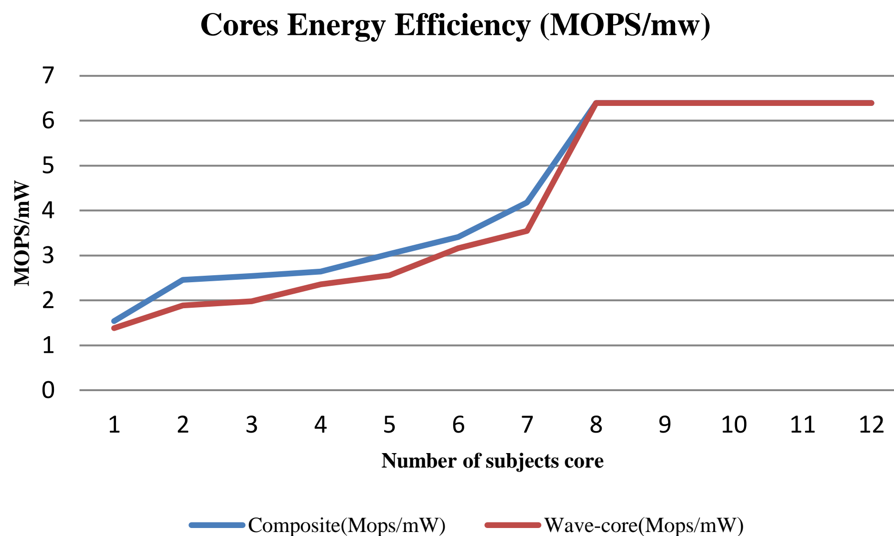
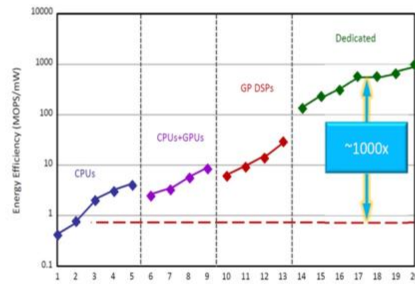


FIGURE 3.27: Comparing the Composite with the Wave-core. the Wave-core uses less energy per million of instruction per second

To calculate MIPS/mW: Assuming 3911874.629 nW/100MHz. To know the number of nW per cycle, 3911874.629 is divided for the number 100MHz, so it is equal to 0.0782. At this point to know the number of operations, 0.0782 is multiplied for the number of instruction IPC. Assuming IPC equal to 1 the amount of power per operation is 0.0782nW. To know the amount of power per million per second, 0.0782 is divided for 1000, that is equal to 12781.6uW. To know how many mW per second 12781.6 is multiplied per 1000, that is equal to 12781596 mW. Finally to know MIPS/mW, 12781596mW is divided per 1000000. so MIPS/mW is equal to 12.7 [9]

Figure 3.28 show the amount of MIPS/mW for different hardware configuration. It shows that standard CPU executes less than 10MOPS/mw while CPUs+GPUs executes up to 10MOPS/mW. GP DSP executes up to 50MIPS/mW and dedicate hardware goes up to 1000MIPS/mW. Comparing this data with core energy efficiency for both Composite and Wave-core, the cores are in the range of CPUs + GPUs and for some of them the one that does better get inside the digital signal processing.

Energy Efficiency for different CPU



Cores Energy Efficiency

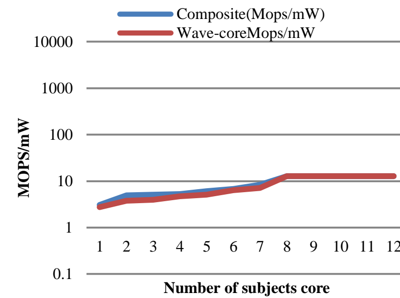


FIGURE 3.28: comparing the energy efficiency for different Hardware with core energy efficiency (MOPS/mW)

3.12 Summary:

This Chapter started by observing the characteristics of the cores generated by the translator tool in term of Inputs/Outputs, number of states and IPC. According to the results the majority of cores require up to 4 inputs/outputs. This suggests that to interface the core to a host CPU a small stack memory and a simple 4 variables 2 ways bus communication to transfer the data from and to memory are needed. The other two important characteristics are: the number of states to perform the FSM and Instruction per Clock Cycle. They describe the number of clock cycle needed to perform the function and the number of operations each state executes.

A novelty of this work is to apply a Stack concept to generate hardware IPs. A weakness of this technique is that it does not execute data in a pipeline, therefore the accelerators generated appear inefficient in fast data execution. However, hardware executes data in concurrency. For this reason, the number of IPC can play an important role in creating faster cores. So far the number of IPC observed was generated from the baseline benchmarks. The baseline benchmarks are software functions generated from a no optimized compiler. This implies a lower number of IPC. but according to the data the cores generates require an entity with few inputs/output and a small stack memory. Thus encourages to investigate in more details in static parallelism to increase the number of IPC, because it appears to be a very advantageous technique for performing fast data execution and decreasing power consumption. (See Chapter 4, power section for more details)

Another and most important novelty introduced in this Chapter is the Wave-core architecture. The Wave-core has the characteristic that it increases the area and so decreases the power density, which is one of the most important factors in designing an embedded system today, without decreasing the timing when compared with the Composite architecture.

To summarize; The Wave-core generates the majority of cores with the area in the rage of 1668um ˆ 2 to 6000um ˆ 2 which is around 2.3x bigger than Composite. The timing remains between 800MHz and 1GHz for both Wave-core and Composite. The leakage stays between around 1500nW to 4500nW that is 2.4x bigger that the Composite. That is to say; leakage plays an important role in an embedded system and battery life duration, therefore the Composite appears to have a better performance. In reality, however the leakage is very small when compared with

the dynamic power, the total power absorbs this difference. Furthermore, the dynamic power for a good fraction of cores appears to be the same in both Composite and Wave-core architecture. But for cores with a high number of states Composite improves the dynamic power for few percentage. In fact the dynamic power for the majority of cores with more than two states in Composite architecture is in the range of 4.3mW/100MHz to 4.5mW/100MHz. In contrast the dynamic power for the Wave-core is in the range of 4.6mW/100MHz to 5.4mW/100MHz

To conclude, the result proves Wave-core architecture improves the power density by around 50 %.

Chapter 4

Impact of Code Optimisation (Stack Scheduling Code)

Processors are being produced with the potential for very many parallel operations on the instruction level... Far greater extremes in instruction level parallelism are on the horizon [19]

With these words Joseph Fisher introduced Instruction Level Parallelism 1981, and with these words this chapter begins.

Chapter 2 introduced different techniques to reduce power consumption and so ameliorate the Dark Silicon problem. It also explained that many techniques are able to move software into hardware. The same concept was expressed by Yan Solihin, North Caroline State University with these words

We have to improve performance by improving energy efficiency. The only way to do that is to move software to hardware [45]

Another similarity is that they implement this small hardware core without using the pipeline and register file technique because these two techniques consume a lot of energy, see Chapter 1.4.1

To translate software into hardware and improve the hardware development life cycle many techniques start the process at compiler level, an example could be GreenDroid project [61] but also PIC: automatically designing custom computers [66]

This work has introduced these concepts but using a novel approach and so applied a stack model that automatically avoids the pipelining and register file techniques, it then considered a traditional state machine to design hardware that here is called, Composite core,(See chapter 2.7). it has also introduced the Wave-core architecture, (See Chapter 2.8)

To produce better machine code to be translated into hardware, this Chapter introduces the stack scheduling technique. Stack order can affect a calculation and the use of memory [36], therefore this section seeks to understand if changing the order of operands can improve the execution data flow and result in the translator tool producing a better hardware IP cores for both Composite and Wave-core.

Before getting inside the topic, it should be clear that there are two approaches to implement parallelism, dynamic that refers to hardware parallelism, pipeline etc., and static that refers to software parallelism, instruction scheduling, etc. [19]. Compiler optimization technique today is widely applied in compiler to produce better machine code. It improves executable timing code minimizing memory occupied [19].

This Chapter introduces three stack scheduling techniques to improve instruction level parallelism and so improve performance of machine code.

An emerging technique for eliminating unwanted memory references in the stack machine is stack scheduling [36]. To date, three major techniques have been devised:

Koopman, Bailey and Shannon [36] [56] [3]. These could have a significant impact on how our cores behave, therefore it is important to investigate this as part of our analysis.

To summarize the three techniques, Koopman looks for local variable scheduling within basic blocks, Bailey Intra-Block scheduling percolation and Shannon Global stack allocation. See Figure 4.1

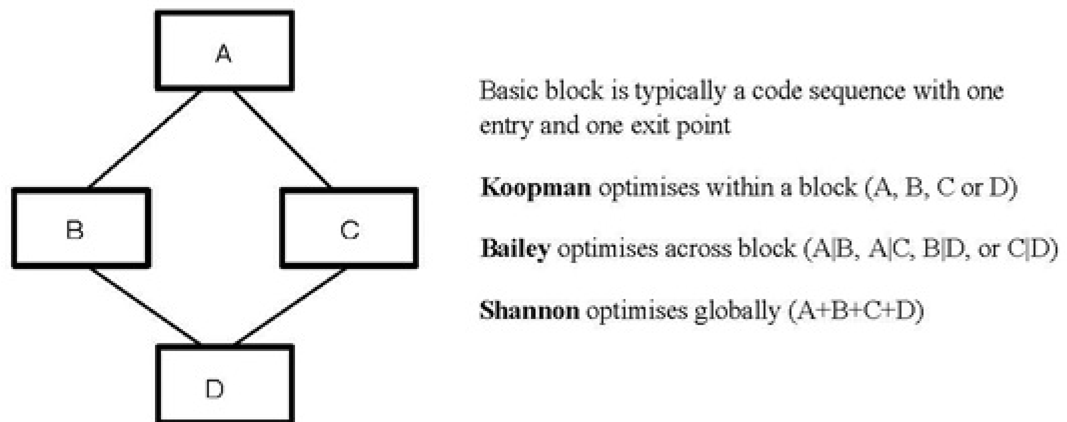


FIGURE 4.1: advance stack optimization. It shows three different stack scheduling techniques

Figure 4.1 shows three different stack scheduling techniques and so assumes a data flow tree and that each block represents a sequence of instruction with one entry and one exit point. Koopman tries to eliminate unwanted memory considering a single block at the time. Bailey extends the range and so considers block A | B, A | C, B | D or C | D. Shannon look at the sequence of instruction globally and so considers A+B+C+D.

To understand if these optimizations help to design better IP cores, it looks for some key characteristics: input/output complexity, IPC and number of states. I/O describes the number of operands needed from the stack memory that involved the communication bus memory to core and vice versa. Number of states and IPC describe the number of clock cycle to implement a function and the number of parallelism fount in each function. These are relevant factor in decreasing power consumption [19]

To understand how these three optimization impact on the hardware cores, the number of inputs and outputs are compared with the baseline cores seen in Chapter 3, after that the number of states inside the aforementioned state machine, see Chapter 2.5.1, and finally Instruction Level Parallelism.

4.1 Comparison of Optimised Cores' Input

The general idea of this new architecture model is to understand how the hardware, cores, behave if they are synthesised using machine code optimised by a stack scheduling algorithm. Chapter 1.1 introduces the concept of Stack CoDA architecture, while Chapter 3.1 introduces the number of inputs of the IPs hardware cores

as the number of items to be collected from the stack memory of the CPU. Therefore the first characteristic to observe is how this is affected, because the number of inputs describe which kind of communication the architecture needs to communicate memory stack to IP core and vice versa

4.1.1 Input operands per core (Smaller benchmarks, with optimization)

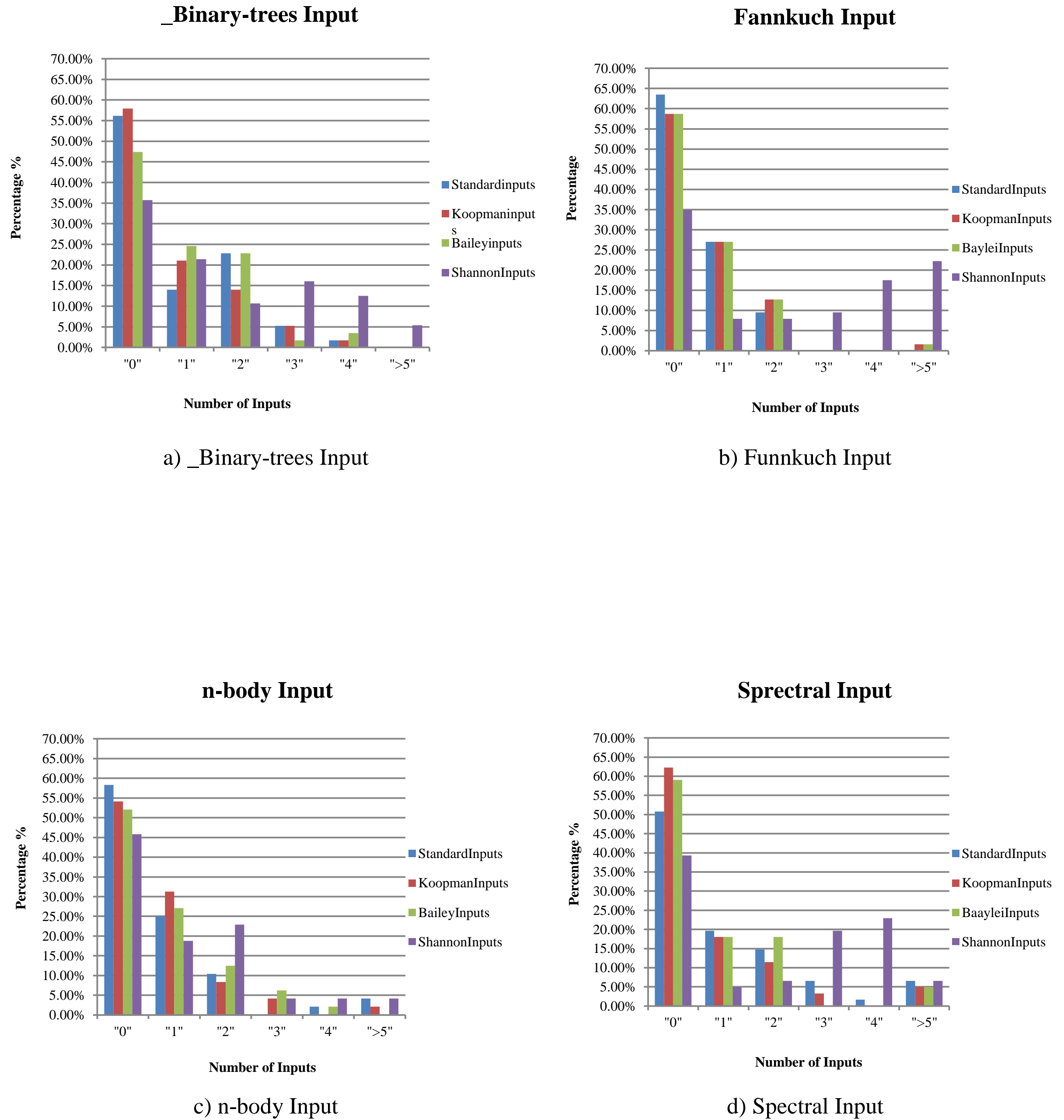
Figure 4.2 shows the comparison of input/s between the cores implemented using the standard, the Koopman, the Bailey and the Shannon optimisation algorithms (standard = no optimised benchmarks.) for the smaller benchmarks

In Figure 4.2 a) the Binary-tree's input has the spread of inputs between one and four. In detail, the standard cores have approximately 50 % of IP hardware cores with zero input. In comparison Koopman increases this to a percentage of around 60 %. However, Bailey scheduling decreases the percentage to around 57 %. Finally, almost 35 % Shannon cores have zero input. Again, nearly 15 % of standard cores have an output of one, while this increases for the Koopman to about 20 %. Bailey has nearly 25 % of cores with one input. Finally, Shannon optimisation has around 20 % of cores with one input. Circa 25 % of cores in Standard and Bailey benchmarks have two inputs. In contrast Koopman has circa 15 % with two inputs. To conclude Shannon has around 10 % of cores with two inputs. The Standard benchmark and Koopman have 5 % of cores with three inputs, Bailey has nearly 2 % of cores with three inputs and Shannon has slightly more than 15 % of cores with three inputs. Standard, Koopman and Bailey have less than 5 % of cores with four inputs, while Shannon has slightly more than 10 % of cores with four inputs and 5 % of cores with five inputs.

The percentage of inputs inside the Funnkuch Figure 4.2 b) appears almost the same as in the Core benchmarks. Around 65 % of cores for the Standard, Koopman and Bailey optimisations have zero inputs nearly 60 %, while for Shannon this is just 35%. Circa 25 % of cores have one input, where in contrast Shannon has just around 10 %. Finally, with 10 % three optimisations have 2 inputs. To conclude, just Koopman and Bailey have a very small percentage of cores >5 inputs, while Shannon has around 10 % of cores with three inputs, slightly more than 15 % with four and slightly more than 20 % of cores with >five inputs.

The n-body Figure 4.2 c) benchmarks present the majority of cores with zero input. Among 20 % and 30 % of the four optimisations have one input, while Standard, Koopman and Bailey have around 10 % with two inputs. Shannon has nearly 25 % of cores with two inputs. Shannon optimisation presents its cores without three inputs. However, in the Standard, Koopman and Bailey optimisations 5 % of cores have three inputs. Again with almost the same percentage Standard, Bailey and Shannon have four inputs. Finally, just a few cores in the Standard, Koopman and Shannon optimisations have >five inputs.

The Spectral Figure 4.2 d) is the last benchmarks to consider. It presents a spread of cores between zero and five. Notably, the Bailey has no cores with three or four inputs, while the Standard has no cores with three or >five inputs. Most cores, between 40 % and 60 %, have zero input. After that Standard, Koopman and Bailey have approximately 20 % of cores with one input, while Shannon has circa 5 % of cores with one input. Standard, Koopman and Bailey have between 10 % and almost 20 % of cores with two inputs while Shannon has just 5 % of cores with one input. Furthermore closely to 5 % of standard and Koopman have three inputs. In comparison Shannon has around 20 % of cores with three and four inputs. Standard optimisation has less than 5 % of cores with four inputs.



a) _Binary-trees Input

b) Funnkuch Input

c) n-body Input

d) Spectral Input

FIGURE 4.2: shows the number of inputs for the smaller benchmarks. The four benchmarks appear to have similar characteristic.

4.1.2 Input operands per core (Larger benchmarks, with optimization)

Figure1 4.3 shows the comparison of inputs between the cores implemented using the standard, the Koopman, the Bailey and the Shannon optimisation algorithms (standard = no optimised benchmarks.) for the bigger benchmarks

The Standard and the Bailey optimisations in the Fasta Figure 4.3 e) benchmark have nearly 50 % of cores with zero inputs. In comparison, about 60 % of cores in the Koopman optimisation have zero input. To conclude, circa 30 % of cores have zero inputs. Cores with one input in the four optimisations are in the range of 20 % to 25 %. Between circa 10 % and 15 % of the four optimisations have two inputs; Standard, Koopman and Bailey have in the range of 0.1 to 0.5 of three inputs, while Shannon increases this percentage up to about 15 %. Notably, only the Standard and Shannon optimisations have cores with four inputs. Finally, circa 5 % of cores have >five inputs.

The Core benchmark Figure 4.3 f) has the number of inputs between zero and five. The majority of cores in the Standard, Koopman, Bailey and Shannon optimisations have zero inputs. Again, Standard, Koopman, and Bailey present the same percentage of cores with one input around 25 % while Shannon has circa 20 %. Koopman has about 5 % of cores with two inputs, while the Standard and Bailey optimisations have circa 10 % of cores with two inputs. Also, Shannon has slightly more than 10 % of cores with two inputs. Finally, with almost the same characteristics, Standard, Koopman and Bailey have a few cores with three, four and >five inputs. In contrast, Shannon has around 10 % of cores with three and five inputs and almost 15 % of cores with four inputs.

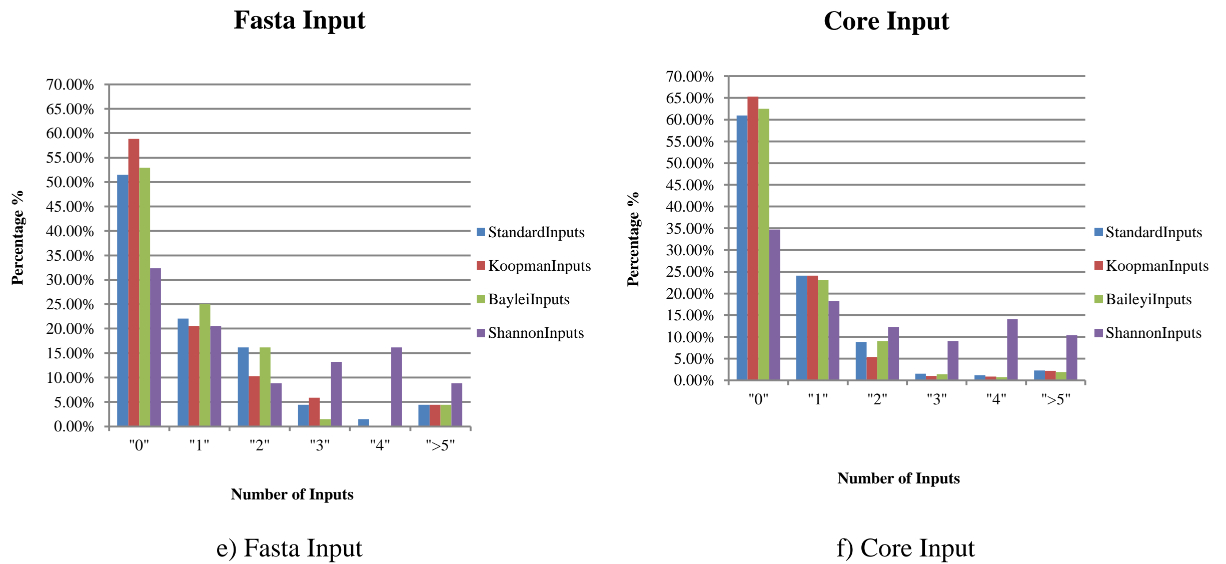
Scimark benchmark Figure 4.3 g) has the most cores of the four optimisations grouped at zero input. Between 10 % and 15 % of cores have one and two inputs for the four benchmarks. Around 5 % of cores have three. A very small percentage of cores have four and >five inputs. An exception is Shannon, which has around 10 % of cores with three and >five inputs and more with almost 15 % of cores with four inputs.

4.1.3 Input operands per core (all benchmarks, with optimization)

Observing the overall benchmarks Figure 4.4, Standard, Koopman and Bailey present similar characteristics in term of input complexity. However, Shannon significantly increases the percentage of the number of inputs. This is because Shannon increases the IPC and by increasing the parallelism, increases the number of inputs. The number of input, in other words are the number of variables needed to process the function. Another observation, however is that, the number of inputs increases the majority of core within a range of four inputs. This means as explained in Chapter 3 that all optimization can use a similar bus of communication memory to IP cores and vice versa. Furthermore, it does not need a complex network or many clock cycle to process a function. Indeed, the number of inputs suggests an opportunity for parallelism that each optimisation extrapolates from each core. This is shown to be the case in later sections.

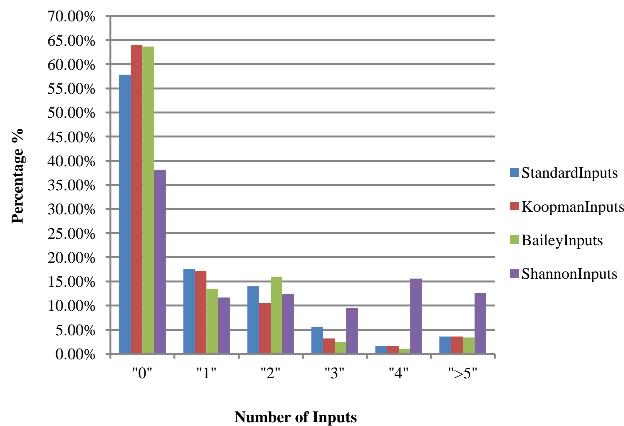
4.2 Comparison of Optimised Cores' Output

The number of values to be pushed back on the stack of the CPU from a core is its output complexity. To understand how the number of items change for the four optimisation cases, how the numbers of outputs change for each benchmark applying



e) Fasta Input

f) Core Input



g) Scimark Input

FIGURE 4.3: Number of inputs for the bigger benchmarks. The three benchmarks appear to have similar characteristic. Shannon scheduling optimization has an increasing of number of inputs, It is because increases the number of IPC.

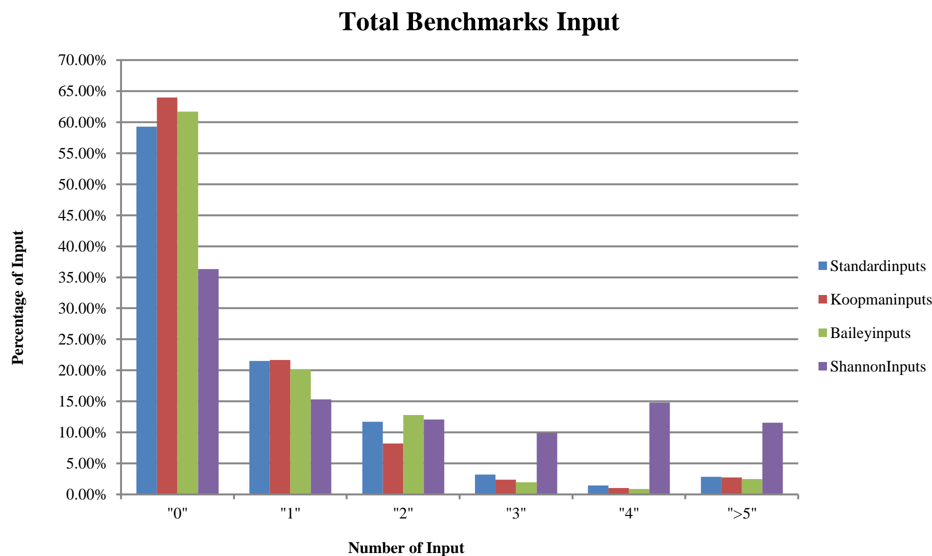


FIGURE 4.4: Total Benchmarks Input. According with the data, Shannon has clearly an increase of IPC because the number of inputs increases

the Standard, Koopman, Bailey and Shannon optimisations is investigated. Figure 4.5 and Figure 4.3 show the total number of outputs for each benchmark.

4.2.1 Output operands per core (Smaller benchmarks, with optimization)

The Binary-tree Figure 4.5 a) for the four optimisations shows a higher percentage of cores at zero output. Standard, Koopman and Bailey have one output with almost the same percentage. In contrast Shannon has about 10 % of cores with one output. Bailey optimisation has around 15 % of cores with two outputs, while Koopman and Standard optimisation have circa 10 %, and Shannon around 12 %. Standard, Koopman and Bailey optimisation have three and four outputs at an equal, very small percentage. Shannon has around 15 % of cores with three and four outputs. Finally, Shannon has 10 % of core with >five outputs.

The Funnkuch benchmark Figure 4.5 b) has Standard and Koopman optimisation with approximately 75 % of cores with zero output, while with around 60 % the Bailey has zero output and more with nearly 30 % Shannon has zero output. Standard, Koopman and Bailey have a very small percentage around 3 % with two and three outputs. Furthermore, Shannon has around 3 % of cores with two outputs, about 10 % of cores with three outputs, almost 35 % of cores with four outputs and finally less than 5 % of cores with >five outputs.

The n-body benchmark Figure 4.5 c) has the majority of the cores with zero output. In fact, Standard and Koopman have almost 80 % of cores with zero output while Bailey has in the range of 60 %, and Shannon has circa 40 % of cores with zero output. Bailey and Shannon optimisations have almost 20 % of cores with one input. In spite of this Koopman and Standard have around 10 % of cores with one output. In more detail, Koopman has no cores with three outputs, and similarly the Standard has no cores with four and five outputs. A very small percentage of cores (around 5 %) have three, four and >five outputs for the three optimisations. Finally, Shannon

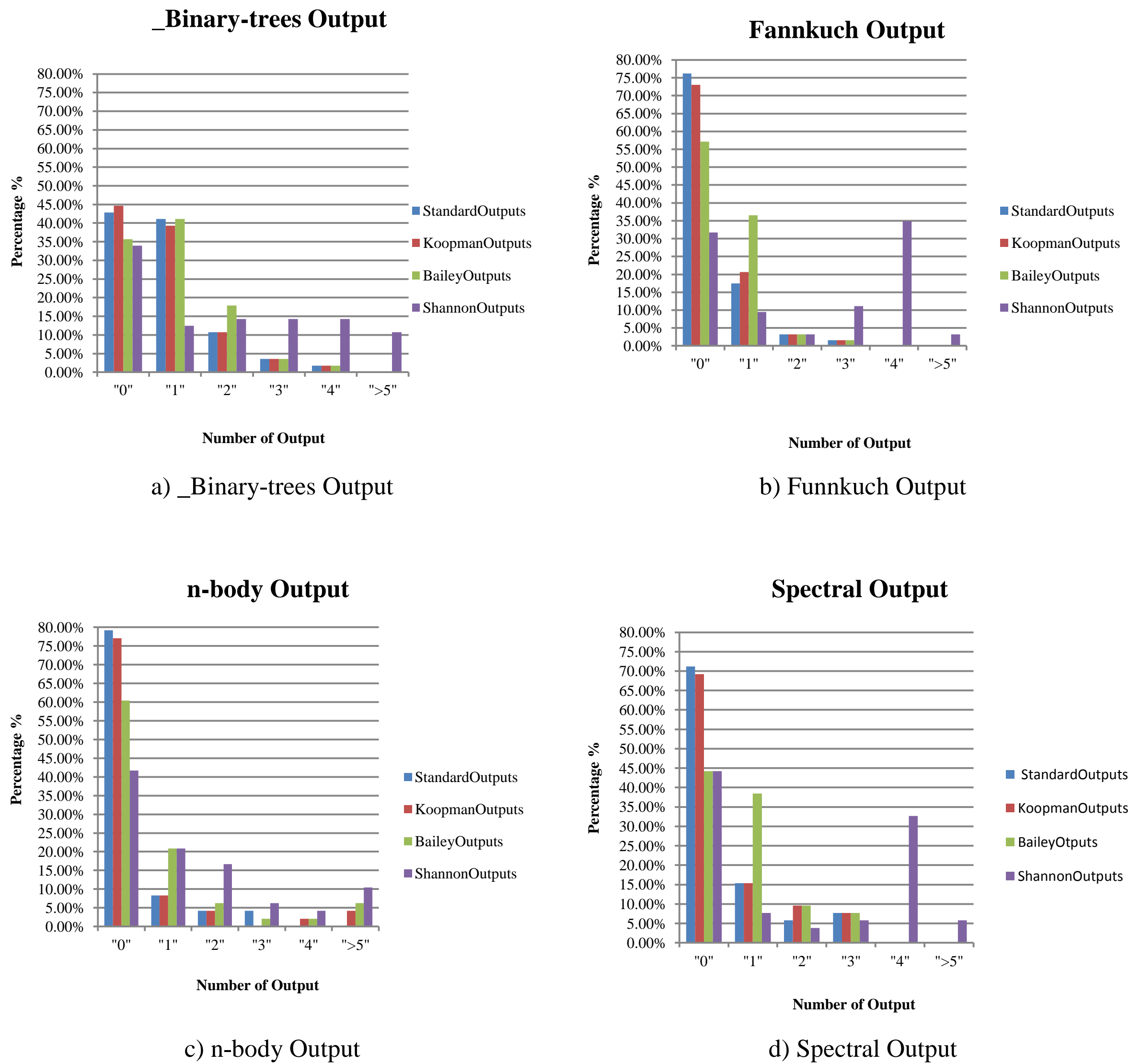


FIGURE 4.5: shows the number of Output for the smaller benchmarks. The four benchmarks appear to have similar characteristic.

has 15 % of cores with two output, around 5 % with three and four outputs, and almost 10 % of cores with >five outputs.

The Spectral benchmark Figure 4.5 d) has Standard and Koopman with almost 70 % of cores with zero output. Despite this Bailey and Shannon optimisations have circa 40 % of cores with zero output. Further, Bailey has almost 40 % of cores with one output. Alternatively, Standard and Koopman optimisations have around 15 % of cores with one input and the Shannon slightly more than 5 %. Again, nearly 10 % of cores optimised for Koopman and Bailey have two outputs, while Standard and Shannon have around 5 % of cores with two outputs. All four optimisations have around 8 % with three outputs, while Shannon has circa 35 % of cores with four outputs and 5 % of cores with >five outputs.

4.2.2 Output operands per core (Larger benchmarks, with optimization)

Figure 4.6 show the output for the bigger size benchmarks.

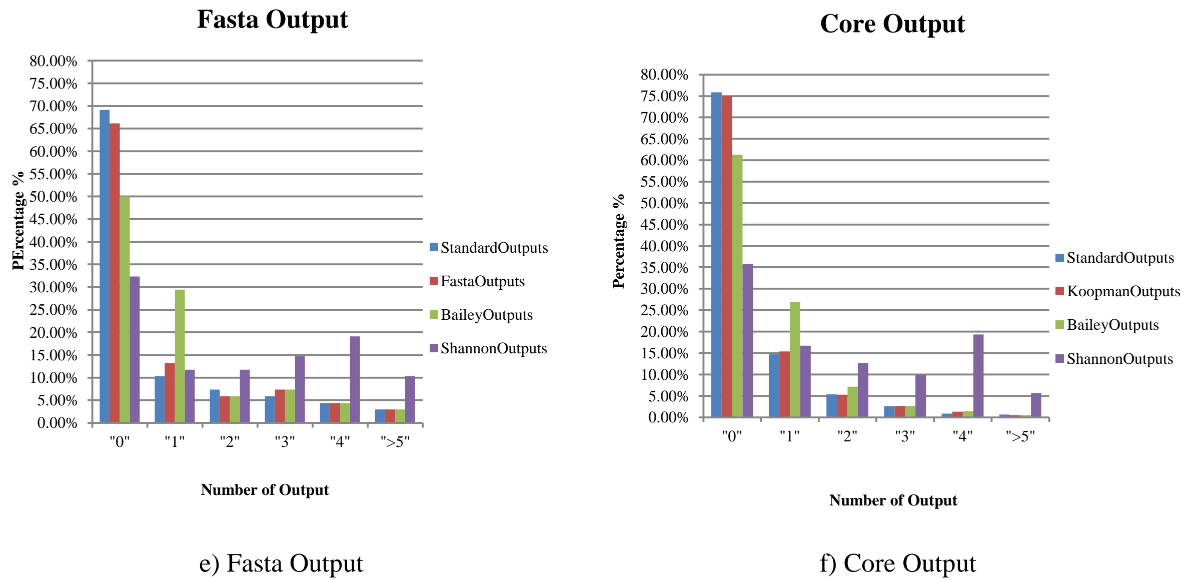
The Fasta benchmark Figure 4.6 e), like all others, has the majority of cores at zero output. In more detail, Standard and Koopman have almost 70 % of cores with zero output. However, Bailey has circa 60 % of cores with zero output and Shannon has around 30 % of cores with zero output. Again, the Bailey optimisation has around 30 % of cores with one output. In comparison, Standard, Koopman and Shannon optimisations have almost 10 % of cores with one output. Finally, a small percentage (around 5 %) have two, three, four and >five outputs for Standard, Koopman and Bailey. In comparison, Shannon has slightly more than 10 % two and >five outputs, around 15 % three outputs and finally nearly 20 % four outputs.

The Core benchmark Figure 4.6 f) has the majority of cores with zero output: around 75 % for Standard and Koopman optimisations. Bailey has a smaller percentage (circa 60 %) with zero output. Finally, Shannon has circa 35 % of cores with zero output. Standard, Koopman and Shannon have around 15 % of cores with one output, while Bailey has slightly more than 25 % of cores with one. Again, Standard, Koopman and Bailey have around 5 % of cores with two outputs while the Shannon increases this number by up to 13 %. Furthermore, the Standard, Koopman and Bailey have around 3 % of cores with three four and >5 outputs. On the other hand Shannon has nearly 10 % of cores with three outputs, around 20 % of cores with four outputs and 5 % of cores with >five outputs.

In the Scimark benchmark Figure 4.6 g) again the majority of cores are centred on zero output. After than Standard and Koopman have around 25 % of cores with one output, while Bailey has around 40 % of cores with one output and Shannon has closely to 10 % of cores with one and two outputs. Standard, Koopman and Bailey have around 5 % and less two, three, four and >five outputs. Furthermore the Shannon has circa 15 % of cores with three outputs, almost 20 % of cores with four, and finally about 5 % of cores with >five outputs.

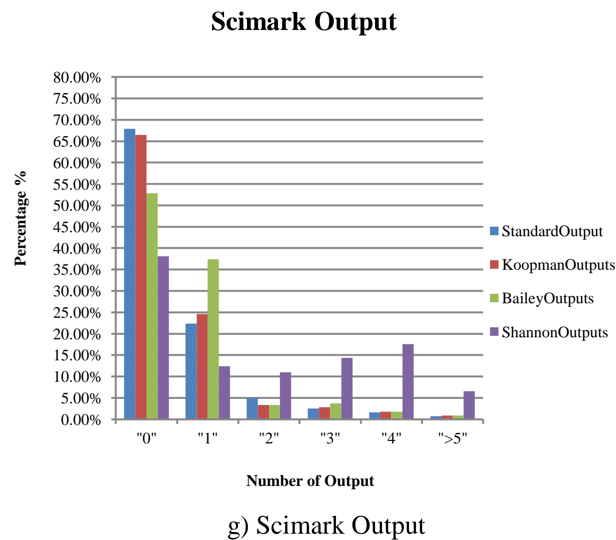
4.2.3 Output operands per core (All benchmarks, with optimization)

Figure 4.7 shows the number of outputs for the four optimised cores. Again, there was a relatively small change for Koopman and Bailey optimisation, but significant differences for Shannon. It is because as is explained in the next sections, Shannon clearly increases the number of IPC, therefore more instruction are executed each time.



e) Fasta Output

f) Core Output



g) Scimark Output

FIGURE 4.6: Output for the bigger benchmarks. The three benchmarks appear to have similar characteristic. The majority of benchmarks has four output..

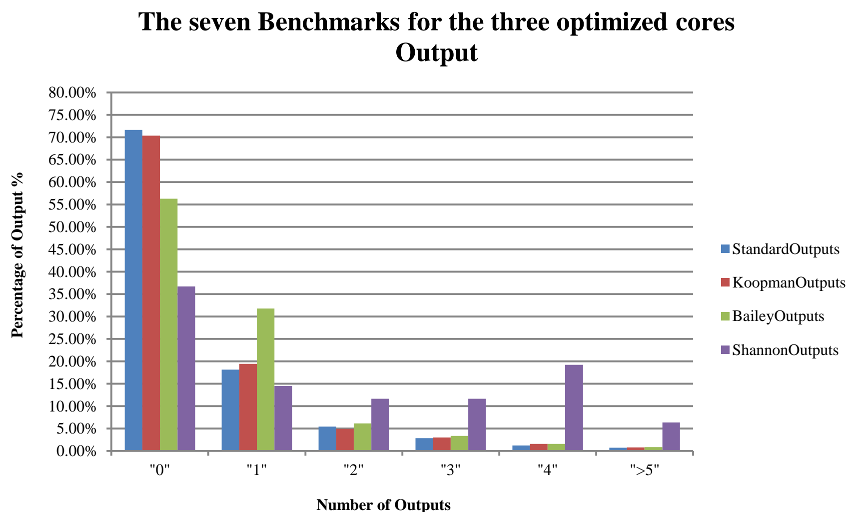


FIGURE 4.7: Output for the seven benchmarks. It shows Shannon optimization increases the number of outputs. It means that Shannon executes more instruction per time

Again, the data shows that Shannon scheduling causes a significant increase in input/output traffic bandwidth. An important observation can now be made; Shannon increases both input and output complexity. However, many of the values coming into a core and then leaving at the exit state may simply be "passing through" in the sense that their position may change in the stack order but not their values. This makes sense since Shannon attempts to keep value on the CPU expression stack in a persistent way for use in a more distant block as it is explained at beginning of this Chapter.

Also these increases of outputs are the evidence that Shannon increases the number of IPC, because performing more operations returns a higher number of results. Also as with the inputs, another important point is that the range of outputs are always inside four. It means that this concept of architecture needs a simple bus of communication with few clock cycles to transfer data from core to memory.

4.3 Comparison of Optimised Cores' States

Sections 4.1 and 4.2 describe an increase in the number of inputs/outputs, which it refers to as the increasing of number of IPC because if a function requires more operands to be performed, it increases the number of instructions per clock cycle. Furthermore, to give another proof of that, the number of states has been observed. When memory references are removed by optimisation, this results in fewer, more complex states and this is a much stronger indicator that there is an increase of IPC.

To observe how the cores behave in this respect, the changes in number of states for each optimisation are observed. By definition [29] comparing two identical functions that are implemented using a different number of states in the generic state machine, it changes the values of power, timing and area. Therefore less states mean the function is implemented with more parallelism, with less clock cycle and the logic is grouped in less area. Figure 4.8 and Figure 4.9 show the seven benchmarks optimised for Standard, Koopman, Bailey and Shannon architecture

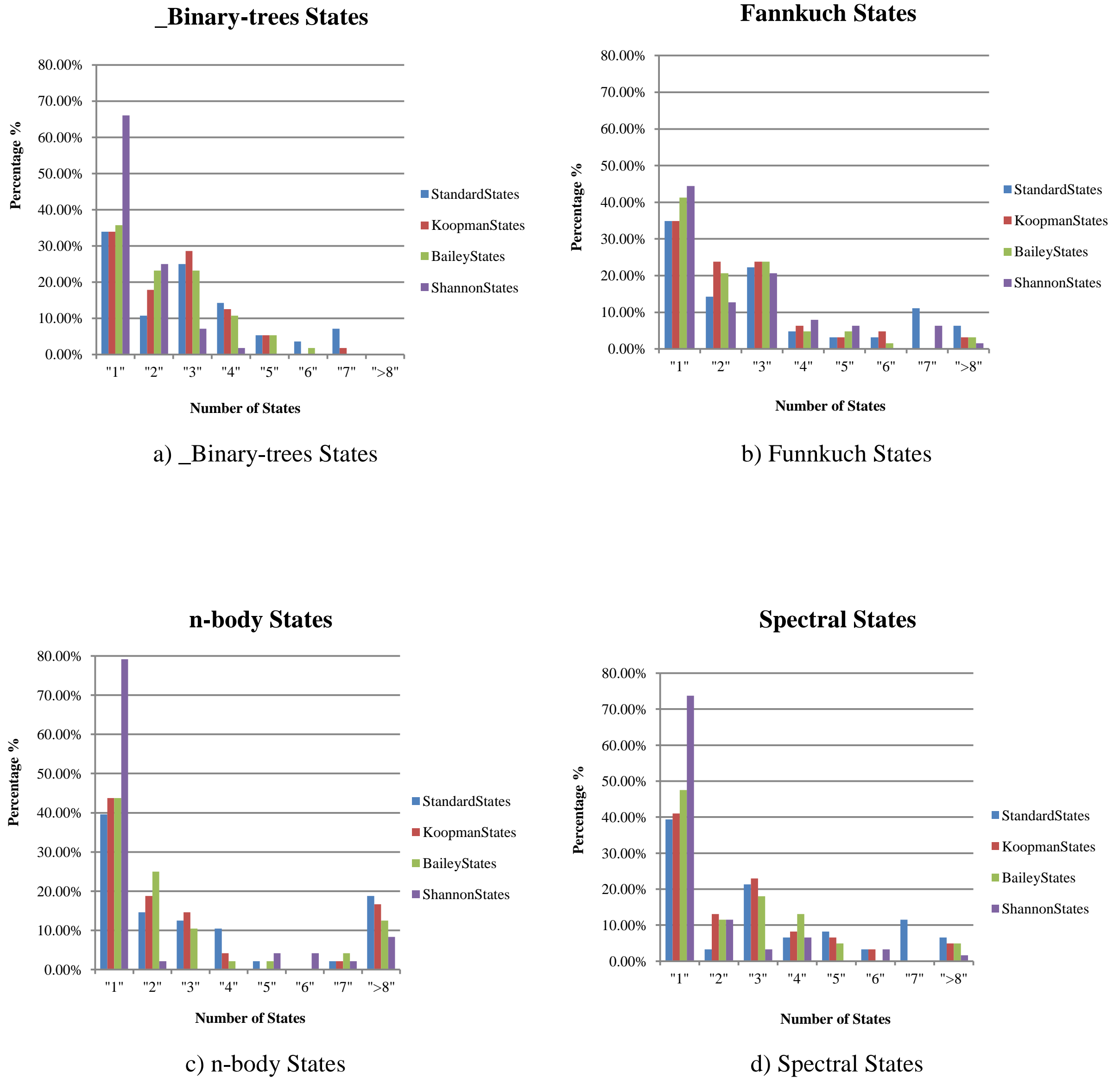
4.3.1 Number of states per core (Smaller benchmarks, with optimization)

The Binary-trees benchmark Figure 4.8 a) has the number of states spread from one to seven. In more detail, around 35 % in Bailey optimisation have one state. In comparison slightly less than 33 % in Standard and Koopman have one state, and Shannon has almost 65 % of cores with one state. Again, around 25 % of cores in Bailey and Shannon have two states, while almost 18 % of cores in Koopman have two states. Finally, just around 10 % of cores in Standard optimisation have two states. Koopman has nearly 28 % of core with three states. In comparison about 22 % in Bailey and Standard have three states, and finally Shannon has less than 10 % of cores with three states. Almost the same percentage closely to 10 % in Standard, Koopman and Bailey have four states, and furthermore Shannon has less than 5 % of cores with four states. Again, around 5 % of cores in Standard, Koopman and Bailey have five states. Only Standard and Bailey have six states, a very small percentage circa 5 %. Finally, in Standard around 8 % of cores have seven states and around 3 % of cores in Bailey have seven states.

The Funnkuch benchmark Figure 4.8 b) has approximately 40 % of cores in Bailey and Shannon optimisation with one state. In comparison, slightly less than 35 % of cores have one state in Standard and Koopman optimisation. Around 20 % in Koopman and Bailey have two states, while circa 15 % in Standard and Shannon have two states. Slightly more than 20 % in the four optimisations have three states. In addition, circa 5 % have four, five and six states for Standard, Koopman, Bailey and Shannon optimisations. Around 10 % have seven states in Standard and around 5 % in Shannon have seven states. Finally, about 5 % in the four optimisations execute cores with >8 states.

The n-body benchmark Figure 4.8 c) has the majority of cores around 45 % in Koopman and Bailey optimisations, and in the range of 40 % in Standard at one state. Furthermore, about 80 % of cores use one state. In addition, Bailey has 25 % of cores with two states; Koopman has circa 20 % of cores with two states; Standard optimisation has nearly 15 % of cores with two states; and Shannon has less than 5 % of cores with two states. Almost 15 % of cores have threes state for the Standard and Koopman benchmark while Bailey has 10 %. Again, 10 % of cores for the Standard have four states and a very small percentage of cores for the other optimisations closely to 5 % have four states. In addition, Koopman has no cores with five states, while about 5 % of cores have five states in Standard, Bailey and Shannon optimisations. A very small percentage, less than 5 % have seven states for the four optimisations. Almost 15 % of cores have >eight states in Standard and Koopman optimisations, a slightly lower percentage around 10 % have >eight states in Bailey, and finally less than 10 % of cores in Shannon have >eight states.

Spectral Figure 4.8 d). As with the others, this benchmark also has the greatest percentage of cores implemented within one state. In fact, around 50 % of cores in Bailey optimisation and almost 40 % in Standard and Koopman optimisations are implemented in one state. Shannon has around 75 % of cores implemented in one state; with a very small percentage of circa 5 % Standard optimisation implement the cores with two states, while the percentage increases to about 10 % for Koopman, Bailey, and Shannon. Around 20 % of cores incorporate three states for Standard, Koopman and Bailey benchmarks, while Shannon implements less than 5 % of cores in three states. In contrast, around 10 % implement cores with four and five states across the four optimisations. Furthermore, Bailey has no cores that implement the function with six or seven states. However, in Standard, Koopman and Shannon around 5 % of cores implement the data using 6 states. In Standard optimisation



a) _Binary-trees States b) Funnkuch States

c) n-body States d) Spectral States

FIGURE 4.8: shows the number of states for the smaller benchmarks. The four benchmarks appear to have similar characteristic.

slightly more than 10 % of cores synthesise the function using seven states. Finally, in Standard, Koopman, Bailey and Shannon about 5 % implement cores using >eight states. In more detail all these data describe that scheduling technique minimize the number of state by increasing the number of parallelism. In other words it increases the number of operations per state, per clock cycle. Therefore it optimizes the hardware using machine code optimized.

4.3.2 Number of states per core (Larger benchmarks, with optimization)

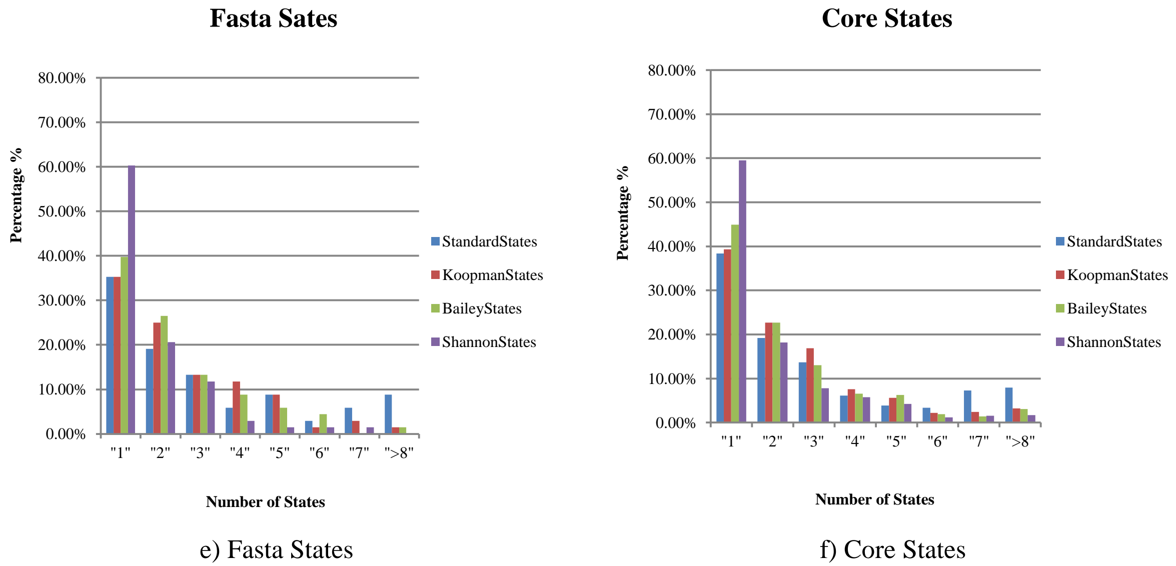
The Fasta benchmark Figure 4.9 e) presents the following data : almost 35 % of cores in Standard and Koopman have one state, while slightly more in Bailey nearly 40 % and Shannon approximately 60 % have one state. Again, around 25 % of cores for Koopman and Bailey have two states, and in comparison about 20 % of cores in Standard and Shannon optimisation have two states. Around 15 % of cores have three states for the four optimisations. However, around 5 % of cores in Standard optimisation and around 10 % in Koopman and Bailey optimisation implement the function using also four and five states. However, Shannon implements slightly less than 5 % of cores in four, five, six and seven states. In addition, circa 5 % in Standard, Koopman and Bailey optimisations implement cores using six and seven states.

The scenario in Core benchmark Figure 4.9 f) is slightly different than Binary benchmark. Almost 45 % of cores have one state in Bailey optimisation, while around 40 % of cores in Standard and Koopman have one state, and about 60 % of cores in Shannon have one state. Around 20 % of cores in the four optimisations have two states. Standard, Koopman and Bailey have the same percentage closely to 13 % with three states, while Shannon has around 5 % of cores with three states. Again, with the same percentage around 5 % the four optimisations have four, five and six states. Finally, almost 10 % of cores in Standard optimisation have seven and >eight, in comparison with nearly 3 % Koopman, Bailey and Shannon cores with seven and >eight states.

The Scimark benchmark Figure 4.9 g) implements the cores for the three optimisations in between one and >eight states. The most populated cores around 60 % in Shannon, about 45 % in Bailey, approximately 40 % in Standard and Koopman implement the function using one state. In addition almost 20 % of cores in Koopman and Bailey implement the cores with two states while in Standard and Shannon optimisation the percentage is slightly lower at circa 15 %. Furthermore, slightly more than 10 % in Standard and Bailey optimisations implement cores using three states, while in contrast Koopman with a percentage of about 20 % implements the function using three states and more that 10 % of cores in Shannon implement the function in three states. A very small percentage less than 10 % of cores synthesise the function using four, five and six states. Because of this 10 % of cores in Standard optimisation implement the data using seven and >eight states while a very small percentage around 2 % in Koopman and Bailey implement the functions using seven states. Finally, circa 5 % of cores in Koopman, Bailey and Shannon execute the function with >eight states.

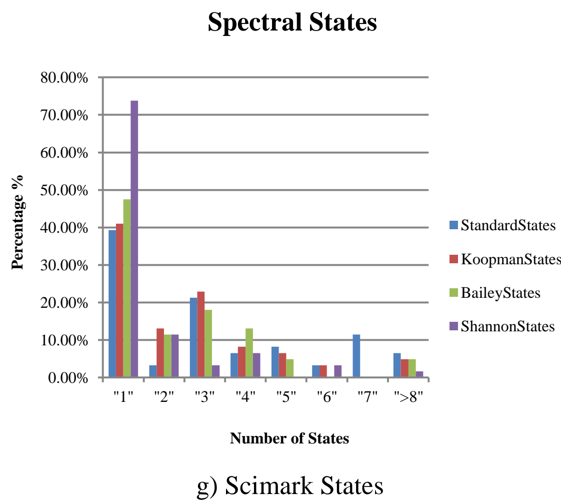
4.3.3 Number of states per core (All benchmarks, with optimization)

Figure 4.10 shows that the distribution of cores versus states tend to skew towards fewer states when the optimisation techniques are applied. This is because increasing the number of IPC reduces the number of memory fetch and store and so the states becomes less.



e) Fasta States

f) Core States



g) Scimark States

FIGURE 4.9: shows the number of states for the smaller benchmarks. The Three benchmarks appear to have similar characteristic. The majority perform the function in just few states.

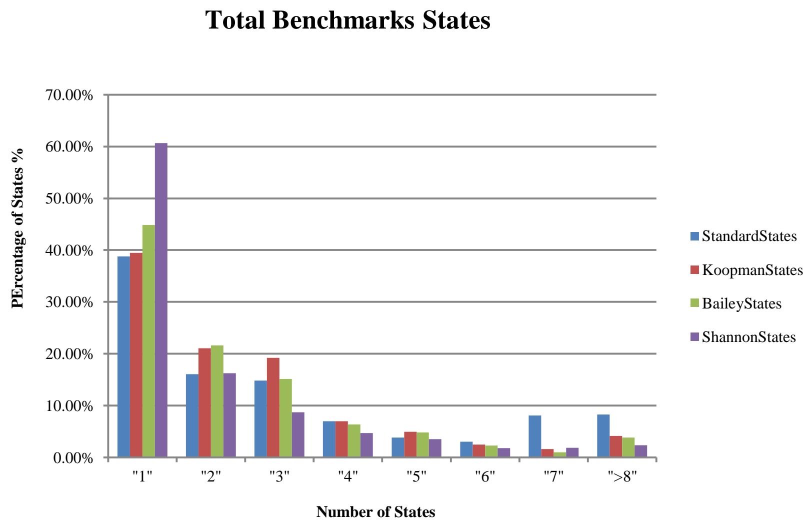


FIGURE 4.10: Number of states for the seven benchmarks. It shows that the overall benchmarks execute the function in few states

As a result of the seven benchmarks; The spread among the cores is between one to >eight with a decreasing value. Bailey has around 45 % of cores implementing the function using just one state, in comparison to Standard and Koopman with almost 40 % and Shannon with nearly 60 % of cores implemented in one state. This allows us to predict that Shannon will implement a higher number of IPC compared to the other three optimisations.

Also, in Koopman and Bailey around 20 % of cores implement the function using two states while in Standard and Shannon almost 15 % do so. Again, almost the same percentage of cores are implemented using three states across the optimisations, with Shannon being the only one to decrease the percentage to 5 %. At this point the percentage drops down to circa 5 % of cores that execute the data with four, five and six states. Finally, almost 10 % of cores in Standard optimisation, and around 5 % in Koopman, Bailey and Shannon implement the cores using seven and >eight states.

In the final analysis, a good fraction of power could be safe by reducing the number of states of the state machine [29] [49]. In other words, the state machine stores the present state and the next state into the register state [57]. When a state machine goes across the different states to perform a function, a percentage of power is consumed by the switching of the state register and number of clock cycles [29] [49]. At this the aim is to understand how minimising the number of states and increasing the number of instructions per state affects the architecture's behaviour in respect of power consumption area and timing.

4.4 Comparison of Optimised Cores' IPC

One of the key methods for ameliorating power consumption adopted in this thesis is applying static parallelism. A technique used to reduce power consumption in FSM architecture is reducing the number of states [29] [49].

Chapter 3 introduced a technique to translate software into hardware and so generate a FSM starting from machine code. To optimize the machine code scheduling techniques are used. Scheduling techniques increase the amount of static parallelism [27].

Increasing the amount of parallelism in the machine code reduces the number of states of FSM, because each state performs more operations and reduces the memory occupancy, therefore it needs fewer states to perform a function. Figure 4.11 and Figure 4.11 show the different parallelism in each benchmark.

4.4.1 IPC per core (Smaller benchmarks, with optimization)

The Binary-trees Figure 4.11 a) shows a very similar trend for Standard, Koopman and Bailey optimised cores. However, a slight increase of parallelism is noticed, with Bailey. In more detail, from core in the region of 13 to 31 Bailey slightly increases the amount of parallelism. The maximum number of parallelism in Standard is 6 while for Bailey and Koopman they are 7. On the other hand, Shannon increases the IPC. Up to to eleven instructions are implemented per clock cycle. It shows how in different points of the trend Shannon doubles the IPC compared to the other three optimisations. An example of this is from point 37 to 41.

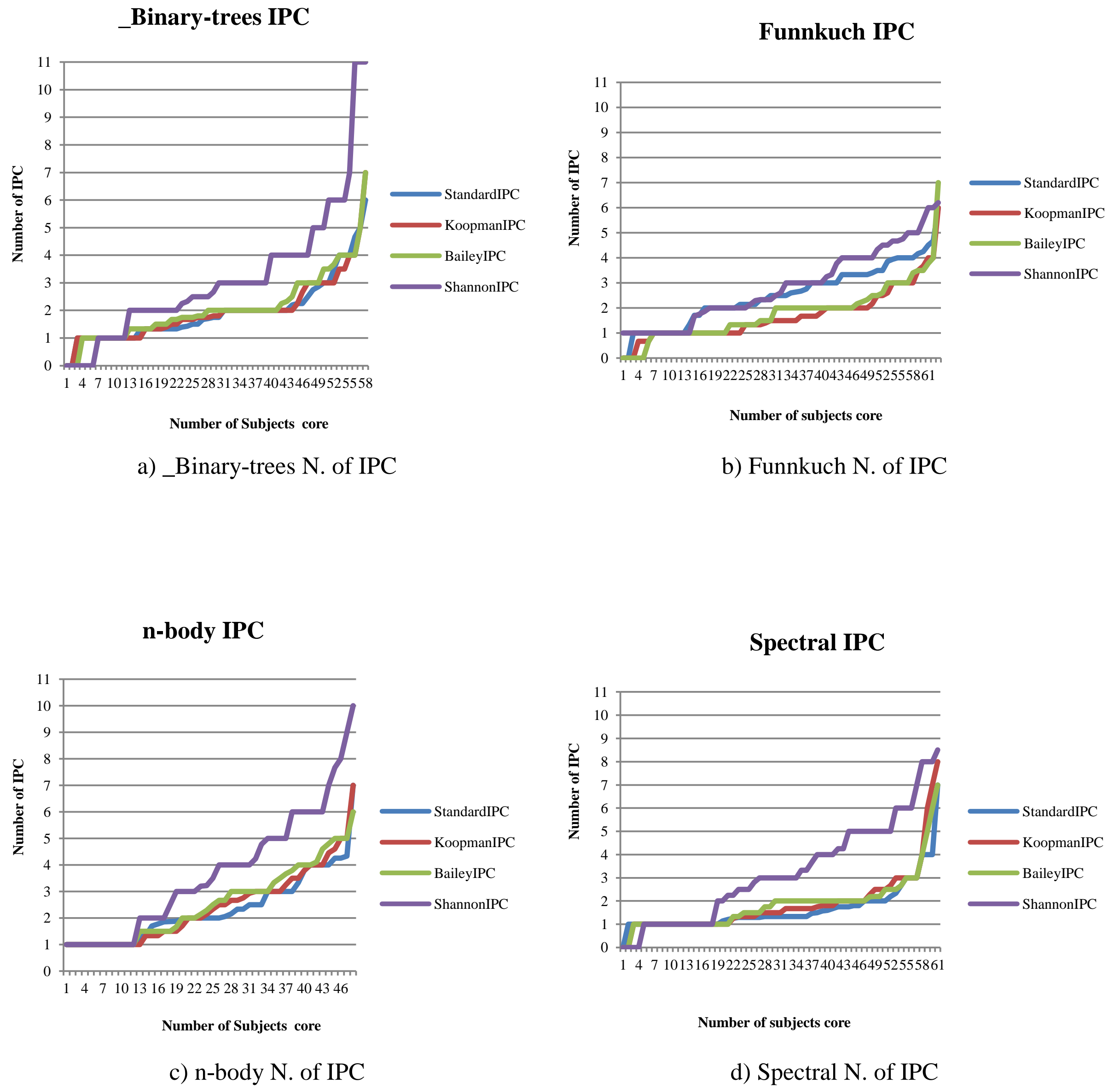
According to the Funnkuch benchmark Figure 4.11 b) Standard and Shannon optimisations extract more IPC. For instance, Standard and Shannon optimisations from core 14 to 58 double the number of parallelisms, followed by Bailey, and with Koopman coming last. However, Bailey extracts up to seven instructions per clock cycle in a core while the maximum instructions per clock cycle drops to six for Koopman and Standard.

The n-body benchmark Figure 4.11 c) has a few cores for which the Standard processes more instructions per clock cycle, such as cores 13 to 20. However, from around 20 to 47 Bailey increases the number of instructions per clock cycle. Finally Koopman and Standard extract the highest amount of parallelism with seven instructions per clock cycle. In addition, Shannon increases the number of instructions per clock cycle, reaching up to 10 Instruction per clock cycle.

The Spectral benchmark Figure 4.11 d) has a few cores where Standard optimisation implements more instructions per clock cycle, from around 19 to around 2. In contrast Bailey increases the IPC from circa 28 to circa 43. Finally, Koopman in the last cores seems to increase the parallelism, for instance the cores from around 46 to the end. it exploits the maximum amount of parallelism as 8 IPC. In addition Shannon exploits more parallelism across the entire trend and at some points the number of IPC is doubled as from core 45 to 53

4.4.2 IPC per core (Larger benchmarks, with optimization)

The Fasta benchmark Figure 4.12 e) presents its data thus: three optimisations have parallelism going from zero to seven, while Shannon gets up to 9. In more detail, Bailey seems to have more parallelism compared to Standard and Koopman, for instance from core around 13 to 29 Bailey has a higher number of parallelism followed by Koopman and finally by Standard. In addition, from core around 40 to the end Bailey seems to slightly increase the number of parallelism also followed by Koopman and at the end there with lower instructions per clock cycle there is Standard optimisation. However, Shannon has an increase of around one point.



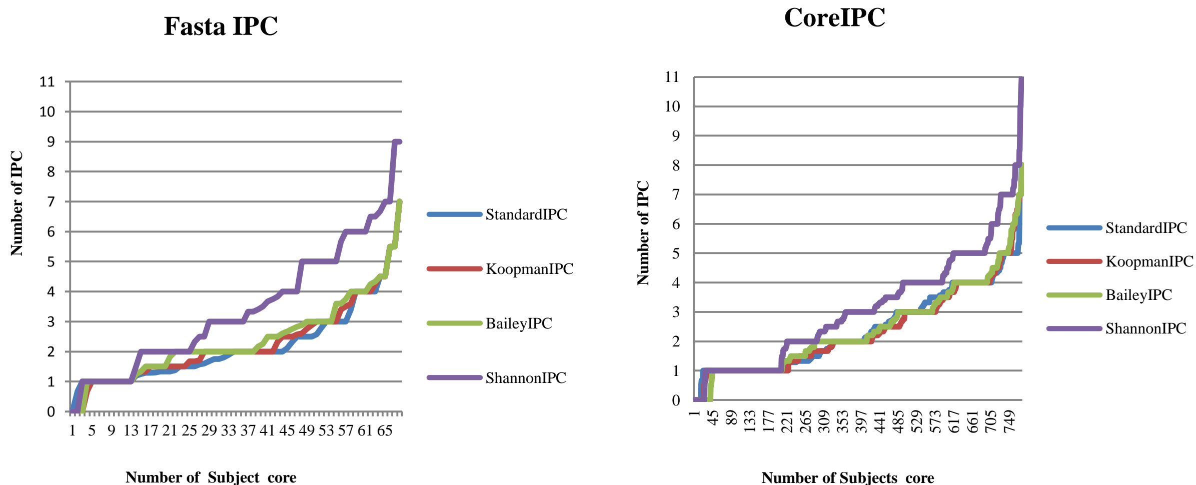
a) _Binary-trees N. of IPC

b) Funnkuch N. of IPC

c) n-body N. of IPC

d) Spectral N. of IPC

FIGURE 4.11: Number of IPC for the smaller benchmarks.Shannon increases the number of IPC



e) Fasta N. of IPC

f) Core N. of IPC

Scimark IPC

Number of IPC

Number of subjects core

StandardIPC

KoopmanIPC

BaileyIPC

ShannonIPC

g) Scimark N. f IPC

FIGURE 4.12: Number of IPC for the Bigger benchmarks. Shannon increases the number of IPC

The distribution of parallelism inside the Core benchmark Figure 4.12 f) indicates that the three optimisations exploit the same amount of parallelism in Standard, Koopman and Bailey. However, in more detail, Bailey at some point and for a few cores seems to slightly increases it, such as from almost 295 to almost 310. In contrast Shannon improves the IPC. From core around 313 to around 677 it adds one point compared to Standard, Koopman and Shannon, and to conclude the last few cores Shannon implements eleven IPC compared with Bailey's that has just seven.

The Scimark benchmarks Figure 4.12 g) has an equal overage in implementing instructions per clock cycle. In more detail, literally just a few cores (around 160 to 165) in Standard optimisation have slightly higher IPC. In contrast, from circa 180 to 330 Bailey has the highest IPC. Again, from around 430 to 511, Standard has a higher number of parallelism. Finally, Bailey and Koopman exploit for really one or two cores up to 8 instructions per clock cycle. On the other hand, from core around 171 to the end Shannon almost doubles the number of IPC.

4.4.3 IPC per core (All benchmarks, with optimization)

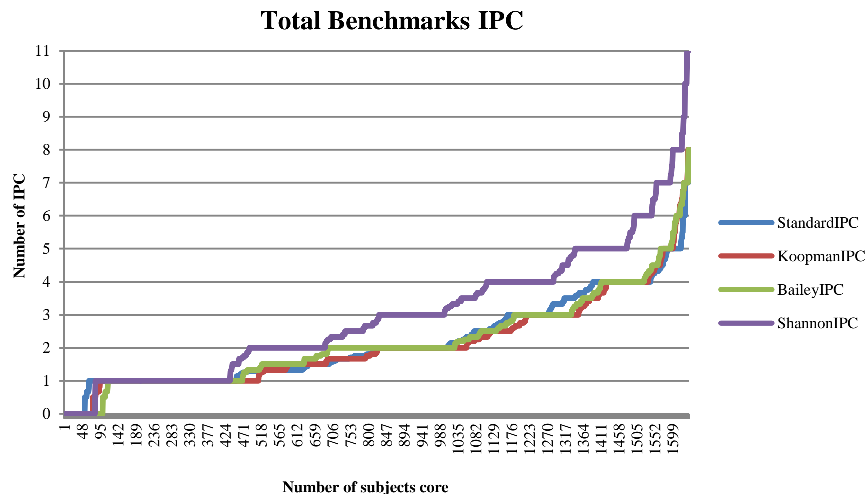


FIGURE 4.13: N. of IPC for the seven benchmarks. The Shannon optimization increases the number of IPC up to 11 instructions per clock cycle.

Figure 4.13 shows the total parallelism in the seven benchmarks. The average of instructions per clock cycle for Standard, Koopman and Bailey optimisations appears to be pretty similar. However, in more detail, Bailey seems to slightly increase the IPC from core around 633 to around 789. In contrast, Standard seems to exploit more parallelism from core in the region from 1265 to 1400. Finally, for the last cores it appears that the number of instructions per clock cycle tend to be, from the largest to smallest: Bailey, Koopman, and Standard. To conclude, Shannon increases the number of parallelism, and further, it is the optimisation to which this Chapter dedicates more attention to understand how the hardware behaves in power, timing and area.

4.5 Comparing the Area of the optimised benchmarks in Composite and Wave-core architecture

This thesis aims to improve the power density of accelerators to alleviate Dark Silicon and so try to squeeze every nW inside the logic design.

Particularly in embedded systems, the power density, which is related to the amount of area per power consumption, has become essential to design devices with a good performance [40]. This section analyses the amount of area that every optimisation generates for the seven benchmarks.This should help to understand how the area changes for each optimisation in both architectures, Composite and Wave-core. In more details it compares; Koopman, Bailey and Shannon optimisation for the benchmarks set combined, compared to the unoptimized Baseline in both Composite and Wave-core architectures.

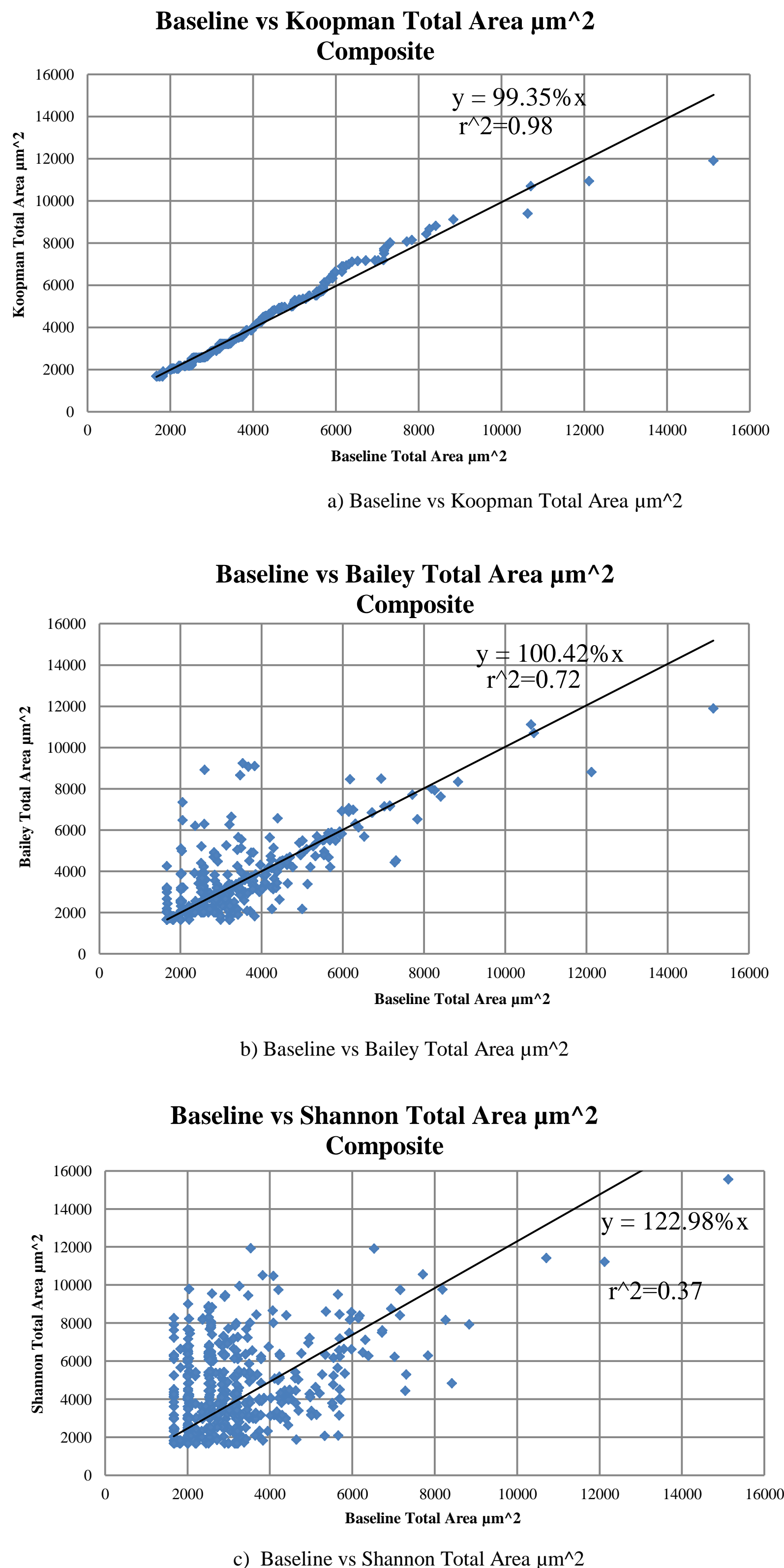
Chapter 2 described this methodology as a valuable option to enhance efficiency of cores and, by combining them, to optimise the power budget of the chip. Chapter 3 analysed the time area and power using Whisker and box plot to understand who between Composite and Wave-core does better and then by Scatter graph to observe how individual cores behave.

Here, to understand how optimization and architectures behave. It starts to analyse at firs the Composite architecture using Scatter graph. Then the same approach will be for the Wave-core. And finally Composite vs Wave-core are compared. Furthermore, to observe the strength of the data the Pearson correlation coefficient is performed. The Pearson correlation measure the linear correlation between two variables plotted on the x,y axis. In this case the Pearson correlation is used to understand if the set of data produce a similar output

4.5.1 Comparing the Area of the optimised benchmarks in Composite architecture

Figure 4.14 a) shows the comparison between Baseline and Koopman in Composite architecture. Koopman has a minimal effect on core area. This could be explained as being due to Koopman having a lower number of IPC per core compared to others. For example, Bailey optimisation compared with Baseline Figure 4.14 b)has a better impact, the cores start be grouped and the distance from the fit line increases. This is because Bailey increases the number of IPC. To prove this, Shannon Figure 4.14 c), which further increases IPC, appears to have a significant impact, the cores are grouped and the distance from the fit line is higher; furthermore the area increases as average by up to 23 %.

According to the data, scheduling optimisation has a significant impact on core generation outcomes and should be considered carefully in such work. Another observation is that; Koopman which optimizes within a block appears to be a linear function while Bailey that optimize across blocks lost the linearity. On the overage it slightly improves the area but on single core because the cores become more complex. A few of them increase the area. That is because as already mentioned, increasing the number of IPC the workload shift between a core to another core. This imply that while a few cores do almost of the job the others act just as a transient. Shannon confirms this theory because using a globally optimization increases the number of IPC and so increases the complexity of the core. For this reason the cores are not well distributed along the fit line. Furthermore, Shannon that has the higher number of IPC has a higher number of core in the region of 12000umˆ 2.



a) Baseline vs Koopman Total Area µm^2

b) Baseline vs Bailey Total Area µm^2

c) Baseline vs Shannon Total Area µm^2

FIGURE 4.14: Composite Total Area (um ˆ 2) comparison between Baseline and optimized benchmarks. Koopman has a similar shape it is because optimizer within a block and the IPC is very love. In comparison Bailey which optimize across blocks has a slightly improving. Finally Koopman with the higher number of IPC improves the area up to 23 %

4.5.2 Comparing the Area of the optimised benchmarks in Wave-core architecture

Wave-core architecture implements each computational structure as a separate circuit to achieve a better power density and at the same time reduces the risk of generating hotspots. Figure 4.15 a to c) shows the comparison of Koopman, Bailey and Shannon optimisation for the benchmarks set combined, compared to unoptimized Baseline in Wave-core architecture

At first, the three optimisations are compared and then to better understand how the cores could be used to improve the overall power consumption of the device in more details are investigated.

The three optimisations decrease the area of logic while increasing the number of IPC. For example, Koopman Figure 4.15 a) decreases the area on average by around 29 %, Bailey just a bit more (around 31 %, Figure 4.15 b) and finally Shannon has the higher number of IPC around 32 % Figure 4.15 c). It is a very important result because it proves that the new way to implement the Wave-core architecture helps to optimize the area because the synthesis tool treats each process or dedicate state as a local process. As see for Composite, increasing IPC a few core do more while other do less. Furthermore, according with the results, because Wave-core has a dedicate logic design, improves the optimization of logics when the workload per state increases.

Figure 4.16 shows the Venn diagram for Composite and Wave-core to better understand the concept.

The Venn diagram shows how the optimisations treat Composite and Wave-core architecture. In Composite the synthesis treats the state machine as a global process and so optimises the architecture, while for Wave-core, it considers one process per time. The separating of processes per time increases the area of logic

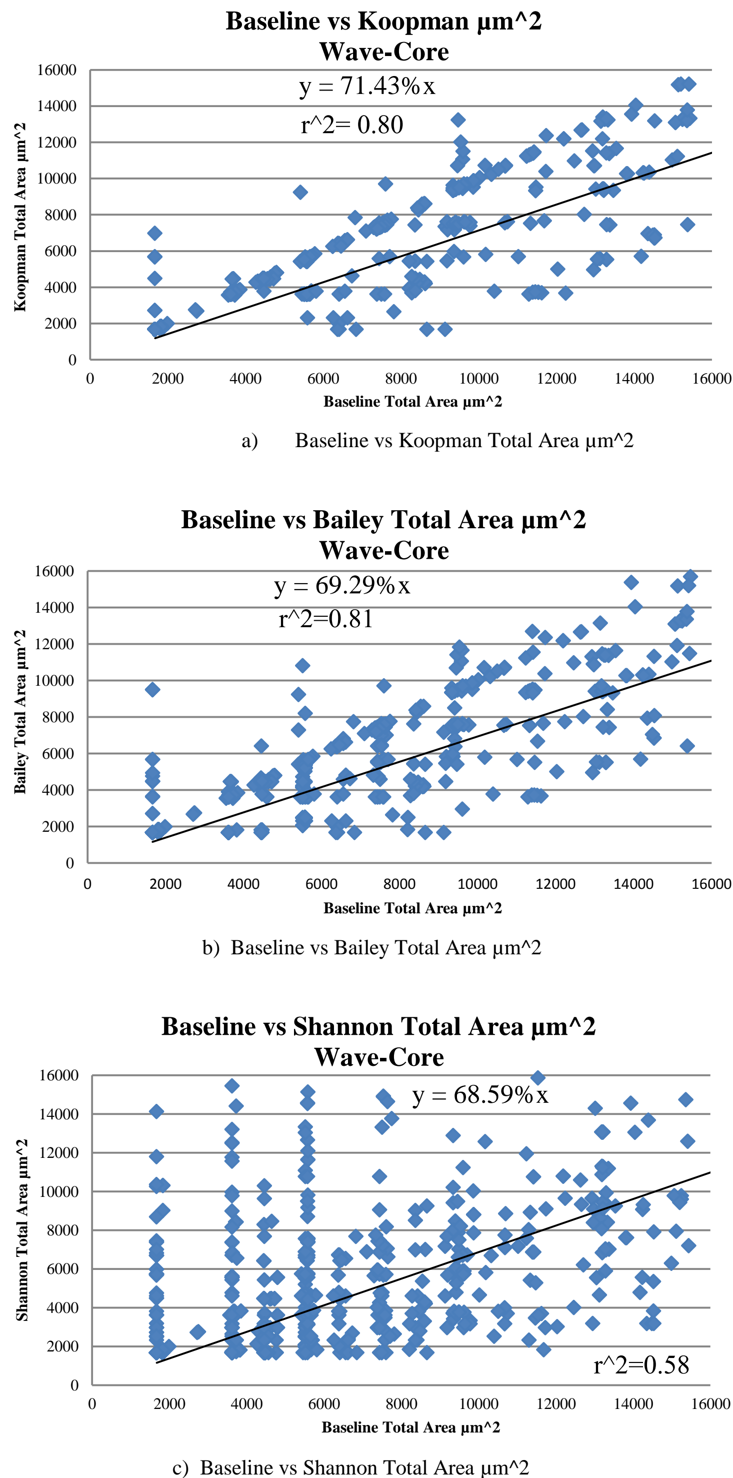
Considering again Baseline vs Shannon Figure 4.15 c). It shows that the spread is not uniform: it has larger and smaller cores in both architectures. The cores with a higher IPC are those with a larger area, up to around 8x. This happens because, to exploit more parallelism, each state needs more concurrent logic, therefore they could not be shared. In contrary with less parallelism could be grouped, for instance the data can share the logic as soon become available. Especially, as will be explained in a later section, cores with higher area are probably the ones designed to improve the overall power and in this case the power density.

Another means of consideration is dividing the Shannon cores above and below the trend-line. The cores above the line are larger, while those below are smaller. This could be the first characteristic to take note of in selecting the cores.

4.5.3 Comparing Composite vs Wave-core architecture total Area for the optimized benchmarks

To understand how the total area behave in Composite and Wave-core architectures for the optimised benchmarks, the Shannon optimisation is considered, which is the one with the highest IPC and therefore the one that can give a better illustration. Figure 4.17 compares both core types (Composite and Wave-core) for Shannon optimisation.

Cores with a small area appear to be the same and not to much attention is paid to them. Alternatively, a more extensive consideration to cores with larger area is given. The Wave-core separates the logics and each state is treated as a separate system, therefore the increase of area means increase in number of logics. Indeed,



a) Baseline vs Koopman Total Area µm^2

b) Baseline vs Bailey Total Area µm^2

c) Baseline vs Shannon Total Area µm^2

FIGURE 4.15: Wave-core architecture comparisons total area (umˆ 2) between Baseline and optimised benchmarks. (Increasing the IPC in Wave-core architecture decreases the total area of the logic)

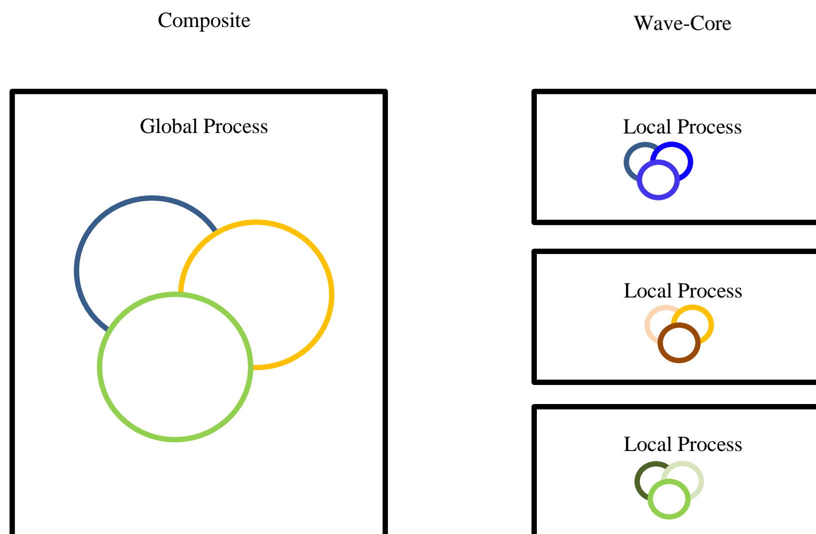


FIGURE 4.16: Venn diagram for Composite and Wave-core. Any circles show all possible logic relations between finite states inside the finite state machine.

the higher the IPC, the higher the number of logic units, and thus the area, which is up to 30x. For this reason the Wave-core can contribute more to reduce power density. Another way to look at the effect that the optimizations have on the cores is to observe the correlation coefficient. The correlation helps to understand if the data produced are different. But also the strength of the relationship between the baseline and the optimised benchmark. Figure 4.14 shows the three optimized benchmarks for Composite architecture. The first point to pay attention is that the three correlation have a positive slope. Then comparing the scatter data between the x and y axis, it shows that the scatter along the y axis increases with the increases of IPC. In fact Shannon that has the higher number of IPC has a higher number of core in the region of 12000umˆ 2. To prove that Table 4.1 calculates the Pearson correlation r. In Baseline vs Koopman, r=0.98, while in Baseline vs Bailey r= 0.84 and finally Baseline vs Shannon r= 0.60. To conclude, it shows how increasing of IPC increase the complexity of a few cores and as a result the coefficient becomes weaker. This is also another proof than increases the IPC the workload increases inside some cores.

On the other side, Wave-core architecture Figure 4.15 shows a positive coefficient as for the Composite. The most important point is that the scatter data increase the distance along the y axis when compared with the three optimized benchmarks and with the Composite, this is because as said before the Wave-core treat each process as a separate logic so the synthesis increases the number of logic per state inside the cores. Looking at the Pearson correlation r: Table 4.1 Baseline vs Koopman the r=0.89, and in Baseline vs Bailey r=0.9 Finally Shannon this the higher IPC r= 0.76.

To summarize in a Wave-core architecture increasing the IPC decreases the strength of the relationship. Returning back to the percentage of improvement, this technique helps to achieve a more efficient area if is applied to functions with higher IPC. Another advantage of Wave-core architecture is that implements the FSM using a dedicate process reduces the possibility of generating hotspot.

To conclude Table 4.1 shows the summary of Pearson correlation coefficient. The

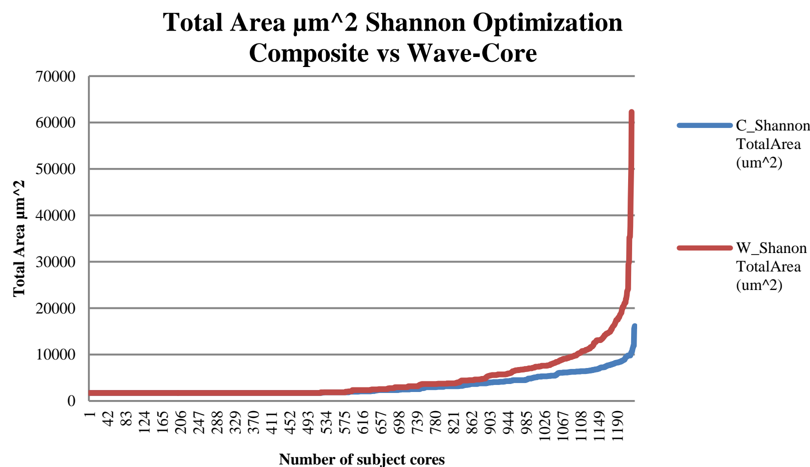


FIGURE 4.17: Total Area umˆ 2 Shannon optimisation Composite vs Wave-core (Wave-core increases the total area of logics

TABLE 4.1: Pearson correlation coefficient total area

Parameters	Composite Arch. (r)	Wave-core Arch. (r)
Baseline vs Koopman Total Area	0.98	0.89
Baseline vs Bailey Total Area	0.84	0.90
Baseline vs Shannon Total Area	0.62	0.76

data show that the strength is quite strong for both architectures in any optimized benchmarks. However, according to the data increasing the number of IPC the strength in Composite becomes weaker than the Wave-Core. Therefore it proves that the different optimized benchmarks produce different data

4.6 Comparing the Timing of the optimised benchmarks in Composite and Wave-core architecture

Chapter 2 introduced the CoDA architecture. This architecture when is on operation mode has the advantage to set up in standby mode almost of the hardware. Also (Chapter 1.1) introduced the Stack CoDA architecture. The data flow in stack code simplifies the identification of parallelism [36]. Applying the back track algorithm key technique of the translator tool, (see Chapter 2.5.1), the stack machine code could be directly translated into hardware by applying a generic state machine (see Chapter 2.5.1). Finally (Chapter 4.3) explained that reducing the number of state in FSM architecture ameliorates the timing and the power of the architecture. Chapter 4 introduced three different stack scheduling techniques. Stack scheduling reduces the number of time to fetch or store data in memory. Reducing the memory occupied reduces the number of states in the FSM (see Chapter 2.5 to understand how the FSM is implemented).

To observe this contention, the timing with regards the three optimisations, Koopman, Bailey and Shannon for the Composite and Wave-core architecture is investigated and analysed.

4.6.1 Comparing the Timing of the optimised benchmarks in Composite architecture

Baseline vs Koopman Figure 4.18 a) shows that Koopman has a very small significant improvement in timing – just 0.5 %, so no overall effect. However, the optimisation has an impact in terms of increasing and decreasing core timing.

For instance, the most populated cores are within the range 1100ps to 1200ps and that explains why both architectures are pretty similar. For both architectures, the minimum time in which a core executes the data is 995ps, while the maximum time is 1503ps for the Baseline and 1534ps for the Koopman. However it is more interested in a single core that has a better timing. An example of this is: the core executes in Baseline in 1420ps while Koopman executes the same core in 1200ps Figure 4.18 a), red spot). This means that Baseline has frequency 700MHz while Koopman increases the frequency to around 850MHz, which is around 18 % greater. Another possibility is to consider the Baseline and for example a core that executes in 1200ps, in Bailey executes in 1500ps (Figure 4.18 b), red spot). Therefore for this core Baseline architecture is perhaps more appropriate for high data computing architecture.

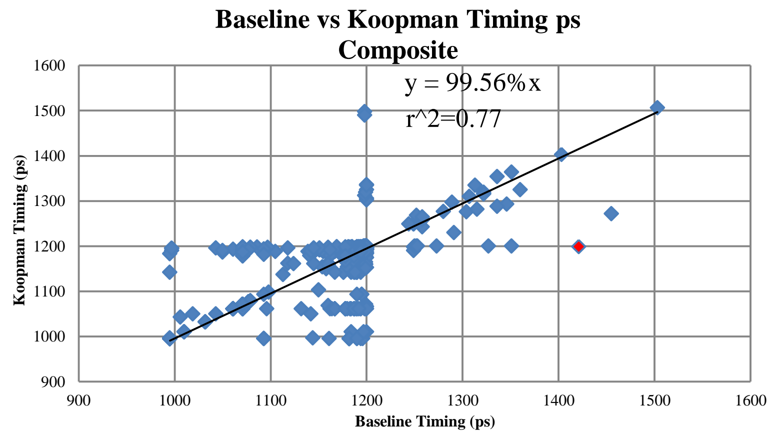
Baseline vs Bailey Figure 4.18 b) shows a scenario very similar to the previous one. Also, the faster core implements the function in 995ps in both architectures, while the slower is 1503ps for the Baseline and 1523ps for the Bailey. Again, the most populated cores are between 1000ps and 1200ps in Bailey, while the Baseline is spread between 1100ps and 1200ps. In view of this we may assume that Bailey can have a higher set or core to be considered.

The last graph to analyse is the Baseline vs Shannon Figure 4.18 c). As it has already showed, Shannon increases the IPC; as a confirmation of this, it improves the timing of slightly more than 1 % in a way to extend the number of cores that perform the function in 995ps. Further, while a few cores perform the data in 1200ps, the same cores in Baseline perform in 1300ps, 1400ps and 1500ps (Figure 4.18 c), red spot). That is to say, Baseline also has a few cores that execute the function in 1200ps compared with the circa 1500ps of Shannon, and this proves that each core could be considered alone to improve the overall performance of the architecture.

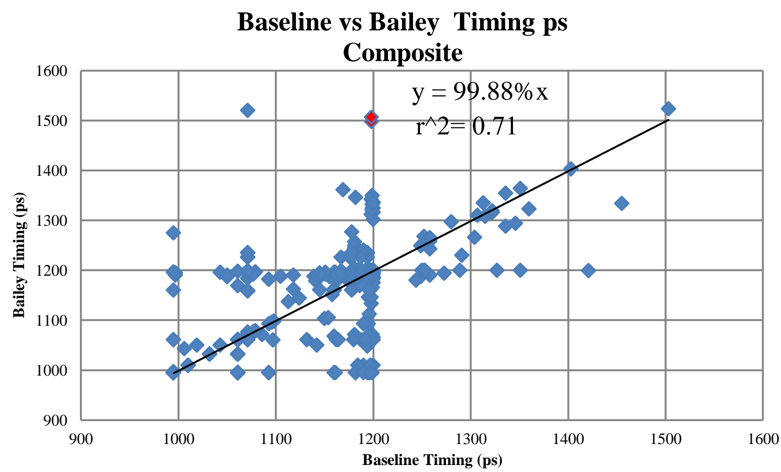
To conclude, at this point the methodology starts to become clear and the system to generate cores; then analysis gives the idea how the real application works. And more, the cores in Composite architecture for the three optimisations run from around 1GHz to 500MHz, while in a real application cores with frequencies from 800MHz to 1GHz are more applicable. In specific cases design engineers could also be interested in cores at 500MHz. An example would be a clock divide creating two clock domains, which clock two groups of cores at 1GHz and 500MHz. This simple technique could give an extra benefit for the total power of the device by grouping cores into different clock regions [23].

4.6.2 Comparing the Timing of the optimised benchmarks in Wave-core architecture

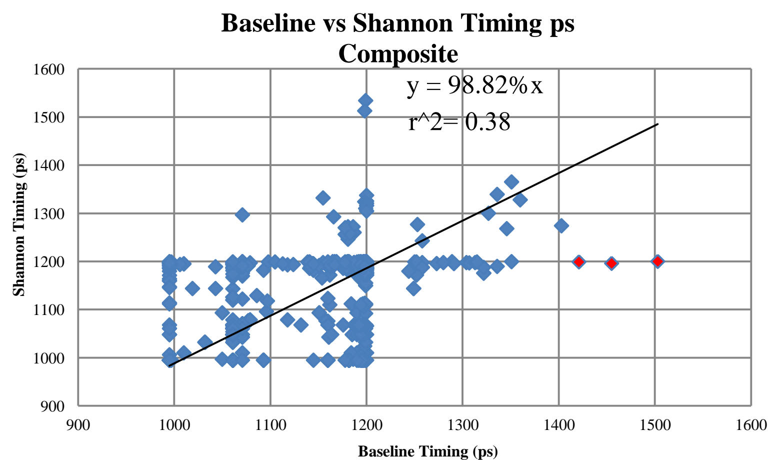
A novel contribution of this project is Wave-core architecture; therefore the timing for the three optimization is investigated.



a) Baseline vs Koopman Timing ps

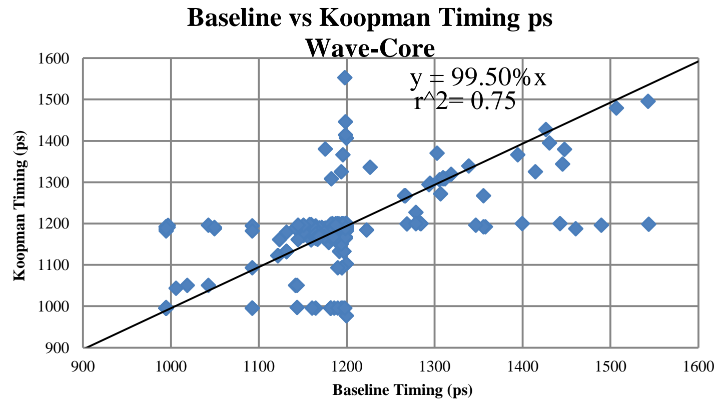


b) Baseline vs Bailey Timing ps

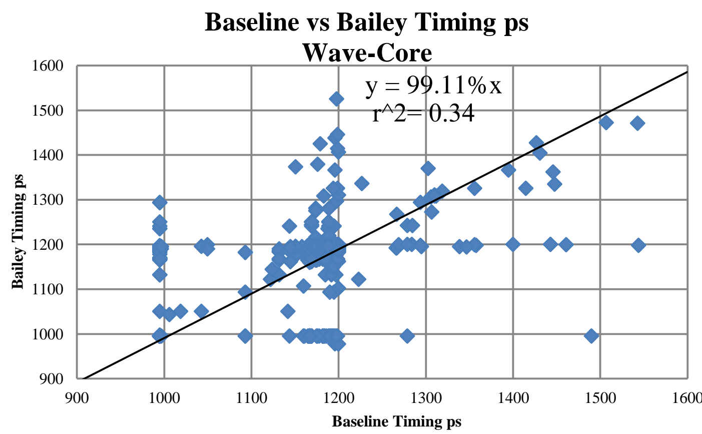


c) Baseline vs Shannon Timing ps

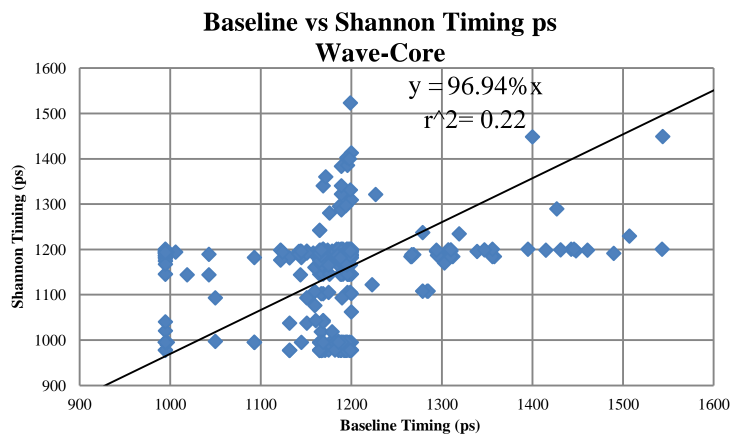
FIGURE 4.18: Compares Composite timing architecture between Baseline and optimized benchmarks.



a) Baseline vs Koopman Timing ps



b) Baseline vs Bailey Timing ps



c) Baseline vs Shannon Timing ps

FIGURE 4.19: Compares Wave-core timing architecture between Baseline and optimized benchmarks.

The timing for the three optimisations in Wave-core architecture is shown in Figure 4.19

The Wave-core optimisations appears to have much similarity with the Composite. For example, they are in the same range of timing. However, the results prove that Wave-core improves the timing when it increases the IPC. For instance, Shannon optimisation reduces the overall timing by around 4 % Figure 4.19 c). In comparison, the overall Shannon optimisation in Composite is about 1 %. This because as already explained in the (Section 4.5 ,total area). The Wave-core and its dedicated states has a better optimization of logic. Therefore, the timing improves

This observation demonstrates another advantage of Wave-core architecture, and as the architecture is further investigated, it reveals itself to be a better option in terms of the whole design.

4.6.3 Comparing Composite vs Wave-core Timing of the optimised benchmarks

To fully understand and compare Composite vs Wave-core architecture , Figure 4.20 compares the cumulative distribution in Composite and Wave-core for Shannon optimisation. It uses Shannon because has the higher number of IPC and so is easier to observe the differences

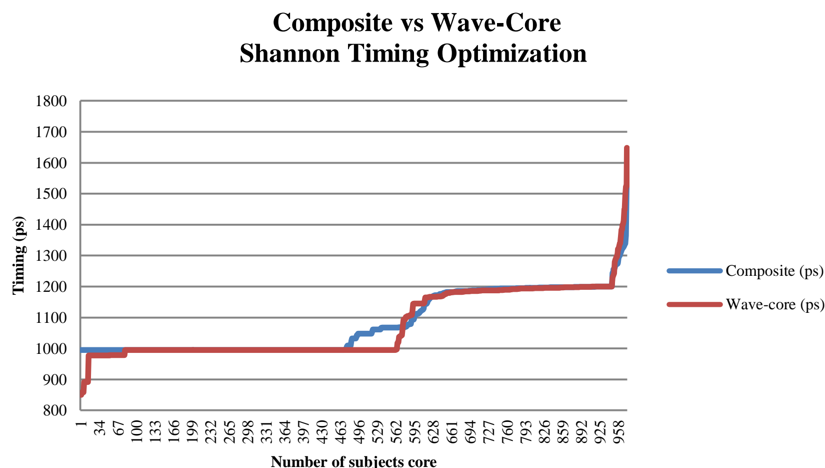


FIGURE 4.20: Compares Composite vs Wave-core for the Shannon timing (ps) optimisation (The Wave-core architecture improves the timing)

Figure 4.20 compares the cumulative distribution in Composite and Wave-core for Shannon optimisation. Wave-core architecture for overall cores improves the timing; in short, the minimum timing is less than 900ps, where in comparison that for Composite is around 1000ps. Furthermore, Wave-core has a higher number of cores that perform the timing in about 1000ps. As a result, this section has established the suitability of Wave-core for increasing IPC but also illustrated another advantage of Wave-core architecture. An optimal choice could be to have a combination of Composite and Wave-core architecture. In this way a CPU architect can always consider cores that do better in one or other architecture.

TABLE 4.2: Pearson correlation coefficient timing

Parameters	Composite Arch. (r)	Wave-core Arch. (r)
Baseline vs Koopman Timing	0.87	0.86
Baseline vs Bailey Timing	0.84	0.58
Baseline vs Shannon Timing	0.61	0.46

Another way to observe the data and compare both Composite and Wave-core architecture is to analyse the Pearson correlation coefficient. This help to understand if the optimized benchmarks produce a different result. Comparing Figure 4.18 and Figure 4.19 they show the correlation coefficient is lower in the Wave-core. It means that the Wave-core has some cores place behind the fit line that run faster than the Composite. Table 4.3 shows the comparison of the Pearson coefficient correlation. Another point to observe is that in Composite architecture the correlation stays medium-strong, but on the other side the Wave-core tends to become weak when IPC increases. In fact Shannon stays at 0.46 which is considered very low-medium correlation. Therefore as already mentioned different optimizations produce a different set of data.

4.7 Comparing the Static Power of the optimised benchmarks in Composite and Wave-core architecture

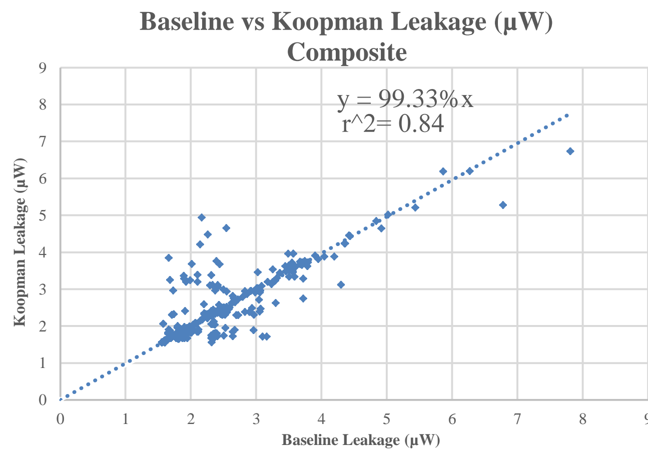
One of the main factors in power consumption is the leakage, therefore at this stage the three optimisations behave in terms of static power are investigated. To understand better the differences, all seven set of optimized benchmark are combined.

4.7.1 Comparing the Leakage of the optimised benchmarks in Composite architecture

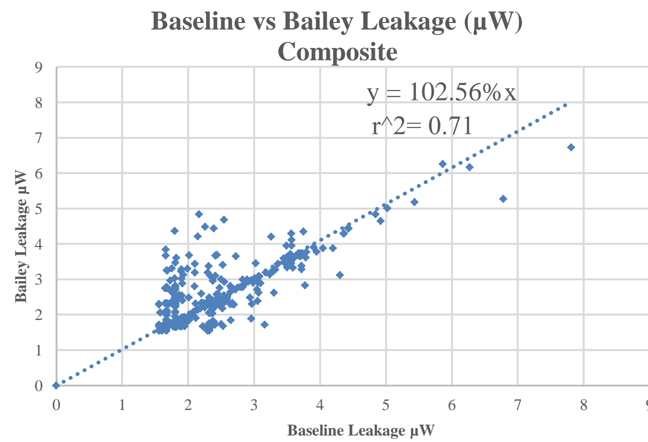
Baseline vs Koopman Figure 4.21 a) shows; Koopman slightly decreases the average of leakage power, at less than 1 %. In comparison most of the cores are centred between 1.5uW and 3uW for Koopman and between 1.5uW and 4uW for Baseline. Both architectures have the core with the minimum leakage at 1.5uW while the maximum is around 7.8uW for Baseline and approximately 6.7uW for Koopman. A further investigation demonstrates that both optimization have cores that consume less energy when compared between each others. Therefore to have an efficient CPU, design engineers must consider each case and compare both architectures.

Baseline vs Bailey Figure 4.21 b) shows that Bailey increases the leakage by about 2 %. Most cores are situated between around 2.5uW and 4uW for both architectures. For both Baseline and Bailey the core that wastes least energy is at 1.5uW; by contrast, the core that consumes most energy in Baseline consumes 7.8uW while in Bailey consumes 6.7uW. Despite this, both benchmarks have a few cores that decrease double the leakage when compared.

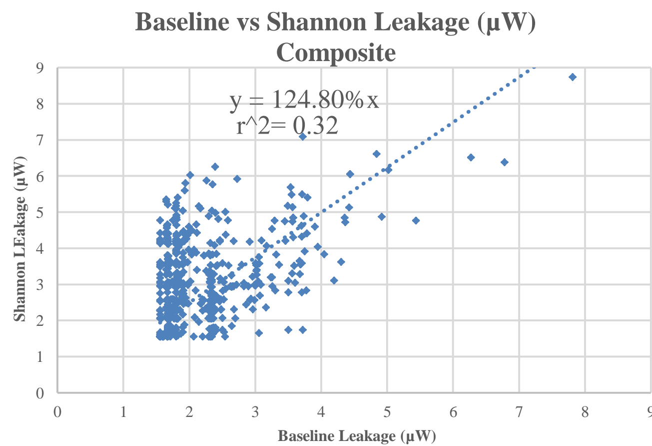
Shannon optimisation Figure 4.21 c) increases the area per logic and as a consequence it increases the leakage. In fact, comparing the Baseline vs Shannon, it increases in static power of 25 %. For both architectures the core that wastes least



a) Baseline vs Koopman Leakage (µW)



b) Baseline vs Bailey Leakage (µW)



c) Baseline vs Shannon Leakage (µW)

FIGURE 4.21: Leakage (uW) for the four optimized benchmarks in Composite. Notice as soon the optimization increases the number of IPC. The leakage becomes less linear; it is because few cores carry more data.

energy is at 1.5uW. Baseline's most energy-wasteful core is at 7.8uW while Shannon's is at 9uW. The cores are centred between 1.5uW and 6uW for Shannon and 1.5uW to 2.5uW for Baseline.

In view of this, it has proved that by increasing the IPC it increases the area of logic and so the static power.

4.7.2 Comparing the Leakage of the optimised benchmarks in Wave-core architecture

So far has explained that the Wave-core architecture increases the area per logic unit, so the static power. Wave-core has also demonstrated that by increasing IPC tends to decrease the area of macro-cells, therefore it improves the static power in that respect. Figure 4.21a to c shows the leakage for the three optimisations. In fact Shannon optimization that increases the number of IPC, improves the leakage more than 30 %. Furthermore Figure 4.22 could be summarized as:

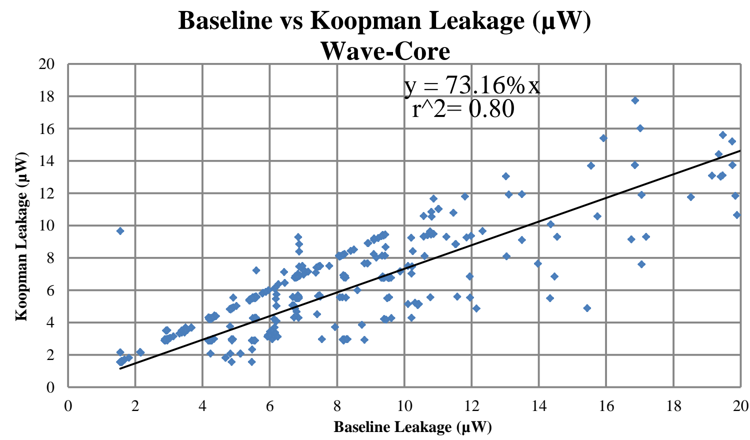
- IPC decreasing the leakage current: looking at the three optimizations, becomes clear that increasing the IPC decreases leakage because during the optimization many cores carry less data while other in small percentage carry more data. Another observation is that increasing the IPC, it loses the linearity because the optimization tries to make a small percentage of cores carry more data than others. In fact Chapter 4.5 showed this evidence in area. While a small percentage of core increase the area when compared with the baseline the majority decreases. To conclude the Shannon Figure 4.22 c) which has the most IPC decreases the leakage on the entire benchmark by more than 30 %., and it is in line with the increasing of area Figure 4.15 c)
- Many cores are situated within 6uW, which also appears in Composite static power. See Figure 4.21
- To finish, as it will show, Wave-core architecture is built with the intention to use a power gating technique, which helps to reduce the leakage (see further section)

Another way to observe the data is to analyse the Pearson correlation coefficient. Table 4.3 shows the r value for both architectures. In Composite architecture Baseline vs Koopman r= 0.91, in Baseline vs Bailey r = 0.84 and finally in Baseline vs Shannon r = 0.56

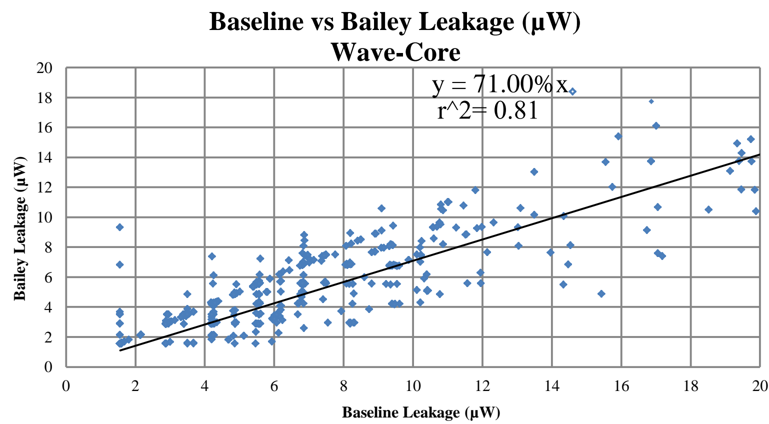
On the other side, in Wave-core architecture the Pearson correlation coefficient is for Baseline vs Bailey r = 0.89, in Baseline vs Bailey r= 0.9 and finally in Baseline vs Shannon r =0.74. Notably how the Pearson leakage correlation is similar to the Pearson area leakage (see Table 4.1). That happen because the two factor as already mentioned are linked. In addition increasing the IPC the Composite tends to lost the strength of correlation when compared with the Wave-core.This suggest that the Wave-core when the IPC increases has a better optimization of logics

In the next sections the analytic model is used to calculate the amount of leakage reduced by applying power gating in Wave-core architecture. That is to say; the power gating could be applied just to the Wave-core architecture, because the logic per state can be isolated with out lost the operations.

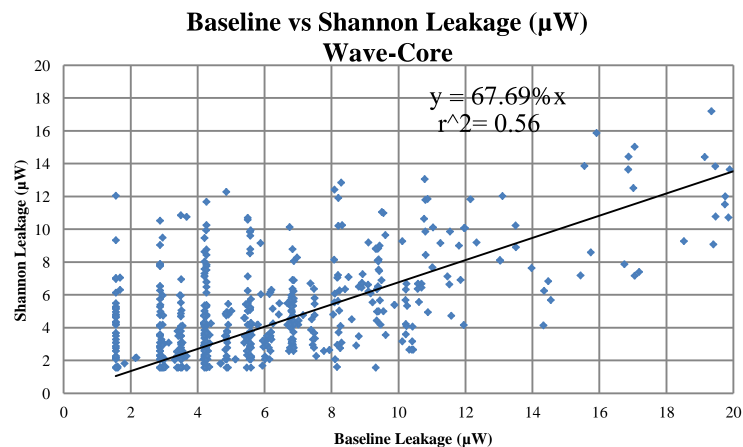
To conclude, again by applying optimisation, Wave-core has given another sign that could be a better candidate compared to Composite.



a) Leakage (µW) Baseline vs Koopman



b) Leakage (µW) Baseline vs Bailey



c) Leakage (µW) Baseline vs Shannon

FIGURE 4.22: Leakage (uW) for the three optimised benchmarks in Wave-core. The optimized benchmarks decrease the leakage.

TABLE 4.3: Pearson correlation coefficient Static Power

Parameters	Composite Arch. (r)	Wave-core Arch. (r)
Baseline vs Koopman S. Power	0.91	0.89
Baseline vs Bailey S. Power	0.84	0.90
Baseline vs Shannon S. Power	0.56	0.74

4.8 Comparing the Dynamic Power of the optimised benchmarks in Composite and Wave-core architectures

So far the number of optimised hardware cores have been introduced, furthermore the increasing of IPC for each optimisation has been observed. At this point to understand how the dynamic power behaves for Composite and Wave-core architectures. At first the Composite architecture is considered and then the Wave-core, after that both Composite and Wave-core are compared. Finally this Section analyses how the number of states influence the power consumption.

4.8.1 Comparing the Dynamic power of the optimised benchmarks in Composite architecture

Baseline vs Koopman Figure 4.23 a) shows that both architectures consume the same amount of dynamic power. Most cores are grouped between 4mW/100MHz and 5mW/100MHz However, Baseline has a few cores between 4mW/100MHz and 4.5mW/100MHz while Koopman increases the power up to 6mW/100MHz. Therefore this situation suggests that in a real situation the Baseline optimization is considered.

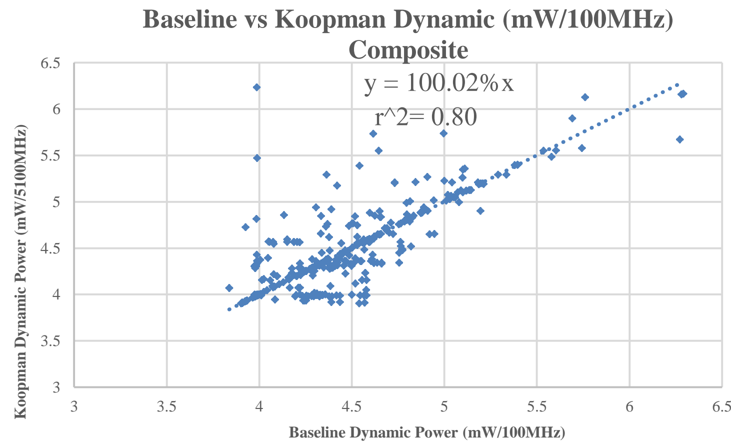
Baseline vs Bailey Figure 4.23 b) shows a similar situation as Baseline vs Koopman. In fact, both optimization have the majority of cores situated between around 4mW/ 100MHz and 5mW/ 100MHz. However, Baseline appear to do slightly better than Bailey.

Baseline vs Shannon Figure 4.23 c) shows that Shannon increases the dynamic power an average of around 4 %, and the most populated cores are in the range of 4mW/100MHz to 6mW/100MHz, while the Baseline has the most populated cores between around 4mW/ 100MHz and 4.5mW/100MHz. As with Baseline vs Bailey, Shannon vs Baseline has a few cores that gain 1mW compared to the other.

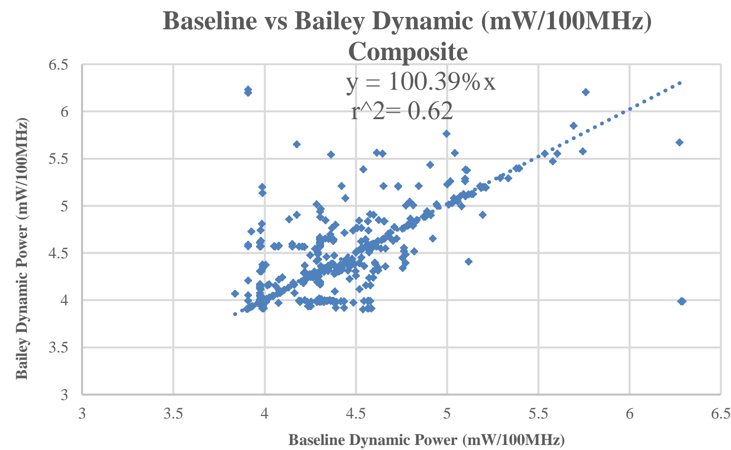
To conclude this Section has demonstrated that the dynamic power changes as the IPC changes, and further it has also demonstrated that the area changes as the IPC changes. Therefore, at this stage is important to understand how to use these differences to gain power benefits.

Dynamic power per state

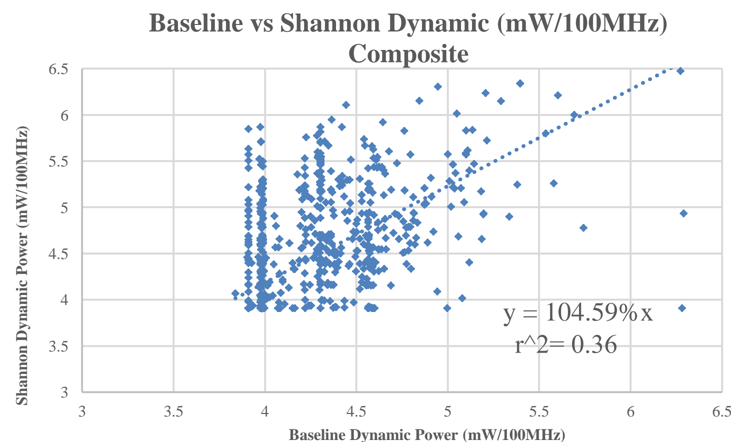
The advantage of using parallelism is that it can reduce the frequency of clock cycle and so the dynamic power without changing the time of execution. Another way to see this improvement is to analyse the number of states in the state machine because each state is performed in a clock cycle. Therefore less state means the CPU applies less clock cycle to perform the function and as a consequence decreases the power. In such as the way in which Shannon's decreasing the number ameliorates the power.



a) Dynamic Power (mW/100MHZ) Baseline vs Koopman



b) Dynamic Power (mW/100MHZ) Baseline vs Bailey



c) Dynamic Power (mW/100MHZ) Baseline vs Shannon

FIGURE 4.23: Dynamic Power (mW/100MHz) for the optimized benchmarks in Composite architecture. The result describes that increasing the number of IPC increases the dynamic power. In fact Shannon has a higher number of IPC and also as the higher value of power density

To demonstrate this theory, Table 4.4 compares the dynamic power per number of states for the Composite architecture in both Baseline and Shannon optimization.

TABLE 4.4: Dynamic power per state.

Core	Ba St	Sh St	B.Dyn.P mW/100MHz	S.Dyn.P. mW/100MHz	Bas nW/c	Sha nW/c	Ratio
I-0012-00	5	2	4.772	4.696	2.382	0.939	39.36 %
I-0012-02	5	2	3.910	3.910	1.955	0.782	40.00 %
I-0013-00	6	2	4.304	5.161	2.582	1.032	39.97 %
I-0014-00	6	3	4.568	4.385	2.741	1.315	47.99 %
I-0014-01	7	3	4.565	4.439	3.196	1.331	41.67 %
I-0014-03	7	3	4.653	4.335	3.257	1.300	39.93 %
I-0014-03	7	3	3.910	3.910	2.737	1.173	42.86 %
I-0015-00	7	4	3.910	3.910	2.737	1.564	57.14 %

Observing Table 4.4; The first column selects a number of cores. At each core associate the number of states the architecture has when is performed in Baseline or in Shannon optimization, see column two and three. Column four and five displays the dynamic power at 100MHz for each core when is performed in Baseline and Shannon optimization. Column six and seven shows the dynamic power per number of states. Finally column eight shows the ratio.

To calculate the dynamic power per number of states. At first the dynamic power for each optimized core is divided for 100MH, this help to calculate the amount of power each state consumes. Then, the result is multiplied for the number of states. Finally to have a better reader the power is converted from mW to nW.

Table 4.4 demonstrates that reducing the number of states reduces the power consumption. For a better view, Figure 4.24 shows the dynamic for the number of states

Shannon reduces the number of states; therefore it achieves a better power efficiency. Again, it shows that the dynamic power for the selected cores improves the efficiency of cores by more than 50 % in some cases. In the next sections this topic becomes clearer. This is an important demonstration of the effect of scheduling: while power per state does not change much. Considering the number of states the power benefit is significant.

4.8.2 Comparing the Dynamic power of the optimised benchmarks in Wave-core architecture

To fully understand the characteristic of Wave-core, the dynamic power is plotted for the three optimisations. Figure 4.25 shows the comparison of the optimized benchmarks cores vs the Baseline cores for the Wave-core architecture.

The optimised Wave-core, in contrast with Composite, changes the dynamic power. For instance Koopman Figure 4.25 a) decreases the dynamic power by around 3 %. Bailey Figure 4.25 b) exceeds this, at around 5 %, and Shannon Figure 4.25 c) at around 3 % is very similar to Koopman. As a result of this, Figure 4.25 demonstrates that; by increasing the number of IPC per state, Wave-core improves the dynamic power. In addition a further reduction could be observed by applying clock gating. (See next Section for more details)

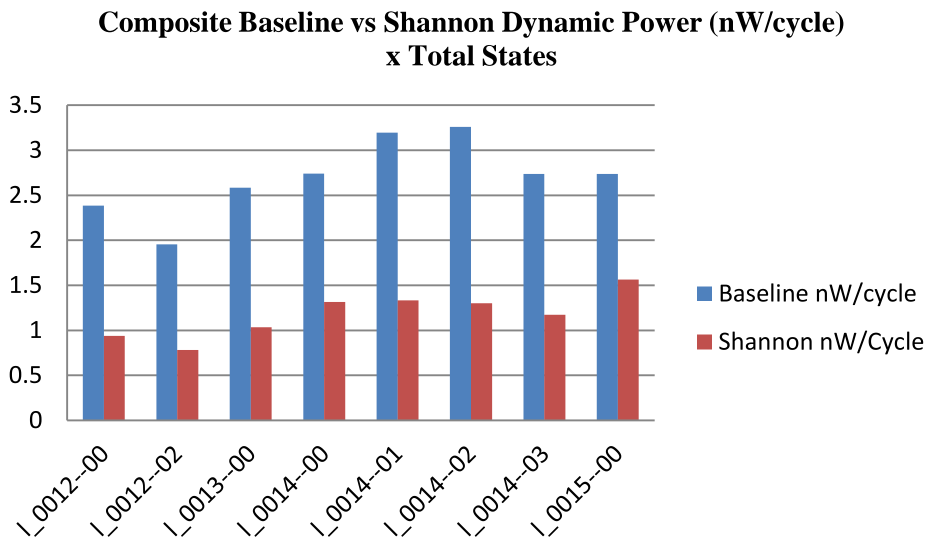


FIGURE 4.24: Comparing the dynamic power for the total states in Composite architecture for Baseline vs Shannon optimisation (Shannon reduces power consumption by more than around 60 %)

4.8.3 Composite vs Wave-core Dynamic power of the optimised benchmarks

To fully observe this phenomenon, the Shannon optimisation for both Composite and Wave-core architecture is compared.

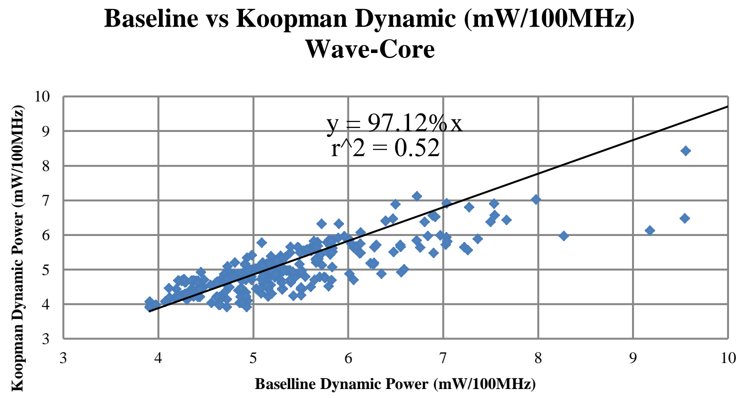
According to the data Figure 4.26, Wave-core architecture in Shannon optimization has a slightly improvement in dynamic power. In fact, for the first 577 cores the dynamic power is the same, that because both cores act just as a transient. Then, Wave-core does better, that because in a few cores Wave-core has a better optimization of logic (See Venn diagram Figure 4.16 to understand better how Wave-core synthesize the hardware).

To observe the data in a different way, Figure 4.27 shows just the most changing cores. The cores selected are the ones for which the dynamic power varies. Looking at data, the first 337 cores appear consuming less power when Wave-core architecture is applied. Then for the remaining cores, Composite does better.

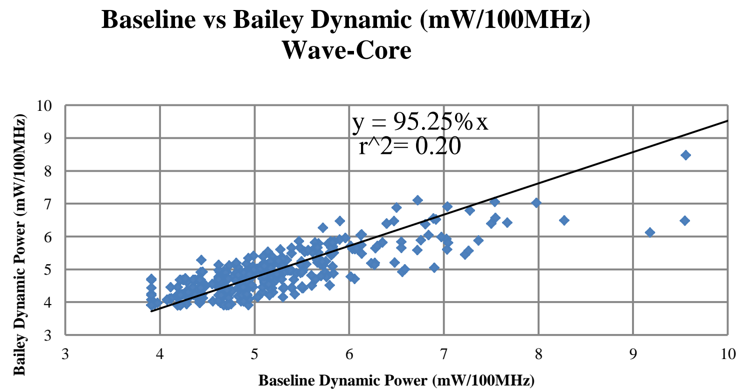
Another important point comes out observing Figure 4.23 and Figure 4.25 is that the scatter data along the y axis increases the distance more in Composite than in Wave-core architecture. So far the opposite has been shown. This happen because the Dynamic power in such way could be seen as opposite of static power. While the static power represents the amount of logics, the Dynamic power represents the active logic per time. Therefore, according with the data. The Wave-core architecture and its multi dedicate state has a better use of active logic. This has been demonstrated in earlier Sections.

To understand if both Composite and Wave-core architectures produce different power according to the optimized benchmark, Table 4.5 shows the Pearson correlation.

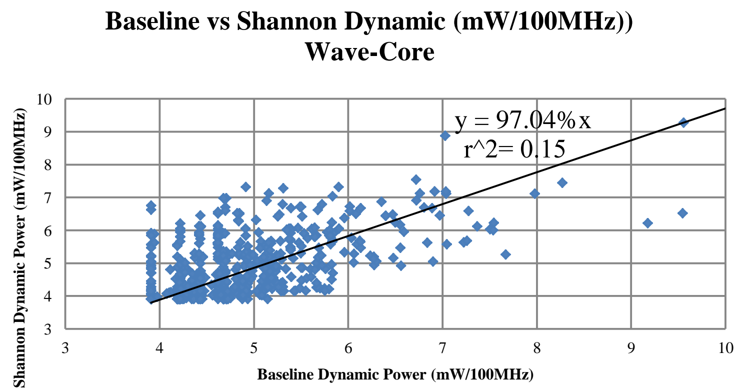
The value of r in Composite architecture for Baseline vs Koopman is 0.89. It means there is a quite strength correlation in more details the increasing of values between x and y axis is very small. On the Baseline vs Koopman the r = 0.78. It



a) Baseline vs Koopman Dynamic (mW/100MHz)



b) Baseline vs Bailey Dynamic (mW/100MHz)



c) Baseline vs Shannon Dynamic (mW/100MHz)

FIGURE 4.25: Dynamic Power for the optimised benchmarks in Wave-core architecture

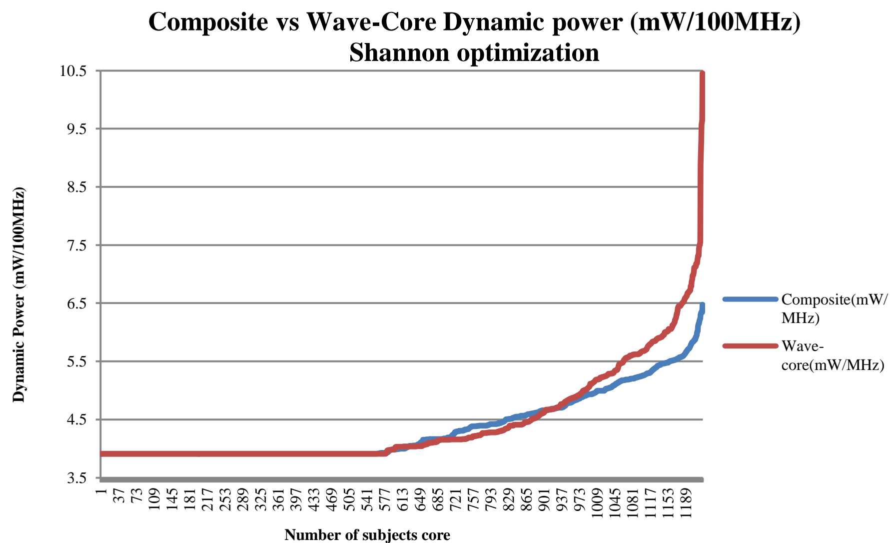


FIGURE 4.26: Composite vs Wave-core Dynamic power (mW/100MHz) Shannon optimisation. Composite does better in power density. But for few cores the Wave-core does slightly better

TABLE 4.5: Pearson correlation coeffcient Dynamic power

Parameters	Composite Arch. (r)	Wave-core Arch. (r)
Baseline vs Koopman D. Power	0.89	0.72
Baseline vs Bailey D. Power	0.78	0.44
Baseline vs Shannon D. Power	0.60	0.38

means that increasing the IPC the strength becomes weaker. In fact Shannon that has the higher IPC r decreases more. It is r = 0.6.

Performing the Pearson correlation coefficient r in Wave-core architecture for Baseline vs Koopman is 0.72. In Baseline vs Bailey r = 0.44. Finally in Baseline vs Shannon r = 0.38. That is to say. IPC decreases the strength of correlation and more another observation is that the core are grouped very close to the positive slope in Koopman and Bailey. But for Shannon the situation seems a bit different. According to the data the Wave-core for higher IPC increase the distance from the line. Therefore clearly the two architecture produce different data.

4.9 Comparing the Total Power of the optimised benchmarks in Composite and Wave-core architecture

The total power is another way to observe power consumption. To calculate the total power, static power plus dynamic is summed [23]. Observing these three factors, static, dynamic and total power, help to understand better how core's behave in consuming energy and in which form the power is dissipated. Below, the Composite and the Wave-core architecture are analysed. This section also tries to understand

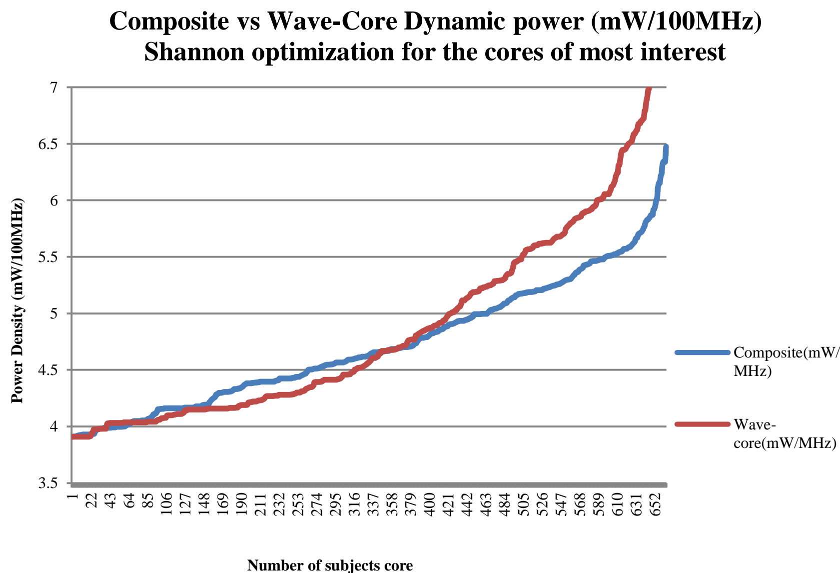


FIGURE 4.27: Composite vs Wave-core Dynamic power (mW/100MHz) Shannon optimisation for the most influential cores (The Wave-core architecture proves that IPC could help to reduce the dynamic power, and in a few cases it could slightly decrease the dynamic power)

how the architectures behave when the clock gating technique is applied in Composite architecture and when clock and power gating are applied in Wave-core.

4.9.1 Composite total power for the optimized benchmarks

To understand how the Composite behaves Figure 4.28 shows the total power for the four benchmarks

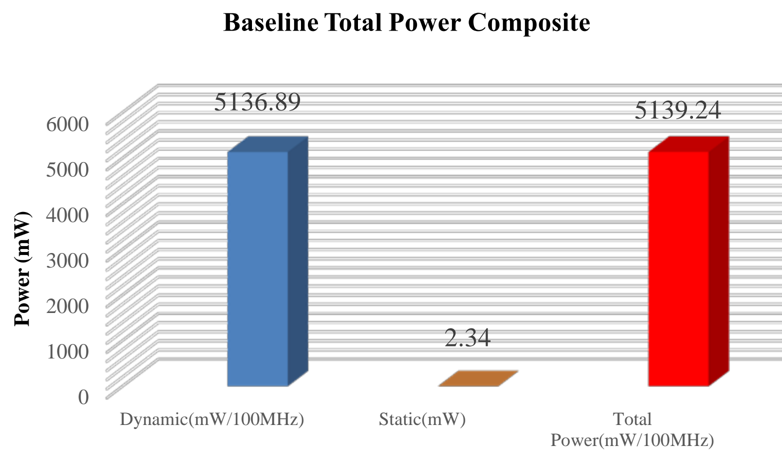
Notice; to have a better reading, the total number of cores are summed by themself and grouped into three main categories; Dynamic power, Static power and Total power.

That is to say; this sum does not represent the exactly value of each core, neither the real power that the benchmarks or cores consume if interfaced to the host CPU. But, it does a better description of the data. Therefore, it clarifies and simplifies the differences between the four benchmarks

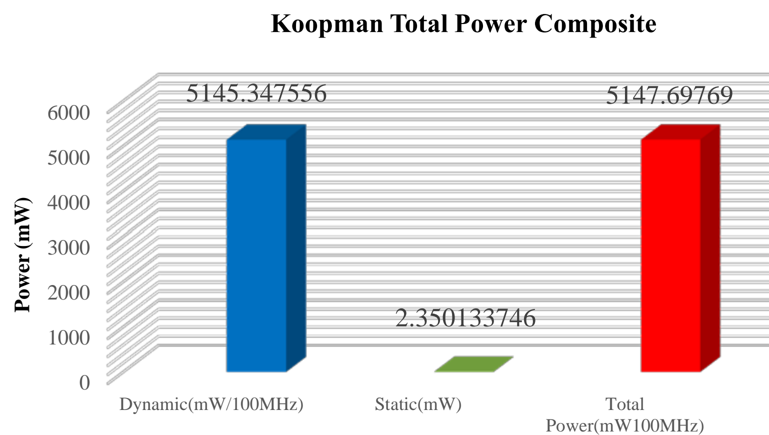
The Baseline power consumption Figure 4.28 a) presents the following data: the sum of all cores for the dynamic power is 5136.89mW @100MHz, the static power is 2.34mW, therefore the total power is 5139.24mW @100MHz.

The Koopman optimisation Figure 4.28 b) presents: the total cores have dynamic power of 5145.34mW @100MHz, the static power is 2.35mW and so the total power is 5147.69 @100MHz. Notably, the power is slightly higher for both dynamic and static. The dynamic power is increased because Koopman slightly increases the instructions per clock cycle; therefore it increases the logic per state. However, the increase is so small that the static power is almost the same.

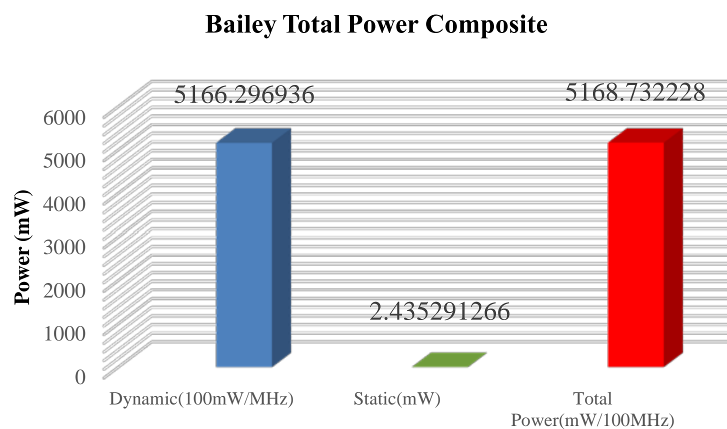
The Bailey optimisation Figure 4.28 c) demonstrates better how the power consumption increases with the increasing of instructions per clock cycle. For instance, the dynamic power is 5166.29mW @100MHz and the static power is 2.43mW.



a) Baseline Total Power



b) Koopman Total Power Composite



c) Bailey Total Power Composite

FIGURE 4.28: Composite total power for the optimized benchmarks, Bailey uses more power

The last optimisation to analyse is Shannon Figure 4.29 d). It shows a higher increase of power consumption when compared to the others. Again, this increases is explained for the increasing of instructions per clock cycle. The number of states in the state machine have been observed. Shannon decreases the number of states at the cost of increasing the number of logics, therefore the dynamic power increases. Similarly, this increasing of logic increases the area, and for this reason the static power increases.

Total power applying clock gating in Composite architecture

Figure 4.31 compares the Baseline power consumption vs the reduced frequency of Shannon power consumption and explains the benefit.

Parallelism is a very good technique to improve either the power or the timing.In this work the increasing of parallelisms is used to decrease the power consumption; for example, if a function could be computed in three states and each state is executed in one second, the full function is completed in three seconds so the clock cycle is set up at 1Hz. For this reason, if the function is optimized and reduce the number of states (let assume to one state), it can decrease the frequency to 0.3Hz, which is equal to three seconds, and so the function has the same time of execution. In this case for the power equation P=V^ 2*F*C, for C and V constant the power decreases of 1/3.

To better understand this technique Shannon optimisation is selected because it has a higher increase of IPC and so a decrease of states. Figure 4.30 compares the number of states in the Baseline vs the number of states in Shannon in terms of percentage and then the same percentage will be subtracted from the dynamic power.

Figure 4.30 compares the Baseline states with the Shannon states. It shows that Shannon decreases the number of states by approximately 35 %. This tells that the frequency could be reduced by nearly 35 % without modifying the execution time of the functions. Reducing the frequency of about 35 % implies a reduction of dynamic power of circa 35 %.

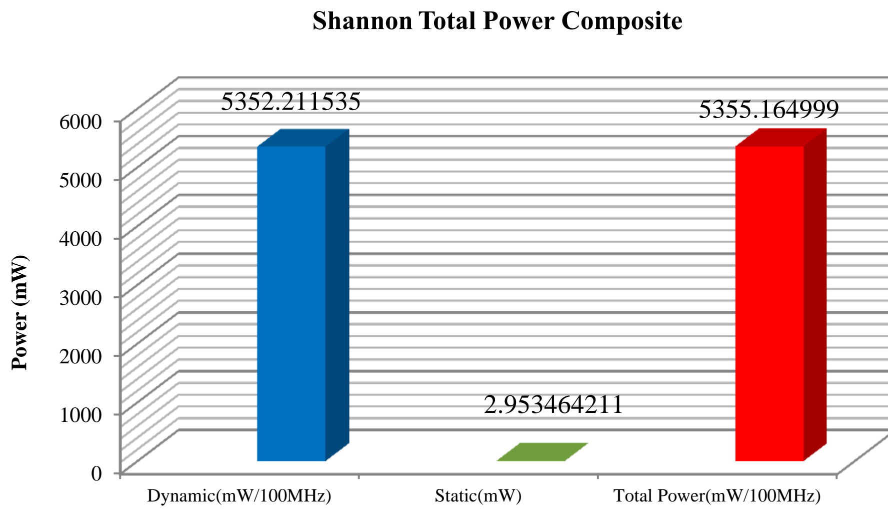
Figure 4.31 a, b shows the Shannon power consumption reducing the frequency by 35 % in comparison with the Baseline power consumption.

Comparing the two data, it shows, that the dynamic power is now 3478.96mW/ 65MHz for Shannon Figure 4.31 b while for Baseline it is 5123.89mW/100MHz Figure 4.31 b).

The static power for Shannon is 2.95mW, while on the other hand the static power for Baseline is 2.34mW, because decreasing the frequency decreases just the dynamic power. Finally, the total power for Shannon is 3841.91mW/65MHz while for Baseline it is 5139.24mW/100MHz. Indeed, the Shannon architecture is now at 65MHz, but it executes the function in the same time that the Baseline executes the function, as it has explained above, while also using less power. As a result it has demonstrated that Shannon optimisation helps the core to decrease the power consumption by around 35 %.

4.9.2 Wave-core total power for the optimized benchmarks

So far, the power consumption for Composite architecture has been observed. Now, the same observation is done for Wave-core architecture. Figure 4.32 and Figure 4.33 show the power consumption for the four sets of benchmarks.



d) Shannon Total Power Composite

FIGURE 4.29: Composite Power Consumption for the Shannon benchmarks

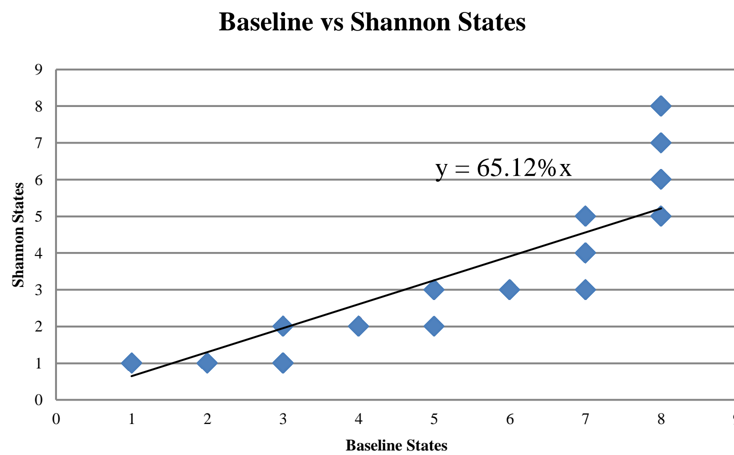
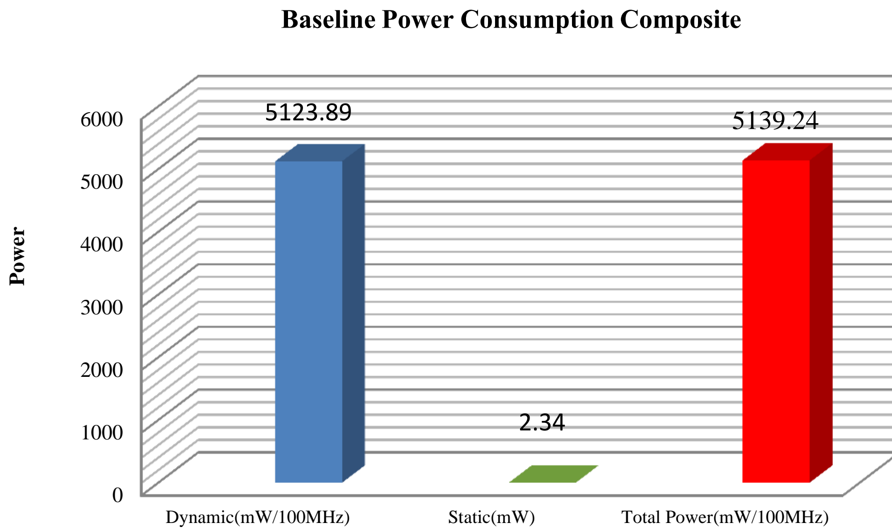
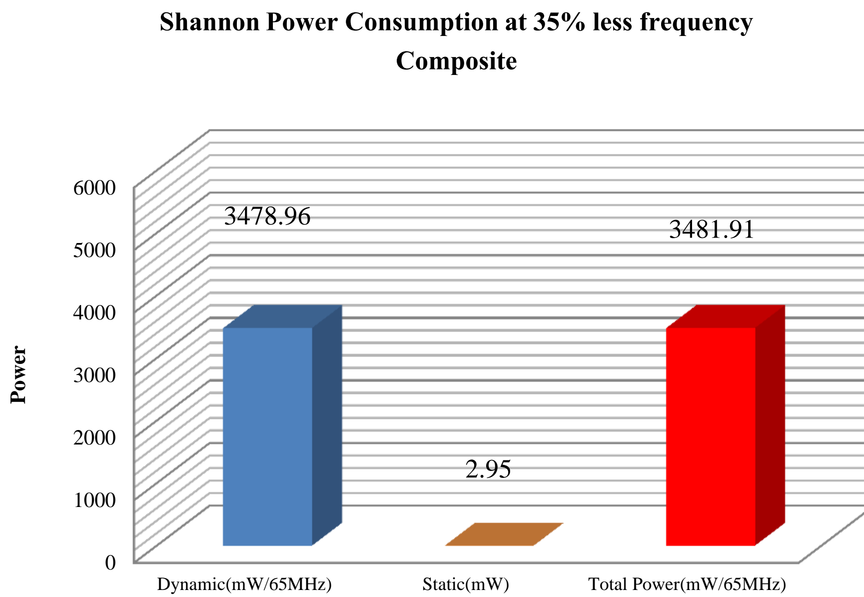


FIGURE 4.30: Comparison of Baseline vs Shannon States. Shannon decreases the number of state around 35 %. Decreasing the number of states mean the architecture can run at 35 % less frequency and so saving energy.

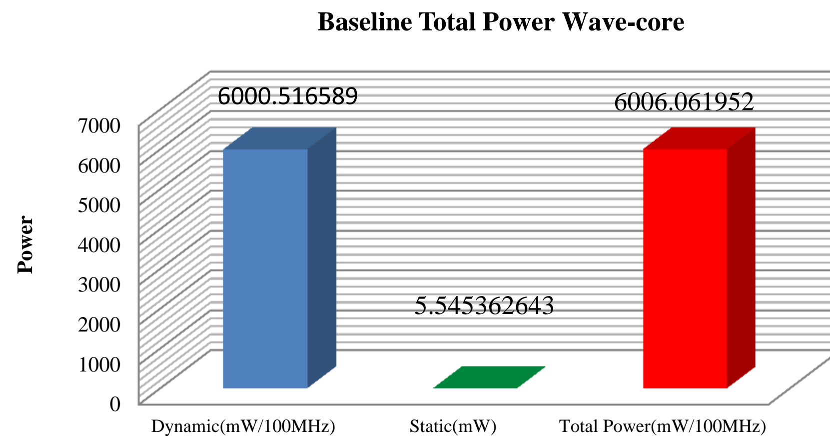


a) Baseline Power Consumption

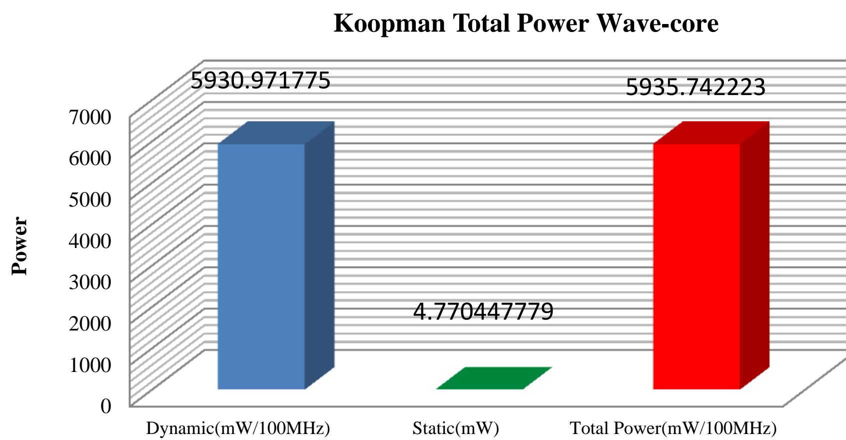


b) Shannon Power Consumption 35% less Frequency

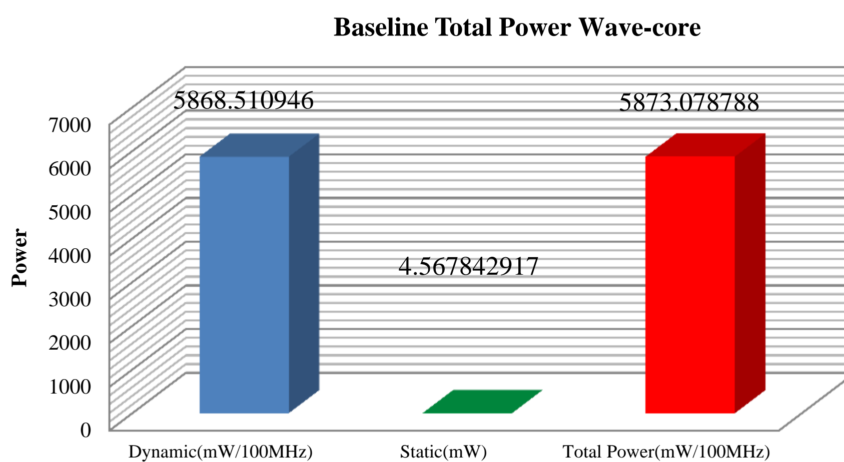
FIGURE 4.31: Baseline Power Consumption, and Shannon Power Consumption at 35 % less frequency (The Shannon optimisation, which increases IPC by 35 %, makes a high contribution to decrease the power consumption)



a) Baseline Power Consumption Wave-Core

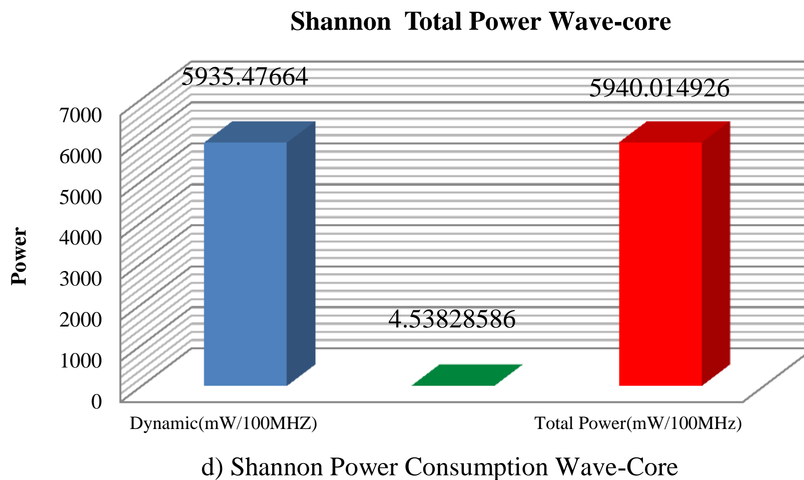


b) Koopman Power Consumption Wave-Core



c) Bailey Power Consumption Wave-Core

FIGURE 4.32: Wave-core total power for the optimized benchmarks, Bailey uses more power



d) Shannon Power Consumption Wave-Core

FIGURE 4.33: Wave-core Power Consumption for the Shannon benchmarks

First, all optimisations reduce total power. The sum of total power for all cores inside the Baseline benchmark Figure 4.32 a) is 6006mW/100MHz, with dynamic power 6000mW/100MHz and static power 5.5mW. Koopman optimisation appears to reduce this slightly Figure 4.32 b) The total power is 5935mW/100MHz, which is divided into 4.7mW for the static power and 5935mW/100MHz for the dynamic power. Bailey Figure 4.32 c) does better; the total power is 5873mW/100MHz, which is the sum of static power 4.5mW plus the dynamic 58.68mW/100MHz. Finally, Shannon Figure 4.33 d) has total power of 5940mW/100MHz. The dynamic is 5935MW/100MHz and the static power 4.5mW. Two observations must be reported. First the dynamic power changes as the number of states change. Furthermore reducing the frequency by 35 % as see for Composite, the dynamic power for Wave-core has a father reduction of 35 %. Second the static power decrease with the increasing of IPC

Chapter 2.8 explained that the Wave-core architecture has the advantage with its multi stage to reduce the static power by applying power gating. In other worlds, the different stages of the Wave-core architecture could be turn off until needed and therefore save energy. (See next section for more details)

4.9.3 Power gating and Clock gating Composite vs Wave-core Total Power

Increasing the number of Instruction per clock cycle gives the opportunity to decrease the frequency and so improve the dynamic power. And more, the characteristic of the Wave-core with its multi processes (see chapter 2.8 for more details), gives the opportunity to implement a clock gating to reduce static power.

To better understand the concept, the Shannon optimisation is analysed in an analytical way; therefore, both architectures Composite and Wave-core are compared using 35 % less frequency, because the FSM decreases the number of state around 35 %. That could be applied for the dynamic power. On the other side the static of Wave-core is considered when just one stage is activated at a time. Must be clear

that; just the Wave-core architecture can support the power gating technique. Because it has a multiprocess architecture and so each process can be turn off without lost the state.

Figure 4.34 shows the power consumption applying power gating and clock gating.

Applying power gating, Wave-core reduces the static power. This is another relevant result. So far this thesis has argued that by extending the area the static power increases, therefore Wave-core architecture appears to suffer. However, this section has demonstrated that this weakness could be obviated by adding a simple and consolidated CMOS technique that just Wave-core architecture can support. Finally, the Wave-core architecture has demonstrated an importance features to reduce static power.

4.10 Comparing the Power Density of the optimised benchmarks in Composite and Wave-core architectures

The main purpose of using accelerators is to improve power consumption, and so ameliorate Dark Silicon, especially the power density of the chip. A methodology is implemented here that synthesises thousands of software functions into hardware cores which are then ready to be interconnected to the main CPU.

This thesis started by explaining that the methodology has some advantages for the design cycle as it gives the opportunity to optimise the function at compiler level and so helps the synthesis tool to generate more power efficient cores. To prove this theory the Baseline benchmarks have been optimized into three optimisations that are here called Koopman, Bailey and Shannon and then the power density in both Composite and Wave-core architectures has been estimated.

4.10.1 Composite Power density for the optimized benchmarks

This section starts by observing the power density for the Composite architecture. Figure 4.35 shows the average power density at 1MHz for the four optimised groups of benchmarks, when the cores work at 0.5 % of the time.

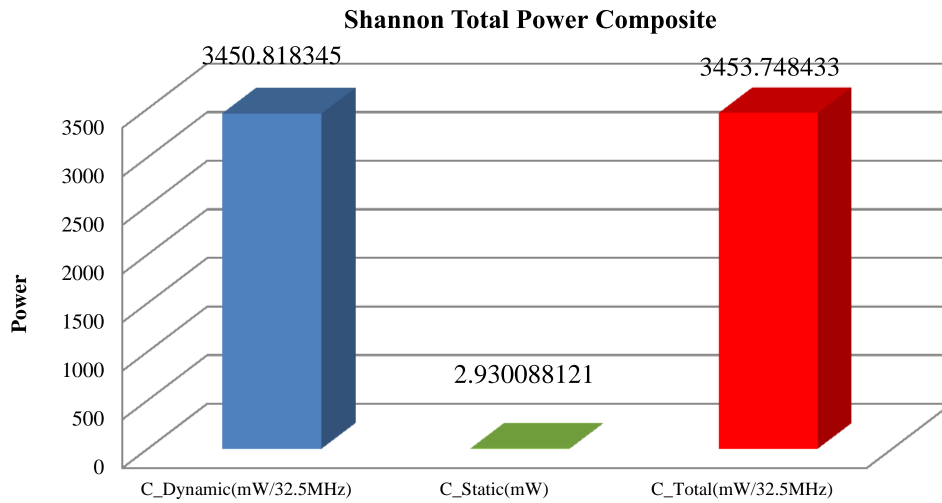
In detail; Figure 4.35 shows that Baseline uses approximately 0.00954W/cmˆ 2, and Koopman around 0.00959W/cmˆ 2, which is almost exactly the same.

Bailey gives a slightly better result. In fact, each core consumes approximately 0.00947W/cmˆ 2. Shannon which has the higher amount of parallelism is the one which consumes less power density. It is 0.08868W/cmˆ 2. For instance, the static power increases by more than the other optimisations, and so this increase of logic returns a benefit in terms of power density.

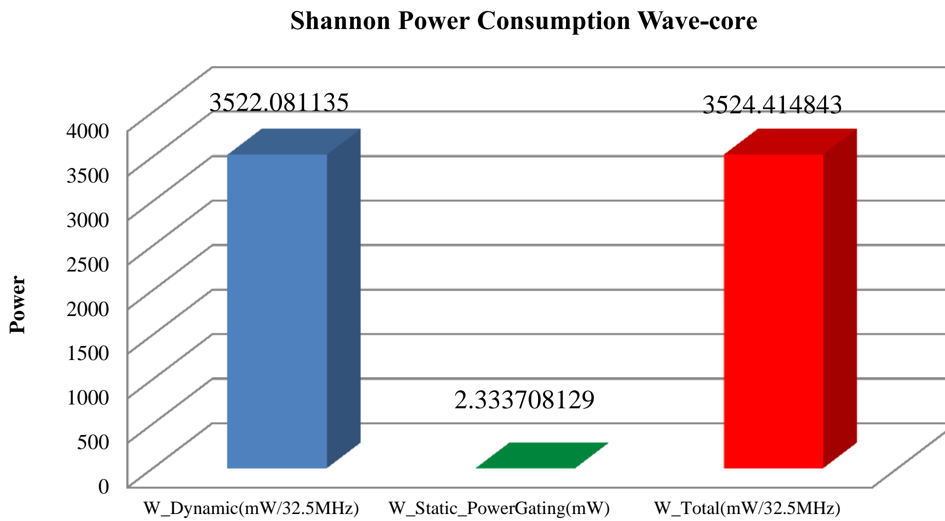
Power Density for the Composite architecture when the clock gating is applied

So far, the meaning of using parallelism to decrease the power in a Composite architecture has been explained. To observe how much improvement the power density gains when a clock gating is applied, first the number of states in each optimised benchmarks vs Baseline benchmarks are compared.

In Figure 4.36 a) Koopman decreases the number of states by around 20 %, which means they can reduce the frequency of the clock by approximately 20 % and so have a decrease of dynamic power by around 20 %.



a) Shannon Composite Power Consumption clock gating



b) Shannon Wave-Core Power Consumption clock gating and power gating

FIGURE 4.34: Comparing Composite vs Wave-core Shannon optimisation applying clock gating for Composite and power gating and clock gating for Wave-core. The Wave-core decreases the static power

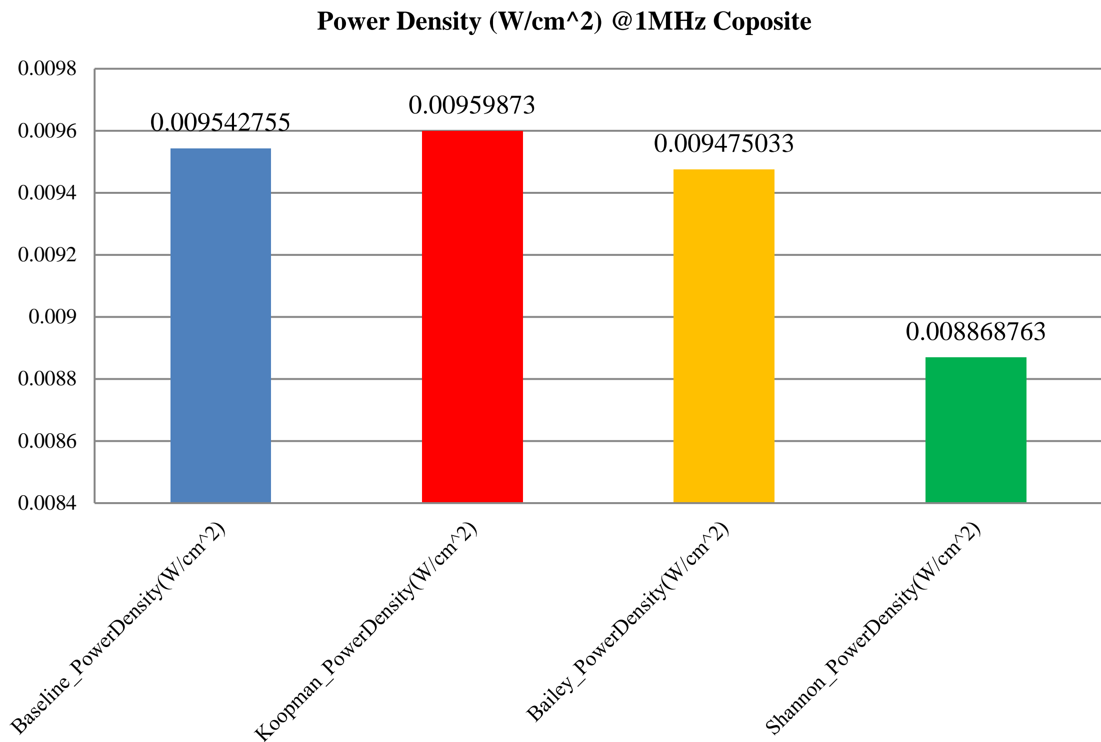


FIGURE 4.35: Composite power density for the four benchmarks. Shannon decreases the power

Comparing the Baseline with Bailey, Figure 4.36 b) shows that the number of states is very similar to Koopman; in fact it decreases the states by about 25 %. Once more, it also means they can decrease the frequency of clock by about 25 %. Finally, Shannon Figure 4.36 c) decreases the state by almost 35 %, which is a large improvement – and it already consumes less power density. For this reason, this section refers to Shannon optimisation to understand how much benefit it can give to the overall architecture.

To try to remove any possible microwatts from the architecture, the clock frequency from Shannon optimization is decreased by about 35 %. Section 4.9.1 describes the number of states directly connected with the time of execution; therefore Shannon sends the same amount of data per time as when the Baseline executes the data if the frequency of the clock cycle is decreased by nearly 35 %. Figure 4.37 shows the power density of the Baseline, the power density of Shannon and the power density of Shannon reducing the frequency about 35 %.

To calculate the Shannon average power density at the 35 % of frequency, firstly, the average dynamic power is calculated at 650kHz which is the 30 % of 1MHz, then the average static power is added and finally the result is divided by the average area.

The formula to calculate the power density is: **Pd = (((DP+SP)/A)*0.5)/100**. Where, Pd= Power density reducing the frequency of 35 %, SP= Static Power, DP= Dynamic power at 650kHz, A=Area.

Figure 4.37 shows the Baseline power density. Shannon power density and the Shannon power density reducing the frequency of 35 %.

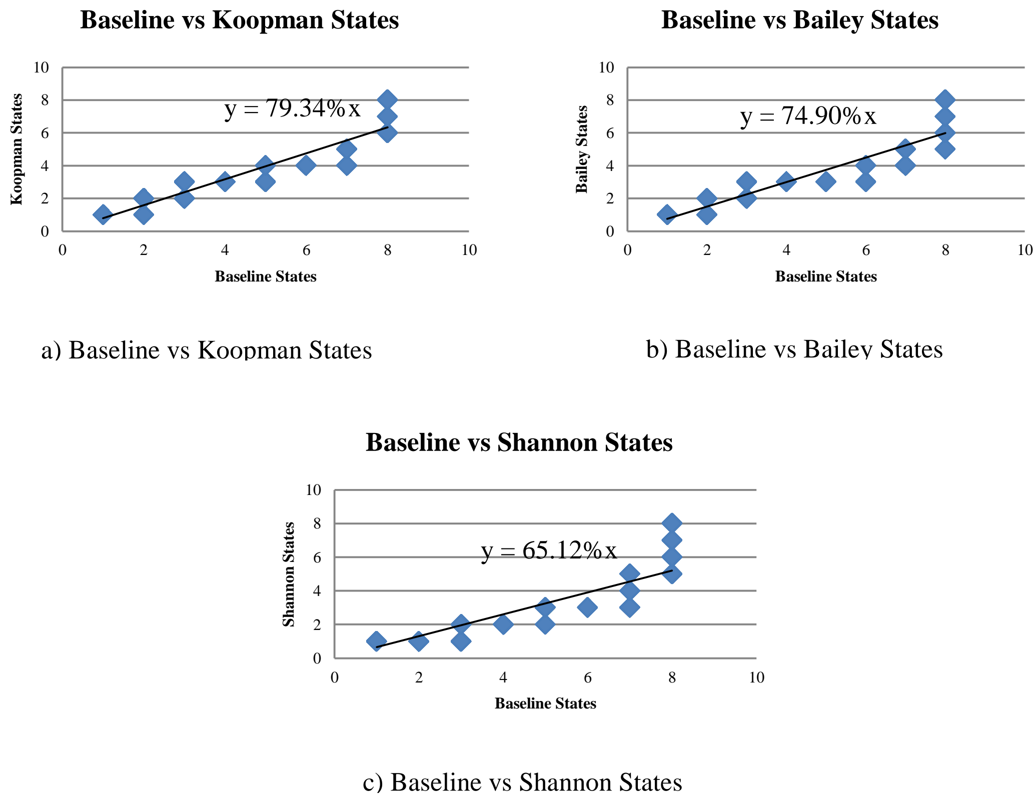


FIGURE 4.36: Comparison of the number of states in the optimised cores with the Baseline. Shannon has the lower number of states. It means is the best candidate to reduce power. Because reducing the number of state it can reduce the frequency and so decreases the dynamic power

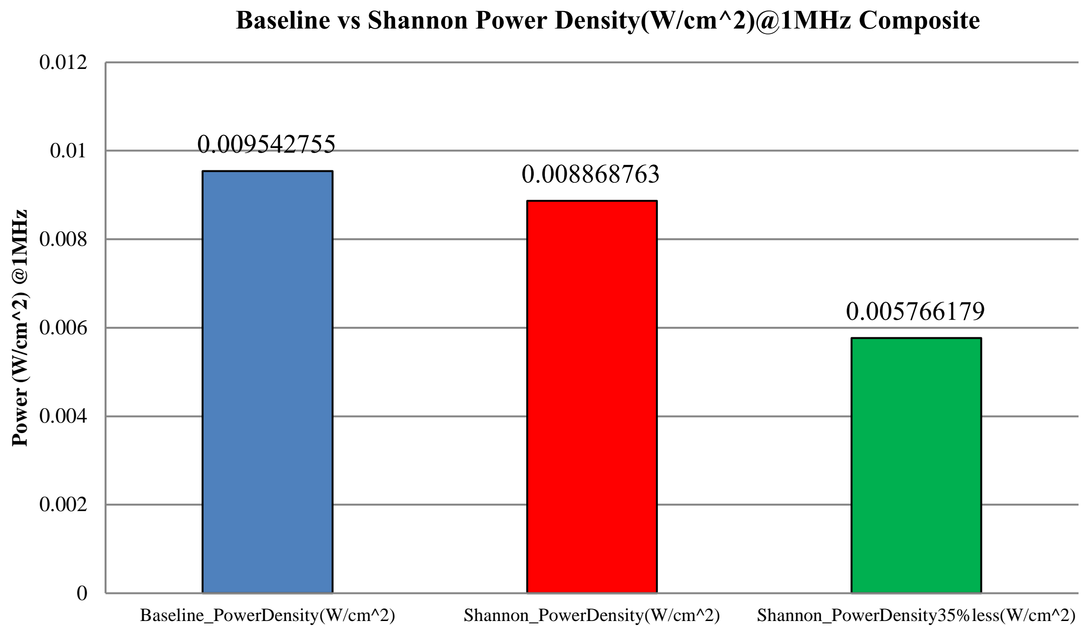


FIGURE 4.37: average Power density for the Baseline, Shannon, and Shannon reducing the frequency by 35 %. The result show; Shannon decreasing the number of state can reduce the frequency and improve the power density

According to the data, the Baseline consumes on average 0.009W/cmˆ 2, while for the Shannon the figure is 0.08W/cmˆ 2, which is around 10 % less. In addition, reducing the frequency on average cores in Shannon consume just 0.005W/cmˆ 2. As a result, Shannon has demonstrated to improve the power density by about 43 %

4.10.2 Wave-core Power density for the optimized benchmarks

This Section observes how the power density behaves for the Wave-core architecture.

Figure 4.38 shows the power density for the four benchmarks. The Wave-core architecture makes the cores more compact for the optimised benchmarks; in fact, reducing the area increases the power density

In more detail, the Baseline has an average per core of 0.00734W/cmˆ 2, and for Koopman this increases slightly to 0.00758/cm. Bailey has a power density of around 0.00794W/cmˆ 2 and finally Shannon, which has the highest IPC, has power density of circa 0.00811W/cm. From this analysis it becomes clear that increasing the number of IPC increases the power density.

4.10.3 Wave-core vs Composite Power density when when power gating and clock gating are applied

Section 4.10.2 showed that increasing the IPC leads to an increase in the power density. However the Wave-core architecture has the ability to reduce the power by applying power and by using clock gating techniques. On the other hand the Composite architecture also has the ability to reduce the power by applying the clock

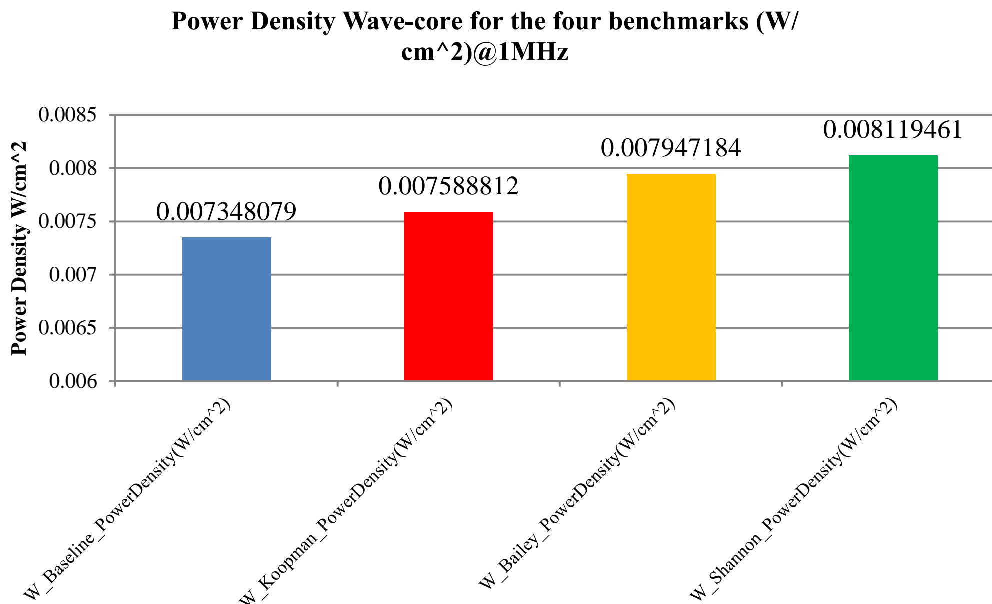


FIGURE 4.38: Power density for the four benchmarks (W/cmˆ 2)@1MHz. According to the data increasing IPC increases the power density.

gating technique. Therefore to obtain a more accurate comparison, this Section calculates the power density for Shannon optimization for Composite both when the clock gating is applied and for the Wave-core when the clock and the power gating are applied.

Figure 4.39 compares the two architectures for power density for Shannon optimisation .

The power density for an approximation of each core in Composite architecture for the Shannon optimisation after applying the estimated clock gating is about 0.0057W/cmˆ 2. On the other hand applying both power and clock gating to Wave-core Shannon optimisation architecture reduces the power density by up to about 0.006W/cmˆ 2, which is around 50% less than Composite. Therefore the Wave-core has been demonstrated to produce a high improvement in dissipating power density.

4.11 Pearson Correlation review of statistical significance of results in Composite and Wave-core architecture

The correlation coefficient is used to understand if the different optimizations produce different results in terms of area, timing and static and dynamic power.

The main characteristic to observe is to understand the straight of Pearson correlation r:

Before starting to analyse and understand the data a brief summary of the Composite and Wave-core architecture is provided.

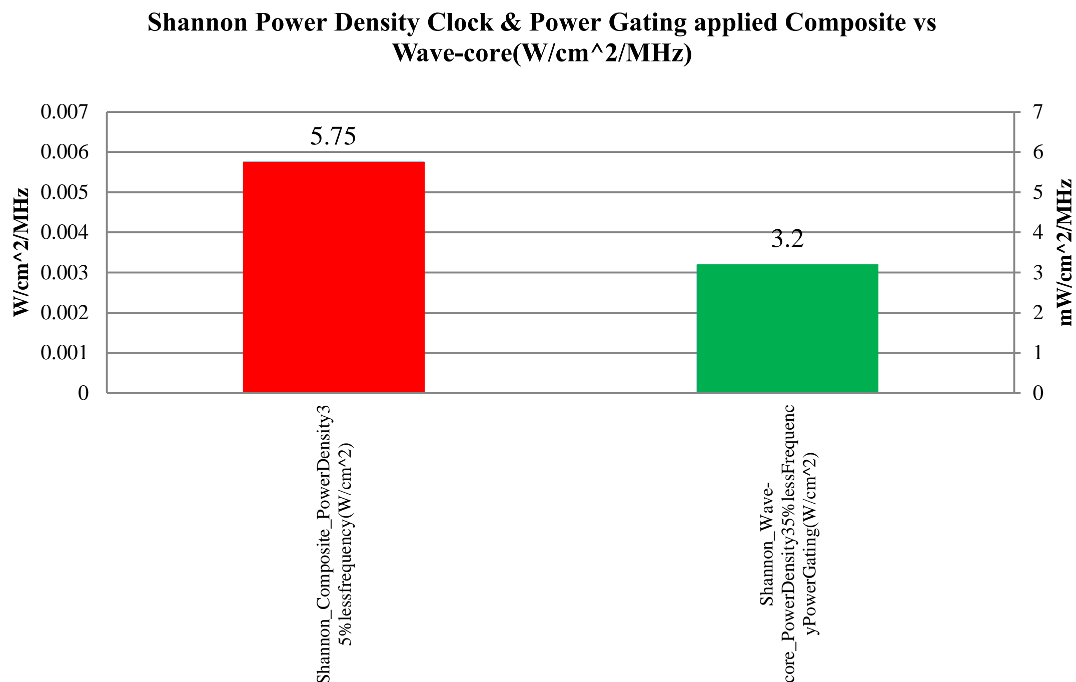


FIGURE 4.39: Power density for the four benchmarks (W/cm^2)@1MHz. According to the data increasing IPC increases the power density.

The Composite architecture, as has already been mentioned implements the core using a classical state machine, therefore optimizes the area per logic. In this way the architecture appears to be grouped, uses less area but increases power density. On the other hand the Wave-core is an evolution of the Composite architecture. It treats each state as dedicated hardware and each unit is interconnected using signals. In this way it extends the area of logic for a better distribution of power and so reduces the power density. The Wave-core also reduces the generation of a hot area because the logics are used for less time. (See section 2.7, 2.8 for more details)

Another point to highlight is that both architecture are generated using four optimized benchmarks with the characteristic to have a different amount of parallelism. (The different benchmarks have been explained earlier in this Chapter)

Observing the Pearson Correlation Table 4.6, it clearly shows that the values change for the different optimizations.

Table 4.6 shows that increasing the number of IPC , the Pearson correlations becomes weaker . In fact comparing the unoptimized benchmarks or baseline with Koopman' benchmarks the correlation is quite strong. On the other hand, comparing the Baseline with Bailey and then with Shannon that has a higher amount of IPC the Pearson correlation becomes weaker. To summarize; the r'area for Composite is from 0.98 to 0.62. On the other hand for the Wave-core r' area is from 0.89 to 0.76.

The r' timing is from 0.87 to 0.61 for Composite, while for the Wave-core is from 0.86 to 0.46.

Looking at the Static power the Pearson correlation r; is from 0.91 to 0.56 for the Composite. In contrast for the Wave-core is from 0.89 to 0.74.

Finally the r' dynamic power for Composite is from 0.89 to 0.6, in comparison the

TABLE 4.6: Pearson correlation coefficient for the four factors

Parameters	Composite Arch. (r)	Wave-core Arch. (r)
Baseline vs Koopman Total Area	0.98	0.89
Baseline vs Bailey Total Area	0.84	0.90
Baseline vs Shannon Total Area	0.62	0.76
Baseline vs Koopman Timing	0.87	0.86
Baseline vs Bailey Timing	0.84	0.58
Baseline vs Shannon Timing	0.61	0.46
Baseline vs Koopman S. Power	0.91	0.89
Baseline vs Bailey S. Power	0.84	0.90
Baseline vs Shannon S. Power	0.56	0.74
Baseline vs Koopman D. Power	0.89	0.72
Baseline vs Bailey D. Power	0.78	0.44
Baseline vs Shannon D. Power	0.60	0.38

Wave-core r′ is from 0.72 to 0.38. To conclude, the data confirms that the different benchmarks produce a different set of data.

4.12 Summary

One of the novel contributions of this work is to try to improve the performance of the hardware at compiler level for stack structure cores.

Chapter 2 has explained some correlated work and to what extend compiler optimisation techniques can improve the performance of the design. However, not much work has been done on compiler optimisation for stack architecture. Therefore this Chapter concentrates on this feature.

Section 4.4 looked at the hardware IP after the function is optimised using a compiler scheduling approach and so identified how the cores behave. For this reason, the section describes the amount of IPC for the three optimized benchmarks. Shannon increases the number of IPC up to 8 instruction per clock cycle.

Section 4.1, 4.2 recognised that the number of inputs/outputs increase as the number of instructions per clock cycle increases, and as an obvious consequence the number of states decreases, See section 4.3. This result suggests that there is an increase of IPC.

To conclude; these changes modify the structure of the architecture. In fact Composite increases the area up to 23 % when IPC increases; on the other hand Wave-core decreases the area by around 30 % when the IPC increases. See section 4.5

Wave-core decreases the timing around 4 % and the leakage by approximately 30 % when increasing the IPC.(See Section 4.6 and 4.7) In contrast Composite increases the leakage by about 24 % (see Section 4.7). In some cases Wave-core architecture decreases the dynamic power when benchmarks are optimized with scheduling optimization. See Section 4.8

Finally, Wave-core reduces power density by around 50 % compared with Composite. (See section 4.10)

To conclude Dark silicon requires the development of more efficient CPUs in terms of power consumption. A possible solution is a CoDA processor, which when surrounded by co-processors can consume just 20 % of energy if compared with a traditional architecture [61]. One of the weaknesses of the co-processor technique is that it suffers from the complexity of hardware design.This thesis has proposed a change of direction in this respect, and in the most populated CPU architecture design, it assumes the use of Stack Machine architecture as a host because a stack architecture is power friendly (see Sections 1.1, 1.5 and Chapter 2).

As a final analysis, one of the main advantages in using a data flow stack structure is that the hardware will remain simple when it is converted from software to hardware. Therefore, this Chapter has also investigated using software optimisation such as scheduling at the compiler level to optimise the hardware, and it has found that this established technique – which is nevertheless quite new if applied to the stack architecture concept – can make a relevant contribution to the hardware IP core in dissipating power.

Chapter 5

CPU vs Core Power Analysis

To deal with power and create a more power efficient CPU, this thesis investigates how to generate accelerators. The effectiveness of the hardware accelerator is well known. To demonstrate and understand how the cores behave and how much benefit they can give to the whole architecture, a comparison is made between functions implemented at software level and at hardware level.To analyse and understand the benefit of the hardware cores, a comparison is required between synthesised hardware cores and the same code executed on a representative CPU architecture. How this is achieved affects the accuracy and scope of the analysis.

The simplest approach, with lowest accuracy, is to use an existing CPU model and make assumptions about average power consumption per instruction. For this the Lund CPU model, which is an independently developed stack processor CPU is used. The advantage of this method is that a large number of cores can be analysed without excessive effort.

In contrast, a high accuracy method of analysis is to use a transistor level simulation of a CPU whilst executing exactly the same code as the core. To achieve this the University of York NOMAD processor CPU design is used. The NOMAD CPU is modelled via a synthesised net-list, using UMC 65 nm CMOS target technology and Farady standard cell libraries. This delivers high accuracy results, but can only be applied to a small number of cores due to the effort involved.

Finally, an average power model can be extracted from the NOMAD core simulations, to allow us to make a medium-accuracy evaluation of a large number of cores. This is effectively the same technique as used in the LUND analysis , but using a more carefully derived average power behaviour.

Using these three methods, Lund Average Power', 'NOMAD netlist Simulation', and 'NOMAD average power, a comparison for CPU verses hardware cores that are broad or deep can be gotten, to give a clearer picture.

5.1 NOMAD MACHINE ARCHITECTURE

The NOMAD processor architecture, as used in the experiments for accelerator cores, operates as a single-issue out-of-order completion stack machine. Instructions must be issued in-order and one per clock cycle. However completion of instructions is allowed to be made out of order. This makes it opslightly more complex than the LUND stack CPU, (see Chapter 1) which is an in-order issue and in-order completion architecture. It is also modelled and synthesised to 65nm CMOS technology, allowing code to be simulated with very realistic power data.

The NOMAD CPU has the following register model:

- HARDWARE STACK : stack elements 0 to 15, labelled S0, S1, S2, etc, 32-bit word-width.

- XP,YP,FP,MP : addressing registers (32 bits).

  MP, XP and YP are general purpose FP is used for addressing the HLL variable stack frame.

- SP, RP : Main memory stack pointers.

- FF : System flag register.

The NOMAD CPU has a range of instruction for stack manipulation, and a set of arithmetic instructions, logical instructions, and memory instructions, including program flow (branch, conditional branch, call, return etc). The instruction set is nominally equivalent to any typical microcontroller. Stack manipulations are designed to facilitate efficient code generation and optimisation with the Shannon code optimisation techniques described in Chapter 4.

The instruction set includes special instructions for accessing local variables as an FP+offset addressing mode. This makes program execution with HLL more efficient.

The machine architecture includes Integer ALU, Shift Unit, Branch Unit, Memory queue, instruction decoder and instructions queue. These all have a penalty to increase the complexity of hardware and generate more energy.

5.1.1 Evaluation of NOMAD in terms of selected cores power analysis is undertaken as follows

1. The CPU is emulated using the Cadence logic simulator tools, using the HLL simuylation models.

2. The test sequence for each core is used to generate a vcd [1] trace for the internal node switching behaviour of the design under the given test code sequence.

3. The vcd file set is then used by the cadence synthesis tools to evaluate power based on the vcd test sequence.

4. Power data, and execution time in clock cycles, is collated for each core and compared to that of the hardware accelerator cores and their respective execution time and power usage.

5.2 Methodology of comparison (NOMAD versus Accelerator Cores)

The process of making one comparison is outlined below, and this si repeated for each code sequence and associated accelerator core, in order to produce a set of comparisons for plotting.

1. **The static and dynamic power consumption of the CPU was obtained whilst executing the chosen code sequence:**

$$CPU Static Power P_{CPU.S} = 105950 nW at 100 MHz$$

[1] cvd- A file format that represents Signals transition during the Simulation run

$$CPU Dynamic Power P_{CPU.D} = 146223020 nW at 100 MHz$$

The power consumed by CPU memory:

$$CPU Dynamic Power P_{CPU.M} = 31326606 nW at 100 MHz$$

2. **The static and Dynamic Power Consumption of the accelerator Core was obtained :**

$$CORE Static Power P_{CORE.S} = 35838 nW at 100 MHz$$

$$CORE Dynamic Power P_{CORE.D} = 11682611 nW at 100 MHz$$

3. **The total power in each case was calculated:**
   CPU Power Total

$$P_{CPU.TOT} = P.CPU.S + P.CPU.D = 105950 + 146223020 = 146328970 nW at 100 MHz.$$

   CORE Power Total

$$P_{CORE.TOT} = P_{CORE.S} + P_{CORE.D} + P_{CPU.S} + P_{CPU.M}$$

   That is equal to:

$$P_{CORE.TOT} = 35838 + 11682611 + 105950 + 31326606 = 43151005 nw.$$

   Notice that Core power also includes CPU static power, since the CPU must be at least in standby state during CORE operation

4. **The number of clock cycles per code sequence iteration are needed for each case:**

$$CYC_{CPU} = 43 (obtained from CPU simulator)$$

$$CYC_{CORE} = 12 (obtained from translator tool)$$

   The core CYC_{CPU} requires fewer clock cycles, and that the CORE must therefore be doing more work (3.58 times more).

   The true power consumption of the core for equivalent amounts of work is therefore:

$$P_{CORE.TOT.ADJUSTED} = P_{CORE.TOT}/3.58$$

   Therefore:

$$P_{CORE.TOT.ADJUSTED} = 43151005/3.58 = 12053353 nW at 100 MHz.$$

5. **The CPU and the CORE performing the same amount of work are compared as follows:-:**

$$P_{CPU.TOT} = 146328970nwat100MHz$$

$$P_{CORE.TOT.ADJ} = 12053353nWat100MHz$$

Therefore the relative power consumption of the core as compared to the CPU is:

$$RelativePowerConsumption = P_{CORE.TOT.ADJ}/P_{CPU.TOT}$$

That is equal to:

$$RelativePowerConsumption = 12053353/146328970 = 0.082 = 8.2\%$$

So in this case the core consumes around 8 % of the power that the CPU would consume when performing an equivalent instruction sequence. In other words, the Core uses 92 % less power than the CPU when doing the same work.

Repeating this method for every chosen code sequence, for both the CPU, the Composite Core and the Wave Core, a set of comparative results that allow to plot relative power performance for CPU versus core can be performed.

The Figure 5.1 illustrates the power model used. Indeed, to calculate the amount of power consumed by CPU, the static and the dynamic CPU power plus the dynamic and the static memory power are summed, while the core is completely hibernate.

On the other side, to calculate the amount of energy used by core, The CPU static power plus the memory static power plus the dynamic and the static power of the core are added.

5.2.1 Results of selected cores analysis for both CPU and Composite and Wave-core architectures

To compare and understand the change that the cores can give to the CPU architecture, the Shannon optimization benchmarks is used. In addition and for a better picture of the benefit that the hardware can give to the whole chip, the functions selected are the one with the higher number of states and IPC, because the hardware IP cores with higher number of IPC and States appear to be more power energy efficient.

Figure 5.2 shows the core power relative to NOMAD in percentage for the Composite and Wave-core architectures. For instance the cores consume very little power when compared with the CPU. In fact both, the Composite architecture Figure 5.2 a) and Wave-core architecture Figure 5.2 b) use from just a little of 2 % of energy to 22 %. This suggest that the CPU consumes up to 98 % more energy when compared with IPs core. In addition The two architectures, Composite and Wave-core appear to use the same power. This happen because the energy used by CPU architecture

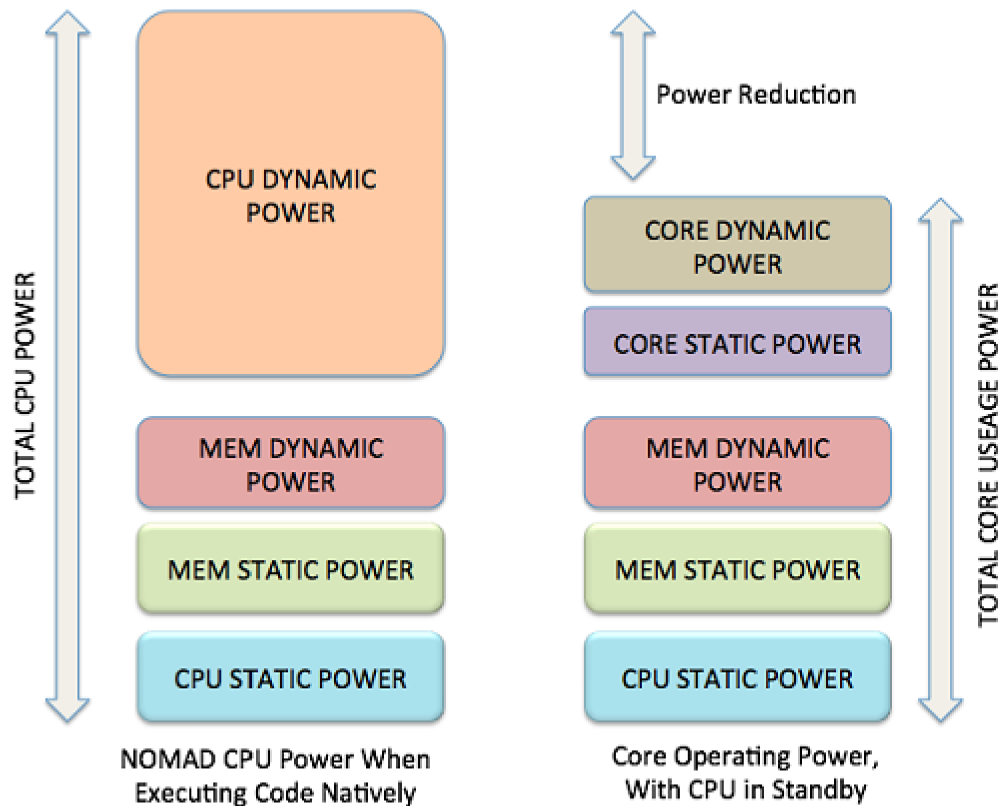


FIGURE 5.1: The figure shows the power model used to compare software vs IP core.

is so higher that cover the difference in power between Composite and Wave-core, therefore the difference becomes infinite small. This suggest that the Wave-core that improves the power density becomes the best candidate in an architecture that has as a main the improvement of Dark Silicon.

To conclude Figure 5.3 shows the differences between Composite and Wave-core architecture. It shows clearly that the Composite does very slightly better that the Wave-core.

5.3 Analysis using overage power for Lund CPU model

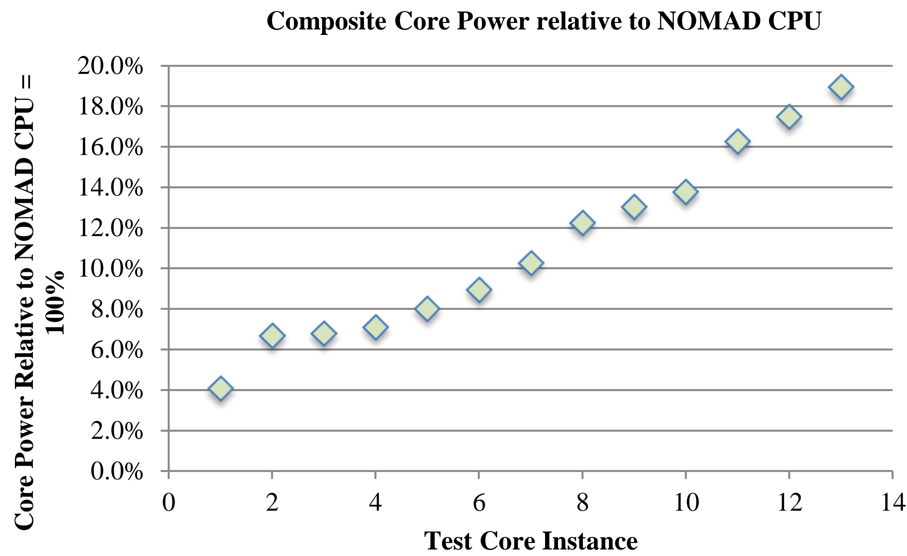
Another way to observe the benefit of the cores is to analyse the hardware IP vs Lund CPU architecture.

The Lund CPU is a stack machine architecture, therefore its operands are placed on the top of stack register and write back the results into the top of the stack register. It operates as a single issue in order completion and execute a single instruction per clock cycle.

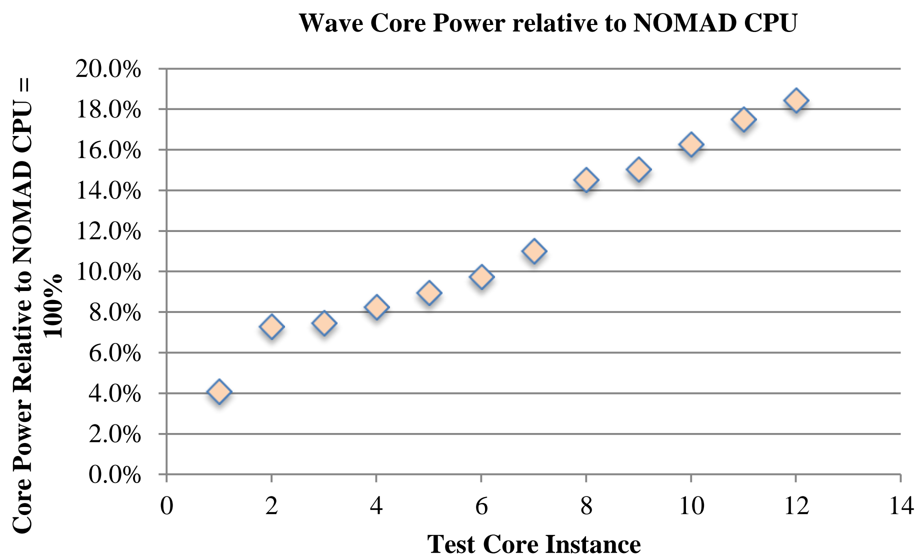
The Lund CPU was designed in 65nm CMOS technology, it covers 0.16 mm ˆ 2 and consumes about 85uW/MHz [60]

5.3.1 Methodology of comparison IP cores vs Lund CPU model

Below is showed the methodology to plot a set of comparison data to analyse the benefit in power consumption that might the cores give when compared with the



a) Core Power relative to NOMAD CPU 100% Composite Architecture



b)Core Power relative to NOMAD CPU 100% Wave-core architecture

FIGURE 5.2: Shows the core power relative in percentage for both Composite and Wave-core architecture when compared with the NOMAD CPU. It shows also that both Composite and Wave-core architecture consumes just up to 20 % of energy when compared with the CPU

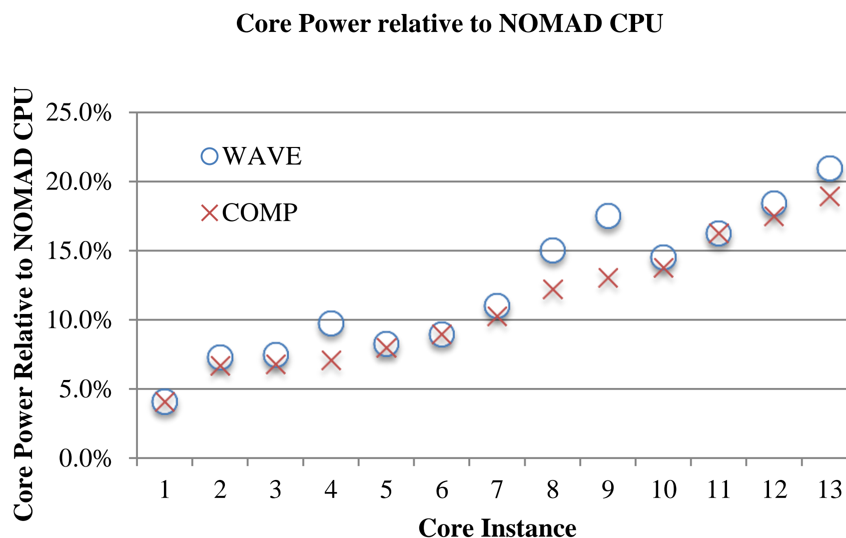


FIGURE 5.3: Core power relative to NOMAD CPU 100 % Composite vs Wave-core;

Lund CPU.

1. **Lund CPU power per instruction:**

   The Lund CPU executes one instruction per clock cycle and use 85uW/MHz. Therefore, dividing 85u/1M=85pw/clock. In this simplified model we assume each instruction requires 1.5 clock cycles on average. For this reason it consumes 1.5*85 = 127.5pW per instruction. Now all we need to do is multiply this assumed constant by the instruction count.

2. **The power consumption used by Lund CPU for each function benchmark:**

   To understand the amount of power the Lund CPU consumes when implements a single function or benchmark. At first the amount of instruction for each function or benchmark is counted and then the result is multiplied by the amount of power per instruction; in this case by 127.5p.

3. **The amount of power the IP core consumes for each benchmark:**

   To observe how much power consumes an IP core for a function benchmark. At first the total power obtained by the translator tool is divided by 100MHz to have the amount of power per clock cycle. For instance the power of an IP core is expressed in 100MHz. Therefore to know the power per instruction, the total power is divided by 100MHz.

   After that to understand how much power a function or benchmark consumes. In the first, the number of states for the selected function are counted and then multiplied for the amount of power each state consumes.

For the sake of clarity Table 5.1 shows the implemented methodology explained above. In more detail; The first column (Func. nW) shows the amount of power

TABLE 5.1: Lund CPU vs IP cores

Func. nW	St	IPC	CPU pW/Inst	CPU pW/MHz	Core pW/Inst	Core pW/MHz	Ratio
3911874.629	1	1.00	127.5	127.5	39.1	39.1	30.6 %
4582767.004	5	1.80	127.5	1147.5	45827.6	229.1	19.9 %
5199848.241	11	1.82	127.5	2552.5	51998.4	571.9	22.4 %
4393228.857	3	1.67	127.5	638.7	43932.2	131.7	20.6 %
3911874.629	1	1.00	127.5	127.5	39118.7	39.1	30.6 %
3911874.629	1	1.00	127.5	127.5	39118.7	39.1	30.6 %
3977542.869	11	4.09	127.5	5736.2	39775.4	437.5	7.6 %
3911874.629	1	1.00	127.5	127.5	39118.7	39.1	30.6 %

the selected function consumes obtained by the translator tool, then the second column (St) shows the number of states the selected function is implemented. After, the third column (IPC) shows the number of IPC of the selected function. Column fourth (CPU pW/Inst) the amount of power per instruction the CPU consumes when implements the selected function and more column fifth (CPU pW) shows the total power the CPU consumes when implements the selected function.

To observe the power consumed by core; column sixth (Core pW/Inst) shows the power consumes by core for each clock cycle, then column seventh (Core pW) shows the amount of power the IP core consumes when implements a specific function and finally column eighth (Ratio) shows the ratio between CPU vs IP core in percentage.

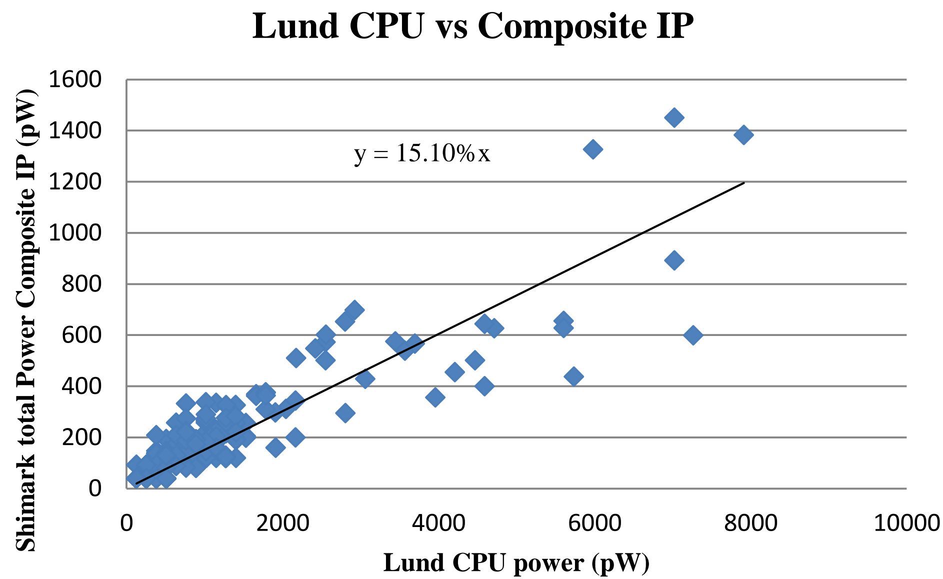
According to Table 5.1 the cores use from around 8 % to around 31 % of power compared to the CPU. For a better comparison next section compare then seven benchmark for the Shannon optimization for both Composite and Wave-core architecture vs Lund CPU architecture.

5.3.2 Comparing Composite and Wave-core architecture vs Lund CPU architecture

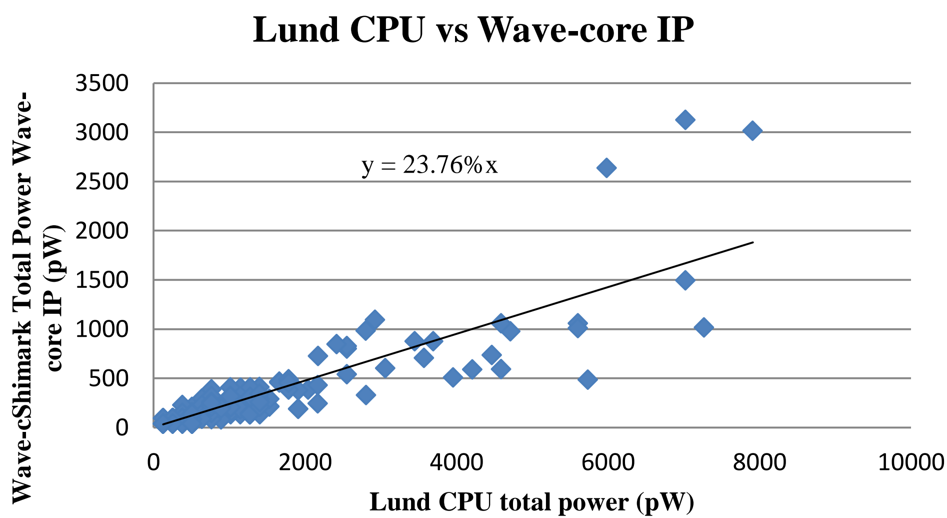
To have a better comparison between the Composite and Wave-core architecture vs Lund architecture, Shannon optimization for the seven benchmarks is used.

In brief Shannon optimization use a global scheduling optimization and has been demonstrated that increases the number of IPC. Increasing the number of IPC the translator tool generates a more efficient IP core, Therefore it appears to be the best candidate to compare with the CPU.

According to Figure 5.4 in general the cores use around 24 % of power when compared with the CPU. It means the CPU wastes more than 70 % of energy. In more details Figure 5.4 a) as expected the Composite that does better in power consumption uses just the 15 % of power. In addition the majority of cores appear to be grouped inside 400 pW while the CPU is already in the order of thousands pW. This means that the benefit in using the Composite core is higher. Despite this, the Wave-core Figure 5.4 b) uses as overage just 24 % of power when compared with the CPU and more the majority of cores seems grouped inside 500pW for the Wave-core and inside thousands pW for the CPU. It means that for these specific cores in Wave-core IP the energy consumed is less around 1000pW.



a) Lund CPU vs Composite IP power estimation



b) Lund CPU vs Wave-core IP power estimation

FIGURE 5.4: Shows the comparison between Lund CPU vs Composite and Wave-core architecture.

5.4 Analysis using average power model for NOMAD CPU

Section 5.2 explained a more accurate comparison between the NOMAD CPU vs Cores. Alternatively, this section compares the NOMAD CPU vs Cores using the NOMAD CPU model as an average behavioural model. A similar approach was undertaken in Section 5.3 (Lund CPU) It helps to compare a large number of accelerators and so it shows a broader but less accurate representation of the data.

The model referred here is similar to Lund CPU model Table 5.1, (see Section 5.4 for more detail). However, the power per instruction the NOMAD uses is higher when compared with the LUND CPU. In fact the NOMAD CPU power per instruction is 469.3pW/Inst. instead of 127.5pW/Inst. for the LUND CPU. This is because NOMAD CPU has a more complex design and instruction set.

To calculate the power per instruction in NOMAD CPU an overage of power is applying considering 1.22 instruction per clock cycle. The number of instructions per clock cycle was calculated using the NOMAD CPU' data experiment shown in Section 5.2 . In addition, to have a similar model with the LUND CPU. The power generated by memory was not included for this analysis.

Figure 5.3 a) shows the comparison NOMAD CPU vs Composite core. According to the data, the Composite architecture consumes approximately just the 3 % of energy when compared with the CPU at a target frequency of 100MHz. This suggest that in complex CPUs using accelerators instead of software function, the overall architecture becomes more power efficient. But this is excluding memory power cost.

On the other side, Figure 5.3 b) shows the comparison NOMAD CPU vs Wave-core. The Wave-core architecture that increases the power consumption uses around 5 % of energy when compared with the CPU. This implies that the best way to reduce the power in a CPU is to use a CoDA network architecture. Furthermore according with Figure 5.2 in real application both Composite and Wave-core appear to consume the same energy. Moreover, the Wave-core architecture improves the power density by around 50 %. For this reason The Wave-core architecture appear to be the best candidate to be used to generate accelerators.

5.5 Summary of power models and results

This section has been observing the comparison CPU vs accelerators. In other words the amount of power consumed by a function if is implemented at software level or at hardware level.

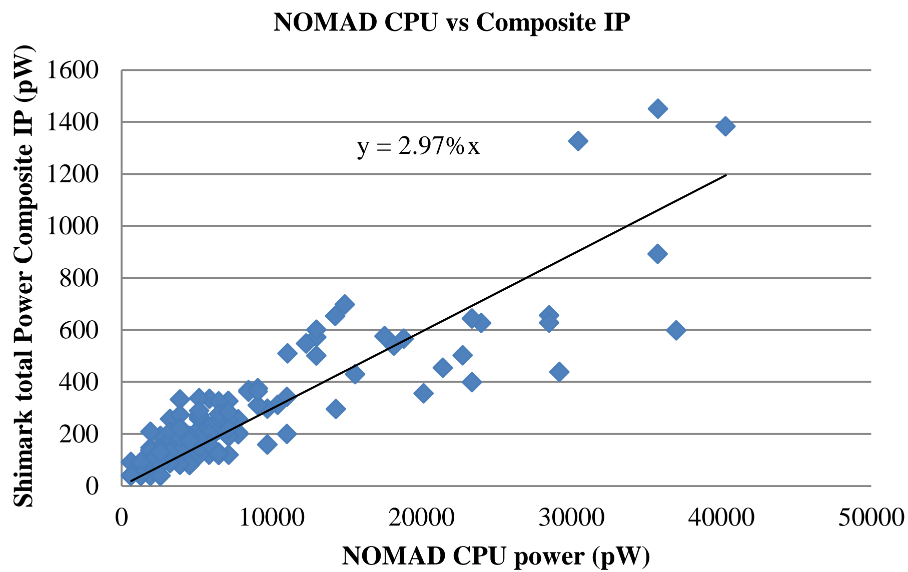
Section 5.2 uses the NOMAD CPU synthesised net-list model using the UMC 65nm CMOS node technology. It has demonstrated that the cores can improve the power an overage of 20 %. Furthermore for a few cores the energy is less up to around 90 %.

A similar result has been observed in Section 5.3. In fact Comparing the NOMAD CPU model with the Composite and Wave-core architecture. The Composite uses just approximately 3 % of power, while the Wave-core nearly 5 %.

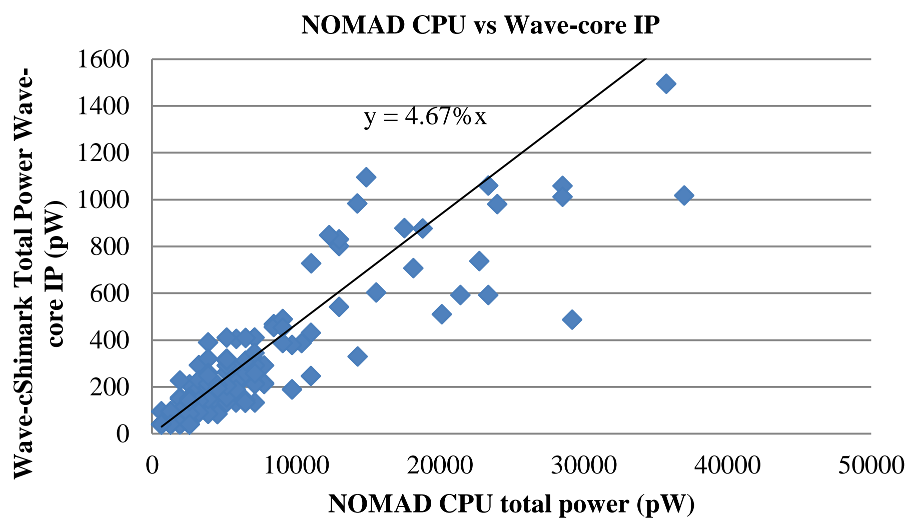
To conclude Comparing the Lund CPU vs cores Section 5.3. The results indicate that the cores consume less energy. In more details; the Composite uses just around the 15 % of energy, while the Wave-core uses approximately 25 % of energy.

To summarize, the key points are:

1. To understand the benefit of using accelerators, The NOMAD CPU synthesis net-list with memory power included is used. It gives a more accurate reading of the power for function running at software level. The results show that if



a) NOMAD CPU vs Composite IP power estimation



b) NOMAD CPU vs Composite IP power estimation

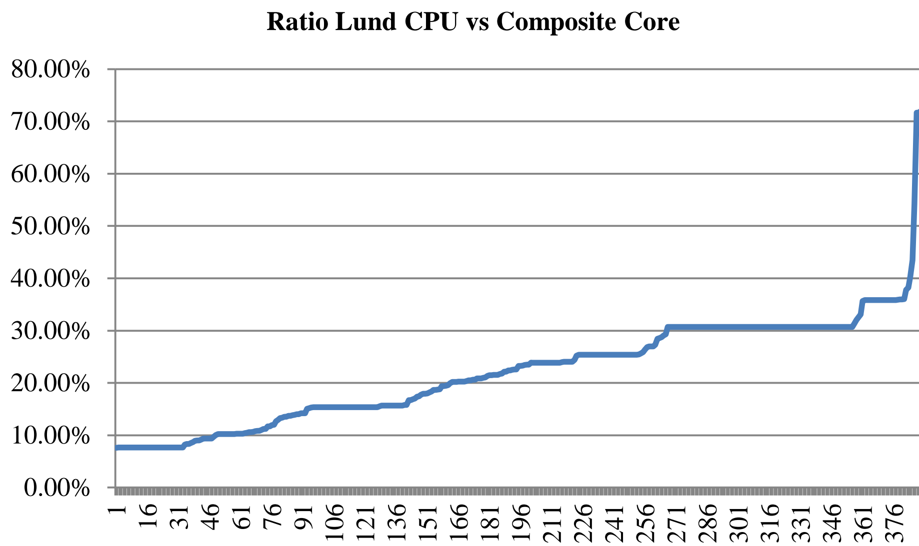
FIGURE 5.5: Shows the comparison between NOMAD CPU vs Composite and Wave-core architecture.The architectures consume less than 5 % of energy when compared with the CPU

the functions are run by hardware core. the cores consumes of the order by 5 % to 20 % less power. In other words; from 5 to 20 times saving

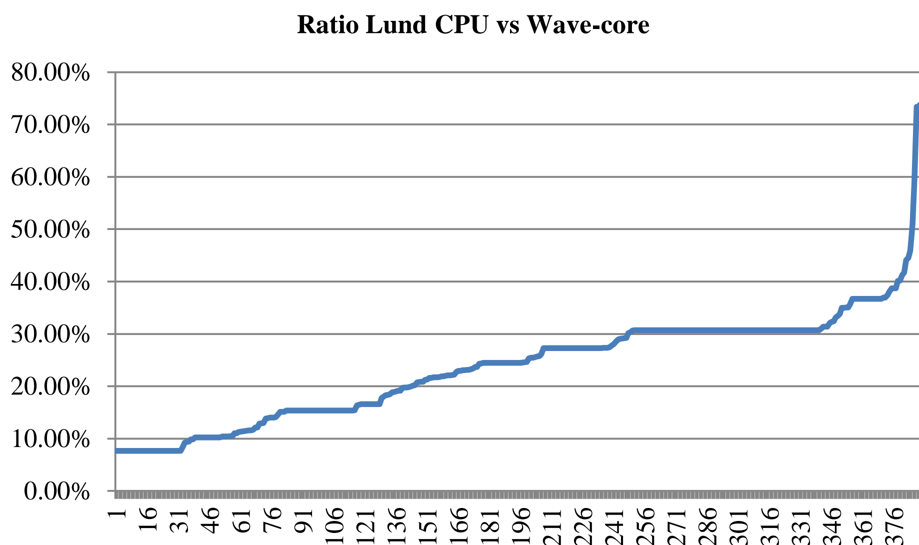
2. To compare a large number of data, the LUND model that does not include the memory power cost. According to the data Figure 5.6. The accelerators consume approximately for the majority of cores between by around 8 and 30 % of power when compared the CPU. That is equal to 3 to 12 fold saving

3. to observe how a large number of cores behave when compared with the NOMAD CPU. The NOMAD CPU model that does not include the memory power is used. According with Figure 5.6, the cores consume between by 2 % to 10 % of power when compare with the CPU. That is like to say; the cores consume between 10 to 50 time less energy than a CPU

In brief: this Chapter has demonstrated that using accelerators the CPU consumes less energy. It has also demonstrated that for complex CPU, accelerators help to dissipate more energy. In Fact observing the NOMAD CPU model the majority of cores consume between by 2 % and 10 %, while in the Lund CPU model the majority of cores are in the range between by 8 % and 30 %. Furthermore in a real application the Wave-core that consume less power density, approximately by 50 % less when compared with the Composite architecture, is the best candidate to improve the Dark Silicon of the chip.

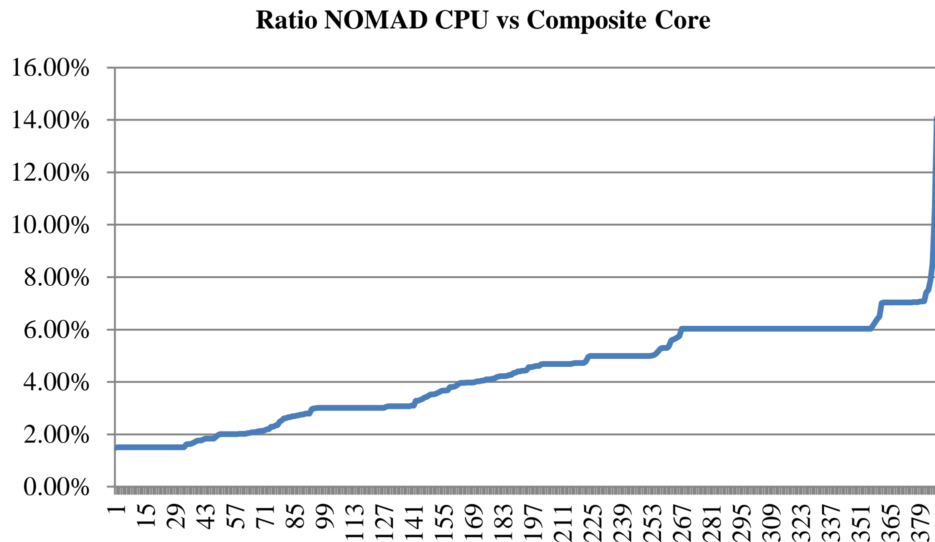


a) Ratio Lund CPU vs Composite Core

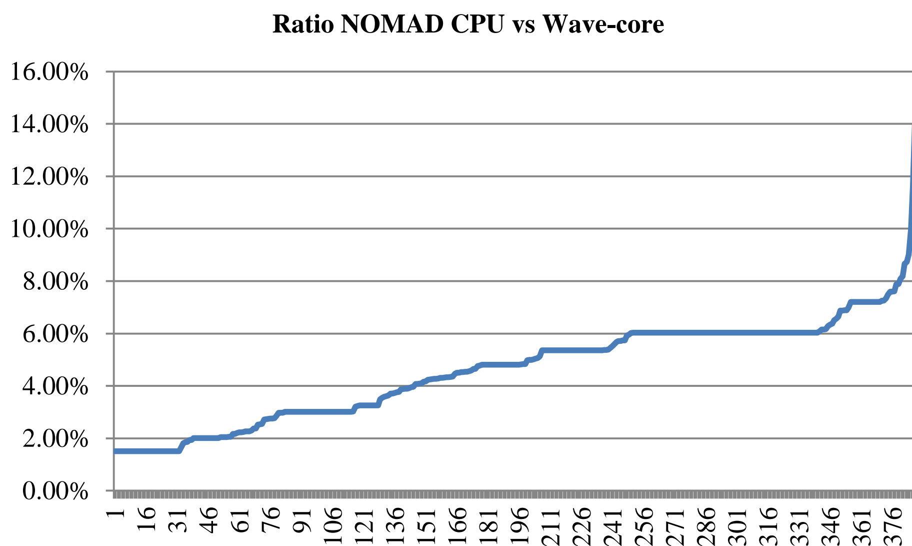


b) Ratio Lund CPU vs Wave-core

FIGURE 5.6: Shows the comparison between the Ration Lund CPU vs Composite and Wave-core architecture. The data shows the cores consume up to around 90 % less power



a) Ratio NOMAD CPU vs Composite Core



b) Ratio NOMAD CPU vs Wave-core

FIGURE 5.7: Shows the comparison between the Ration NOMAD CPU vs Composite and Wave-core architecture. The data shows for the majority cores the power consumed is between by 2 % and 10 %

Chapter 6

Conclusion

In this thesis, the key aims were to ameliorate Dark Silicon by using accelerator cores to enhance power efficiency and power density in a novel context – that of a stack-based processor host architecture. The contributions reported in section 6.1 clearly support all of the key expectations of this goal, and supports the validation of the hypothesis.

Furthermore, an automated Accelerator generation for a Dark Silicon Design Space has been introduced. It has been proved that by using IP cores instead of software functions the CPU can improve the power consumption by up to 20 %. The effective performance of Wave-core, which reduces the power density by up to 50 % when is compared with the Composite has also been proved. The Wave-core also has been demonstrated to be the best candidate when is interfaced to a CPU, because the difference in power consumption between Composite and Wave-core is already quite small, and when the whole power model is considered (for example, memory and static CPU stand by power), these differences become even smaller as part of the overall power cost.

In conclusion this work has shown that applying accelerator cores to a Stack Machine architecture generates a more energy efficient architecture. This accord with the theory that describes the stack architecture as a friendly method of power consumption. It has also been demonstrated that there is no reason to restrict such heterogeneous core models to mainstream register-based processor models. In fact, it is possible to extend this concept into novel architectures such as stack-machines, while exploiting valuable code optimisation approaches that are only available in those architectural models.

6.1 Main Contributions and Implications

The main expectations of the present hypothesis were as follows:

1. Demonstrate that a translation of stack machine code, software function, into cores, hardware units, is feasible.

2. Evaluate how cores perform in terms of power, area and timing when comparing the classical implementation of a state machine with a novel implementation of state machine architecture core design.

3. Observe whether advanced code optimization can contribute to better core performance in terms of power, area and timing.

4. Compare CPU versus cores to understand what performance the cores give.

These expectations have clearly fulfilled in the results presented. The following is a summary of the contributions as they relate to the quoted hypothesis.

1. **Demonstrate that a translation of stack machine code, software function, into cores, hardware units, is feasible.**

   This Thesis has presented a novel fully automated translator tool, which performed software to hardware translation on thousands of software test cores. This demonstrated that translation of assembly stack code into hardware IP (accelerators) was entirely feasible. (See Chapter 2).

   Furthermore, starting from the intermediate code optimization using a stack compiler, the translator tool is able in a few steps to split the software function into many stages and associate the sequence of software instructions into VHDL. Consequently, translating software functions into hardware IP cores was made considerably easier. (See section 2.5).

   In addition, the translator tool incorporates a well known algorithm, known as backtracking. This facilitates the extraction of inputs and outputs operand counts, an essential requirement where stack code translation is performed. (See section 2.5.1).

   Likewise, the translator tool generates a csv file to count the number of inputs/outputs the cores required, (see Chapter 2.5). According to the data the numbers of inputs/outputs the cores use are quite low, around 4, (see Chapters 3.1, 3.2, 4.1 and 4.2). This suggests that the amount of operand traffic sent to and from cores can be managed efficiently with one or two operand buses. This makes the interconnection core memory very simple. In addition a small number of operands need small stack memory interface. As a result little energy is consumed.

   The translator tool also records the numbers of states and IPC in the csv file. This is very important information for understanding the structure of the hardware cores (see Chapter 3.3, 3.4, 4.3 and 4.4). For example, observing how compiler scheduling techniques optimized the code used by the cores, helps to decrease the number of states and increase the amount of parallelism in the core architecture. All of these improve power consumption.

2. **Evaluate how cores perform in terms of power, area and timing when comparing the classical implementation of a state machine with a novel implementation of state machine architecture core design.**

   Two core structure alternatives were implemented: Composite that uses a classical state machine implementation architecture and a novel implementation of the state machine, the Wave-core based on synchronous sequential stage-per-stage systems. (See Chapters 2.7 and 2.8)

   Composite uses a shared logic design to perform a sequential logic states, it optimises the logic per area

   In contrast Wave-core architecture, which has the goal of improving the power density, treats each state as an independent hardware structure. Each state receives the data from the memory performed by the previous states, and stores the results in the memory to be passed to the next state, (see Chapter 2.8.1).

   The main advantages in implementing the Wave-core architecture are; Wave-core increases the area per logic and so improves the power density. In Fact,

comparing the Composite vs Wave-core. Wave-core improves the power density around 50 % (see Chapter 3.10). Therefore, this last appears to be the best candidate to improve the power efficiency of the CPU and the Dark Silicon area(see Chapter 5.2.1).

Furthermore, the multi-stage in Wave-core architecture makes it possible to simply isolate each state and applying the power gating technique improves the static power.

For instance, Chapter 2.5 explains how the generic finite state machine is implement to convert SW/HW. In particular the state machine increments each state at each swap of information from and to the memory.

Indeed, the Wave-core and its dedicated states, (see Chapter 2.8.1) enables to hibernate and have fully control of states without losing or stopping the execution process. On the other hand, the Composite architecture using a share logic design each state can not be hibernate, in other words the power gating cannot be applied because it loses the state and stop the process of execution.

In more detail and highlighting the results:

Wave-core architecture was observed to increase the area and so the leakage around by 2x when compared with Composite core style. Chapters 3.5 and 3.7.

The Dynamic Power for Wave-cores was slightly higher than Composite by about 12 %. (See Chapter 3.8 and 5.2.1).

The timing for both Wave-core and Composite were found to be very similar. It follows that there is no cost penalty in terms of timing as far as core style is concerned. That is to say: it is true that the Wave-core increases wired interconnections between stages, but because the cores run around 1GHz and a 65nm node technology can run much faster, the delay between the transistor and the wires can not be observed.

Indeed, core timings tended toward the 1GHZ frequency range +/- 200 MHz, which is comparable to processors in the 65nm process domain. (See Chapters 3.6)

3. **Observe whether advanced code optimization can contribute to better core performance in terms of power, area and timing.**

   Stack Scheduling was found to have a significant impact on the number of states in a core and on the parallelism (instructions per state). Stack scheduling results in fewer execution states and higher data-flow parallelism in each core generated. (See Chapter 4)

   It was also observed that the numbers of inputs/outputs for a typical core increases after the optimization. For example, Shannon has the higher number of Inputs/Outputs. In detail; Chapter 4.1 and 4.2 show that Shannon has a more than 5 % of cores with >5 I/O. Similarity with IPC, Shannon increases the IPC up to 11 (see Chapter 4.4.3). Conversely, increasing the numbers of IPC decreases the numbers of states. Shannon has the majority of cores between 1 and 3 states. (See Chapter 4.3.3).

   In more detail and highlighting the results obtained by Wave-core architecture for the Shannon optimization that has the higher number of IPC

   Observations relating to the use of stack code optimisation, and particularly the Shannon optimisations shows some clear outcome that are worth noting:

In terms of area cost, it is seen that Shannon (global) code optimisation causes changes in area cost for both core types. For Wave-Core a reduction of about 30 % in area is typically observed. However, for composite cores the area cost increases, and is typically of the order of 20 % larger.

Regarding timing behaviour, both core types generally benefit from a small improvement in critical path delay. However these gains are quite small (of the order of 1 to 3 %). Another way to look at this result is that the cores do not become slower after applying the Shannon optimisation, in other words, the Shannon optimisation can be applied without any timing penalty.

A significant effect of using stack scheduling is the effect upon IPC per core. This is very clear in the graph of Figure 4.13 in earlier (Chapter 4). It is shown that a large majority of cores achieve increases of IPC of the order of 20 % to 50 %. This is easily explained by the fact that scheduling eliminates unnecessary memory references, and this results in the number of states being reduced. Remember that states are defined by sequences of instructions bounded by memory operations. If there are less states but the same overall work, the work per state (IPC) increases. This is confirmed by Figure 4.10 (Chapter 4) where it is clear that the number of cores with fewer states is increased significantly after Shannon optimisation.

For power behaviour, the Shannon optimisation causes Composite cores to consume more static power, around 25 % (similar to the area increase), but the wave core reduces static power by around 30 %, again in line with area cost. This suggests that extra components, and thus extra area, consume more static power, which is what would be expected. However the absolute power consumption, after optimisation, still indicates that Composite cores consume less static power overall, tending toward about half the power for Composite cores compared to Wave-cores.

For dynamic power, the effect of Shannon optimisation is to reduce dynamic power overall for both core types, with Composite cores seeing a bigger gain than Wave-cores overall. Taking this into account for power density (power and area effects together), the results presented in (Section 4.10) and particularly Figure 4.38 indicates that power density increases by about 10-12 % for Wave-core using Shannon optimisation, versus the baseline Wave-core case.

What is interesting then, is that applying Shannon optimisation increases power density by a small but noticeable amount, but on the positive side there is significant increase in IPC, no reduction in path delay, and reduced number of states per core (and thus less clock cycles needed to complete a given core operation). This suggests that a small increase in power density for the highly optimised version of the cores (Wave-core and Shannon) is worthwhile to deliver the range of other performance benefits noted above.

4. **Compare CPU versus cores to understand what performance the cores give.**

   To understand the benefit that the cores can give to the whole CPU architecture in term of energy, Chapter 5 investigated on the power consumed by functions when run at software and at hardware level.

   to have a better overview three types of experiment were made.

The first and most accurate but just for a few cores were using the NOMAD CPU synthesis net-list. According to the data the cores were able to use between 4 % and 20 % of power compared with the CPU. Indeed, in this experiment the memory power was considered. (See Section 5.2)

The second and less accurate experiment was to use the Lund CPU model, The Lund CPU model does not include memory power. That is to say; the advantage of using the Lund CPU model is that a large number of cores could be easily synthesised. According to the data the cores consume an average of 15 % of energy when compared to the CPU for the Composite and approximately 20 % of energy when compared to the CPU for the Wave-core architecture. (See Section 5.3)

Finally, to analyses how a complex architecture behaves for a large number of benchmarks the NOMAD CPU model without memory power was used. According to the data, the accelerator consumes just around 3 % of energy when compared to CPU with the Composite architecture and just nearly 4 % when compared with a CPU with the Wave-core architecture.(See Section 5.4)

To conclude Chapter 5 has demonstrated that to have more power efficiently CPU, new architectures must include accelerators. In fact they use a very small amount of energy when compared to CPU, between 2 % to 10 %.

In addition to alleviate Dark Silicon problem the Wave-core architecture is the best candidate to implement accelerators. It has been demonstrated that the power consumption is almost identical to the Composite architecture when interfaced to a real CPU. This implies that the CPU static power and the memory dynamic and static power absorbs the small difference that the Wave-core has when compared with the Composite. Furthermore, the Wave-core has been demonstrated to decrease the power density by about 50 %, therefore, it is the best candidate to reduce the Dark Silicon of the chip.

6.2 Future Work

This thesis provides a solid groundwork for further investigation into a CoDA architecture based on a Stack Machine in order to understand the concept more deeply.

First of all, a CoDAs platform based on a Stack Machine architecture should be developed , which sees a Stack Machine silicon chip surrounded by many FPGAs to test the cores.

Building a physical platform will give the opportunity to generate empirical results. It gives the chance to test and compare real applications with a classical CPU architecture, but also opens new areas of research, starting from compiler and optimisation technique and finishing with hardware performance.

The CoDAs processor based on a Stack Machine could be designed by first interconnecting one core and then adding "n" cores that will operate intermittently.

The aim of this thesis is to compare the SW/HW, therefore it could be the first observation of how the cores behave for real applications when both Composite and Wave-core architectures are compared with the software function.

Selection of cores based on individual performance.

This work has introduced the characteristic of cores and the opportunity to match them as more appropriate to improve the energy efficiency of the device. Selecting a real application, extracting the most power-wasteful function and implementing it

at hardware level we can observe how the cores behave and produce new algorithms with which we can expresses the relationship of size/time, size/power, power/time etc.

Further input code enhancement.

The Scheduling techniques have a significant impact on core generation; therefore the results suggest investigating further and coming out with new algorithms to extract more IPC at software level. For example, re-ordering the instruction sequences may have no benefit when code is executed on a CPU, but when translated to a hardware core we have seen cases where this might allow more efficient hardware with few core states.

Composition of cores into "meta-cores".

Observing the number of inputs and outputs has shown that many cases can be generated to operate in isolation. However, many of these cores are related to each other through the original progress structure. Therefore it would be possible to investigate methods to cluster multiple cores together to form a "meta-core" where the cores would exchange operands core to core and only communicate with the CPU at the start and end of this behaviour. This would allow structures such as loops to be translated efficiently into hardware cores.

Impact of target frequency.

Evaluate power cost across a range of target frequencies. At higher clock rates, the static memory and the static CPU power components will reduce compared to the core power. These characteristics should change in proportion to the clock rate.

Appendix A

VHDL Code- Composite and Wave-core

```
//Composite architecture VHDL:
LIBRARY ieee;
USE ieee.std_logic_1164.all;
USE ieee.numeric_std.all;
USE ieee.std_logic_unsigned.all;

ENTITY darc1 IS
PORT (
i00 : IN std_logic_vector(31 DOWNTO 0);
i01 : IN std_logic_vector(31 DOWNTO 0);
i02 : IN std_logic_vector(31 DOWNTO 0);
i03 : IN std_logic_vector(31 DOWNTO 0);

r00 : OUT std_logic_vector(31 DOWNTO 0);
r01 : OUT std_logic_vector(31 DOWNTO 0);

FP : IN std_logic_vector(31 DOWNTO 0);
FPout : OUT std_logic_vector(31 DOWNTO 0);
M_ADDR :   OUT std_logic_vector(31 DOWNTO 0);
M_DATA : INOUT std_logic_vector(31 DOWNTO 0);
M_RD  : INOUT std_logic;
M_WR  : INOUT std_logic;
M_RDY : IN std_logic;
reset : IN std_logic;
CLK  : IN std_logic
);
END ENTITY;

ARCHITECTURE yy OF darc1 IS
TYPE States IS (ST_INIT,WS_INIT,ST_RESET,ST00,WS00,ST01,WS01,
ST02,WS02,ST03,WS03,ST_END);
SIGNAL Mstate : States;
BEGIN


-- CONTROL PROCESS --------
PROCESS(clk,reset)
```

```
BEGIN
IF reset='1' THEN
Mstate <= ST_RESET;
ELSIF(rising_edge(clk)) THEN
CASE Mstate IS
WHEN ST_RESET => Mstate <= ST_INIT;
WHEN ST_INIT => IF M_RDY='1' THEN Mstate <= ST00;

ELSE Mstate <= WS_INIT; END IF;

WHEN WS_INIT => IF M_RDY='1' THEN Mstate <= ST00; END IF;

WHEN ST00 =>  IF M_RDY='1' THEN Mstate <= ST01;

ELSE Mstate <= WS00; END IF;

WHEN WS00 =>  IF M_RDY='1' THEN Mstate <= ST01; END IF;

WHEN ST01 =>  IF M_RDY='1' THEN Mstate <= ST02;

ELSE Mstate <= WS01; END IF;

WHEN WS01 =>  IF M_RDY='1' THEN Mstate <= ST02; END IF;

WHEN ST02 =>  IF M_RDY='1' THEN Mstate <= ST03;

ELSE Mstate <= WS02; END IF;

WHEN WS02 =>  IF M_RDY='1' THEN Mstate <= ST03; END IF;

WHEN ST03 | WS03| ST_END =>  WHEN OTHERS =>

END CASE;
END IF;
END PROCESS;


-- EXECUTE COMP PROCESS --------
PROCESS(clk,reset)
VARIABLE T,s0,s1,s2,s3,s4,s5,s6,s7,fpi
:std_logic_vector(31 DOWNTO 0);

BEGIN
IF(reset='1') THEN
-- reset any internal states --

s0 := (OTHERS=>'0');
s1 := (OTHERS=>'0');
s2 := (OTHERS=>'0');
s3 := (OTHERS=>'0');
s4 := (OTHERS=>'0');
```

```
s5 := (OTHERS=>'0');
s6 := (OTHERS=>'0');
s7 := (OTHERS=>'0');
fpi:=(OTHERS=>'0');   -- ref E --
M_ADDR <= (OTHERS=>'Z');
M_DATA <= (OTHERS=>'Z');
M_RD <= 'Z';
M_WR <= 'Z';
r00 <=(OTHERS=>'0');
r01 <=(OTHERS=>'0');

ELSIF(rising_edge(clk)) THEN
M_DATA <="ZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZ";
CASE Mstate IS
WHEN ST_INIT =>
-- COMP. connect 4 input params here --
s0 := i00;
s1 := i01;
s2 := i02;
s3 := i03;
fpi := FP;
--lit ;
s7 := s6;
s6 := s5;
s5 := s4;
s4 := s3;
s3 := s2;
s2 := s1;
s1 := s0;
s0:= std_logic_vector(to_unsigned(24, 32)); --fp- ;
fpi:=fpi+s0 ;
s0 := s1;
s1 := s2;
s2 := s3;
s3 := s4;
s4 := s5;
s5 := s6;
s6 := s7;
WHEN ST00 =>
--copy1 ;
s7 := s6;
s6 := s5;
s5 := s4;
s4 := s3;
s3 := s2;
s2 := s1;
s1 := s0;
WHEN ST01 =>
WHEN ST02 =>
--copy1 ;
s7 := s6;
```

```
s6 := s5;
s5 := s4;
s4 := s3;
s3 := s2;
s2 := s1;
s1 := s0;
--lit ;
s7 := s6;
s6 := s5;
s5 := s4;
s4 := s3;
s3 := s2;
s2 := s1;
s1 := s0;
s0:= std_logic_vector(to_unsigned(7, 32)); -- basr NOT IMPLEMENTED ;
--lit ;
s7 := s6;
s6 := s5;
s5 := s4;
s4 := s3;
s3 := s2;
s2 := s1;
s1 := s0;
s0:= std_logic_vector(to_unsigned(1, 32)); --and ;
s0:=s0 AND s1 ;
s1 := s2;
s2 := s3;
s3 := s4;
s4 := s5;
s5 := s6;
s6 := s7;
--ldi ;
s7 := s6;
s6 := s5;
s5 := s4;
s4 := s3;
s3 := s2;
s2 := s1;
s1 := s0;
s0:= std_logic_vector(to_unsigned(255, 32)); --and ;
s0:=s0 AND s1 ;
s1 := s2;
s2 := s3;
s3 := s4;
s4 := s5;
s5 := s6;
s6 := s7;
-- bzp ### ignored ### ;
-- recover 2 results here --
r00 <= s0;
r01 <= s1;
```

```
FPout <= fpi;

WHEN OTHERS => s0 := s0;

END CASE;
END IF;

END PROCESS;

END ARCHITECTURE;

*******************************************************************************

//Wave-core Architecture VHDL:
LIBRARY ieee;
USE ieee.std_logic_1164.all;
USE ieee.numeric_std.all;
USE ieee.std_logic_unsigned.all;

ENTITY darc1 IS
PORT (
i00 : IN std_logic_vector(31 DOWNTO 0);
i01 : IN std_logic_vector(31 DOWNTO 0);
i02 : IN std_logic_vector(31 DOWNTO 0);
i03 : IN std_logic_vector(31 DOWNTO 0);

r00 : OUT std_logic_vector(31 DOWNTO 0);
r01 : OUT std_logic_vector(31 DOWNTO 0);

FP : IN std_logic_vector(31 DOWNTO 0);
FPout : OUT std_logic_vector(31 DOWNTO 0);
M_ADDR :   OUT std_logic_vector(31 DOWNTO 0);
M_DATA : INOUT std_logic_vector(31 DOWNTO 0);
M_RD  : INOUT std_logic;
M_WR  : INOUT std_logic;
M_RDY : IN std_logic;
reset : IN std_logic;
CLK  : IN std_logic
);
END ENTITY;

ARCHITECTURE yy OF darc1 IS
TYPE States IS (ST_INIT,WS_INIT,ST_RESET,ST00,WS00,ST01,WS01,
ST02,WS02,ST03,WS03,ST_END);
SIGNAL Mstate : States;
SIGNAL u0s0, u0s1, u0s2, u0s3, u0s4, u0s5, u0s6,
u0s7, FPi0  : STD_LOGIC_VECTOR(31 DOWNTO 0);

SIGNAL u1s0, u1s1, u1s2, u1s3, u1s4, u1s5, u1s6,
u1s7, FPi1  : STD_LOGIC_VECTOR(31 DOWNTO 0);
```

```
SIGNAL u2s0, u2s1, u2s2, u2s3, u2s4, u2s5, u2s6,
u2s7, FPi2  : STD_LOGIC_VECTOR(31 DOWNTO 0);

BEGIN


-- CONTROL PROCESS --------
PROCESS(clk,reset)
BEGIN
IF reset='1' THEN
Mstate <= ST_RESET;
ELSIF(rising_edge(clk)) THEN
CASE Mstate IS
WHEN ST_RESET => Mstate <= ST_INIT;
WHEN ST_INIT => IF M_RDY='1' THEN Mstate <= ST00; ELSE Mstate <= WS_INIT; E

WHEN WS_INIT => IF M_RDY='1' THEN Mstate <= ST00; END IF;

WHEN ST00 =>  IF M_RDY='1' THEN Mstate <= ST01;

ELSE Mstate <= WS00; END IF;

WHEN WS00 =>  IF M_RDY='1' THEN Mstate <= ST01; END IF;

WHEN ST01 =>  IF M_RDY='1' THEN Mstate <= ST02;

ELSE Mstate <= WS01; END IF;

WHEN WS01 =>  IF M_RDY='1' THEN Mstate <= ST02; END IF;

WHEN ST02 =>  IF M_RDY='1' THEN Mstate <= ST03;

ELSE Mstate <= WS02; END IF;

WHEN WS02 =>  IF M_RDY='1' THEN Mstate <= ST03; END IF;

WHEN ST03 | WS03| ST_END =>  WHEN OTHERS =>

END CASE;
END IF;
END PROCESS;


-- WAVE EXECUTE PROCESS --------

proc_P00  :  PROCESS(clk,reset)
VARIABLE T,s0,s1,s2,s3,s4,s5,s6,s7, fpi :std_logic_vector(31 DOWNTO 0);
BEGIN
IF(reset='1') THEN

-- reset any internal states --
```

```
fpi :=(OTHERS=>'0');
FPi0 <=(OTHERS=>'0');  -- ref A --
u0s0 <= (OTHERS=>'0');
u0s1 <= (OTHERS=>'0');
u0s2 <= (OTHERS=>'0');
u0s3 <= (OTHERS=>'0');
u0s4 <= (OTHERS=>'0');
u0s5 <= (OTHERS=>'0');
u0s6 <= (OTHERS=>'0');
u0s7 <= (OTHERS=>'0');
s0 := (OTHERS=>'0');
s1 := (OTHERS=>'0');
s2 := (OTHERS=>'0');
s3 := (OTHERS=>'0');
s4 := (OTHERS=>'0');
s5 := (OTHERS=>'0');
s6 := (OTHERS=>'0');
s7 := (OTHERS=>'0');
M_ADDR <= (OTHERS=>'Z');
M_DATA <= (OTHERS=>'Z');
M_RD <= 'Z';
M_WR <= 'Z';


ELSIF(rising_edge(clk)) THEN

IF(Mstate=ST_INIT) THEN

-- WAVE. connect 4 input params here --
s0 := i00;
s1 := i01;
s2 := i02;
s3 := i03;
fpi  := FP;
--lit ;
s7 := s6;
s6 := s5;
s5 := s4;
s4 := s3;
s3 := s2;
s2 := s1;
s1 := s0;
s0:= std_logic_vector(to_unsigned(24, 32)); --fp- ;
fpi:=fpi+s0 ;
s0 := s1;
s1 := s2;
s2 := s3;
s3 := s4;
s4 := s5;
s5 := s6;
```

```
s6 := s7;
-- generate output signals --

u0s0 <= s0;
u0s1 <= s1;
u0s2 <= s2;
u0s3 <= s3;
u0s4 <= s4;
u0s5 <= s5;
u0s6 <= s6;
u0s7 <= s7;
FPi0  <= fpi;

ELSE

M_ADDR <= (OTHERS=>'Z');
M_DATA <= (OTHERS=>'Z');
M_RD <= 'Z';
M_WR <= 'Z';



END IF;

END IF;


END PROCESS; -- proc 0
-----------------------------------------------
proc_P01  :  PROCESS(clk,reset)
VARIABLE T,s0,s1,s2,s3,s4,s5,s6,s7, fpi :std_logic_vector(31 DOWNTO 0);
BEGIN
IF(reset='1') THEN

-- reset any internal states --

-- #### States 3, s = 0

u1s0 <= (OTHERS=>'0');
u1s1 <= (OTHERS=>'0');
u1s2 <= (OTHERS=>'0');
u1s3 <= (OTHERS=>'0');
u1s4 <= (OTHERS=>'0');
u1s5 <= (OTHERS=>'0');
u1s6 <= (OTHERS=>'0');
u1s7 <= (OTHERS=>'0');
fpi :=(OTHERS=>'0');   -- ref C --
M_ADDR <= (OTHERS=>'Z');
M_DATA <= (OTHERS=>'Z');
M_RD <= 'Z';
M_WR <= 'Z';
```

```
ELSIF(rising_edge(clk)) THEN

IF(Mstate= ST00) THEN
-- get input signals --

s0 := u0s0;
s1 := u0s1;
s2 := u0s2;
s3 := u0s3;
s4 := u0s4;
s5 := u0s5;
s6 := u0s6;
s7 := u0s7;
fpi  := FPi0;   -- ref D --


-- main dataflow sequence ----

--copy1 ;
s7 := s6;
s6 := s5;
s5 := s4;
s4 := s3;
s3 := s2;
s2 := s1;
s1 := s0;
-- generate output signals --

u1s0 <= s0;
u1s1 <= s1;
u1s2 <= s2;
u1s3 <= s3;
u1s4 <= s4;
u1s5 <= s5;
u1s6 <= s6;
u1s7 <= s7;
FPi1  <= fpi;

ELSE

M_ADDR <= (OTHERS=>'Z');
M_DATA <= (OTHERS=>'Z');
M_RD <= 'Z';
M_WR <= 'Z';



END IF;
```

```
END IF;


END PROCESS; -- proc 1
-----------------------------------------------
proc_P02  :  PROCESS(clk,reset)
VARIABLE T,s0,s1,s2,s3,s4,s5,s6,s7, fpi :std_logic_vector(31 DOWNTO 0);
BEGIN
IF(reset='1') THEN

-- reset any internal states --

-- #### States 3, s = 1

u2s0 <= (OTHERS=>'0');
u2s1 <= (OTHERS=>'0');
u2s2 <= (OTHERS=>'0');
u2s3 <= (OTHERS=>'0');
u2s4 <= (OTHERS=>'0');
u2s5 <= (OTHERS=>'0');
u2s6 <= (OTHERS=>'0');
u2s7 <= (OTHERS=>'0');
fpi :=(OTHERS=>'0');   -- ref C --
M_ADDR <= (OTHERS=>'Z');
M_DATA <= (OTHERS=>'Z');
M_RD <= 'Z';
M_WR <= 'Z';


ELSIF(rising_edge(clk)) THEN

IF(Mstate= ST01) THEN
-- get input signals --

s0 := u1s0;
s1 := u1s1;
s2 := u1s2;
s3 := u1s3;
s4 := u1s4;
s5 := u1s5;
s6 := u1s6;
s7 := u1s7;
fpi  := FPi1;   -- ref D --


-- main dataflow sequence ----

-- generate output signals --

u2s0 <= s0;
u2s1 <= s1;
```

```
u2s2 <= s2;
u2s3 <= s3;
u2s4 <= s4;
u2s5 <= s5;
u2s6 <= s6;
u2s7 <= s7;
FPi2  <= fpi;

ELSE

M_ADDR <= (OTHERS=>'Z');
M_DATA <= (OTHERS=>'Z');
M_RD <= 'Z';
M_WR <= 'Z';



END IF;

END IF;


END PROCESS; -- proc 2
----------------------------------------------
proc_P03  :  PROCESS(clk,reset)
VARIABLE T,s0,s1,s2,s3,s4,s5,s6,s7, fpi :std_logic_vector(31 DOWNTO 0);
BEGIN
IF(reset='1') THEN

-- reset any internal states --

-- #### States 3, s = 2

r00 <=(OTHERS=>'0');
r01 <=(OTHERS=>'0');
fpi :=(OTHERS=>'0');   -- ref C --
M_ADDR <= (OTHERS=>'Z');
M_DATA <= (OTHERS=>'Z');
M_RD <= 'Z';
M_WR <= 'Z';


ELSIF(rising_edge(clk)) THEN

IF(Mstate= ST02) THEN
-- get input signals --

s0 := u2s0;
s1 := u2s1;
s2 := u2s2;
s3 := u2s3;
```

```
s4 := u2s4;
s5 := u2s5;
s6 := u2s6;
s7 := u2s7;
fpi  := FPi2;   -- ref D --


-- main dataflow sequence ----

--copy1 ;
s7 := s6;
s6 := s5;
s5 := s4;
s4 := s3;
s3 := s2;
s2 := s1;
s1 := s0;
--lit ;
s7 := s6;
s6 := s5;
s5 := s4;
s4 := s3;
s3 := s2;
s2 := s1;
s1 := s0;
s0:= std_logic_vector(to_unsigned(7, 32)); -- basr NOT IMPLEMENTED ;
--lit ;
s7 := s6;
s6 := s5;
s5 := s4;
s4 := s3;
s3 := s2;
s2 := s1;
s1 := s0;
s0:= std_logic_vector(to_unsigned(1, 32)); --and ;
s0:=s0 AND s1 ;
s1 := s2;
s2 := s3;
s3 := s4;
s4 := s5;
s5 := s6;
s6 := s7;
--ldi ;
s7 := s6;
s6 := s5;
s5 := s4;
s4 := s3;
s3 := s2;
s2 := s1;
s1 := s0;
s0:= std_logic_vector(to_unsigned(255, 32)); --and ;
```

```
s0:=s0 AND s1 ;
s1 := s2;
s2 := s3;
s3 := s4;
s4 := s5;
s5 := s6;
s6 := s7;
-- bzp ### ignored ### ;


-- WAVE. generate final output signals --

r00 <= s0;
r01 <= s1;
FPout <= fpi;

END IF;

END IF;


END PROCESS; -- proc 3
------------------------------------------------

END ARCHITECTURE;
```

Appendix B

Hamming distance and Verilog test-bench

This section describes Hamming distance to understand better how the test cases were selected. After that, an example of test case sequence is showed. Finally an example of Verilog Test bench is written.

Hamming distance is applied because it changes one bit per time therefore it simulates a variation of switching. Figure B.1 shows a simple example.

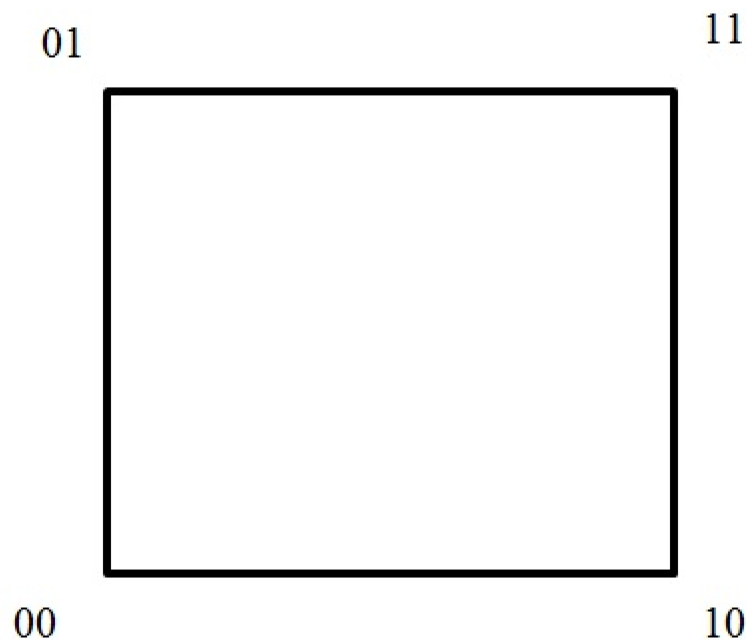


FIGURE B.1: Hamming distance double ripetition.

Figure B.1 shows at each corner the bit change of one value. However as we explained in chapter 2 it needs a quadruple repetition, therefore it builds a 4D cube Figure B.2

The figure above shows as it selects the most switching sequence. In the following page there is a test case sequence and an example of Verilog testbench.

Test cases sequence :

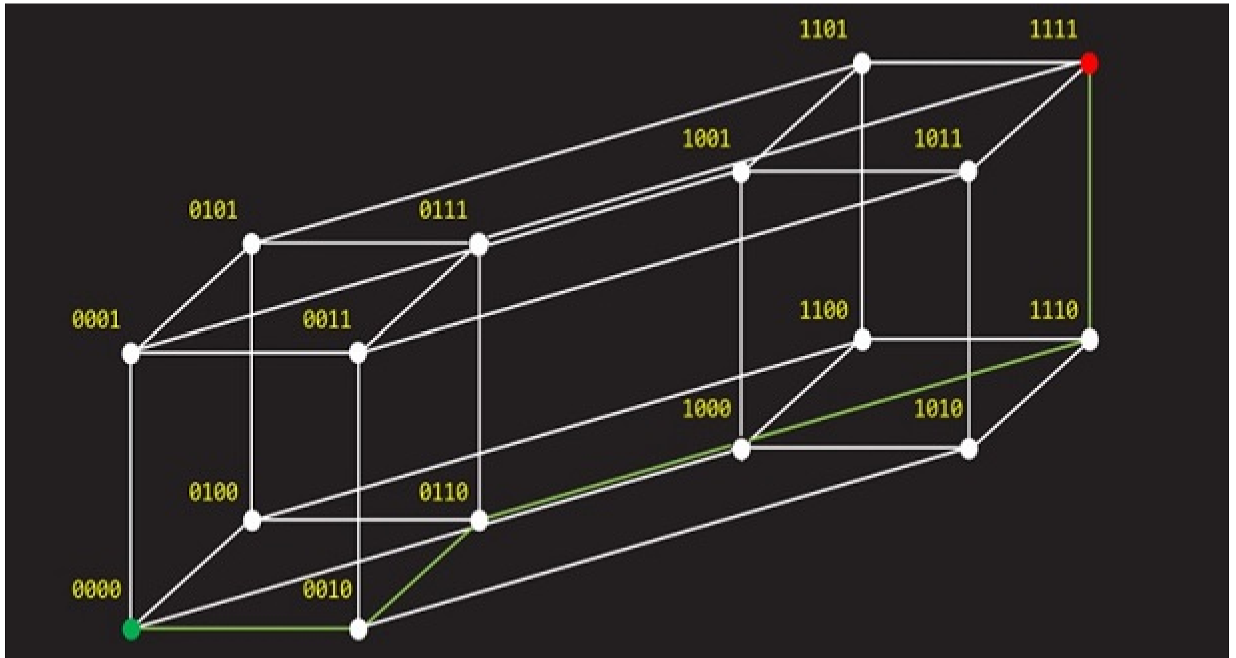


FIGURE B.2: Hamming distance cube 4D.

```
//First test case
localparam Test_case_1_MDAT = "00000000000000000000000000000000";
localparam  Test_case_1_i00 = "00000000000000000000000000000000";
localparam  Test_case_1_i01 = "00000000000000000000000000000000";
localparam  Test_case_1_i02 = "00000000000000000000000000000000";
localparam  Test_case_1_i03 = "00000000000000000000000000000000";
localparam  Test_case_1_i04 = "00000000000000000000000000000000";
localparam  Test_case_1_i05 = "00000000000000000000000000000000";
localparam  Test_case_1_i06 = "00000000000000000000000000000000";

// Second test case
localparam Test_case_2_MDAT = "00010001000100010001000100010001";
localparam  Test_case_2_i00 = "10111011101110111011101110111011";
localparam  Test_case_2_i01 = "11001100110011001100110011001100";
localparam  Test_case_2_i02 = "11011101110111011101110111011101";
localparam  Test_case_2_i03 = "10011001100110011001100110011001";
localparam  Test_case_2_i04 = "11011101110111011101110111011101";
localparam  Test_case_2_i05 = "11011101110111011101110111011101";
localparam  Test_case_2_i06 = "10011001100110011001100110011001";


// Third test case
localparam Test_case_3_MDAT = "00100010001000100010001000100010";
localparam  Test_case_3_i00 = "10101010101010101010101010101010";
localparam  Test_case_3_i01 = "11111111111111111111111111111111";
localparam  Test_case_3_i02 = "01110111011101110111011101110111";
localparam  Test_case_3_i03 = "11101110111011101110111011101110";
localparam  Test_case_3_i04 = "01110111011101110111011101110111";
```

```
localparam  Test_case_3_i05 = "01110111011101110111011101110111";
localparam  Test_case_3_i06 = "10101010101010101010101010101010";

// Forth test case
localparam Test_case_4_MDAT = "00110011001100110011001100110011";
localparam  Test_case_4_i00 = "01110111011101110111011101110111";
localparam  Test_case_4_i01 = "11101110111011101110111011101110";
localparam  Test_case_4_i02 = "11111111111111111111111111111111";
localparam  Test_case_4_i03 = "10101010101010101010101010101010";
localparam  Test_case_4_i04 = "11111111111111111111111111111111";
localparam  Test_case_4_i05 = "11111111111111111111111111111111";
localparam  Test_case_4_i06 = "01110111011101110111011101110111";

// Fifth test case
localparam Test_case_5_MDAT = "01000100010001000100010001000100";
localparam  Test_case_5_i00 = "11101110111011101110111011101110";
localparam  Test_case_5_i01 = "10011001100110011001100110011001";
localparam  Test_case_5_i02 = "10001000100010001000100010001000";
localparam  Test_case_5_i03 = "11011101110111011101110111011101";
localparam  Test_case_5_i04 = "10001000100010001000100010001000";
localparam  Test_case_5_i05 = "10001000100010001000100010001000";
localparam  Test_case_5_i06 = "11001100110011001100110011001100";

//Sixth test case
localparam Test_case_6_MDAT = "01010101010101010101010101010101";
localparam  Test_case_6_i00 = "11011101110111011101110111011101";
localparam  Test_case_6_i01 = "10001000100010001000100010001000";
localparam  Test_case_6_i02 = "11001100110011001100110011001100";
localparam  Test_case_6_i03 = "10011001100110011001100110011001";
localparam  Test_case_6_i04 = "11101110111011101110111011101110";
localparam  Test_case_6_i05 = "11011101110111011101110111011101";
localparam  Test_case_6_i06 = "11001100110011001100110011001100";

//Seventh test case
localparam Test_case_7_MDAT = "01100110011001100110011001100110";
localparam  Test_case_7_i00 = "10101010101010101010101010101010";
localparam  Test_case_7_i01 = "11101110111011101110111011101110";
localparam  Test_case_7_i02 = "01110111011101110111011101110111";
localparam  Test_case_7_i03 = "11111111111111111111111111111111";
localparam  Test_case_7_i04 = "11101110111011101110111011101110";
localparam  Test_case_7_i05 = "11011101110111011101110111011101";
localparam  Test_case_7_i06 = "10101010101010101010101010101010";

//Eighth test case
localparam Test_case_8_MDAT = "01110111011101110111011101110111";
localparam  Test_case_8_i00 = "00100010001000100010001000100010";
localparam  Test_case_8_i01 = "11111111111111111111111111111111";
localparam  Test_case_8_i02 = "01000100010001000100010001000100";
localparam  Test_case_8_i03 = "01110111011101110111011101110111";
localparam  Test_case_8_i04 = "11011101110111011101110111011101";
localparam  Test_case_8_i05 = "11011101110111011101110111011101";
```

```
localparam  Test_case_8_i06 = "00100010001000100010001000100010";

//Ninth test case
localparam Test_case_9_MDAT = "10001000100010001000100010001000";
localparam  Test_case_9_i00 = "01000100010001000100010001000100";
localparam  Test_case_9_i01 = "00100010001000100010001000100010";
localparam  Test_case_9_i02 = "01010101010101010101010101010101";
localparam  Test_case_9_i03 = "00010001000100010001000100010001";
localparam  Test_case_9_i04 = "00100010001000100010001000100010";
localparam  Test_case_9_i05 = "00010001000100010001000100010001";
localparam  Test_case_9_i06 = "01000100010001000100010001000100";

//Tenth test case
localparam Test_case_10_MDAT = "10011001100110011001100110011001";
localparam  Test_case_10_i00 = "00000000000000000000000000000000";
localparam  Test_case_10_i01 = "00010001000100010001000100010001";
localparam  Test_case_10_i02 = "01000100010001000100010001000100";
localparam  Test_case_10_i03 = "01010101010101010101010101010101";
localparam  Test_case_10_i04 = "10111011101110111011101110111011";
localparam  Test_case_10_i05 = "01110111011101110111011101110111";
localparam  Test_case_10_i06 = "01010101010101010101010101010101";

//Eleventh test case
localparam Test_case_11_MDAT = "10101010101010101010101010101010";
localparam  Test_case_11_i00 = "01100110011001100110011001100110";
localparam  Test_case_11_i01 = "01100110011001100110011001100110";
localparam  Test_case_11_i02 = "01110111011101110111011101110111";
localparam  Test_case_11_i03 = "00110011001100110011001100110011";
localparam  Test_case_11_i04 = "01000100010001000100010001000100";
localparam  Test_case_11_i05 = "01100110011001100110011001100110";
localparam  Test_case_11_i06 = "00110011001100110011001100110011";

//Twelfth test case
localparam Test_case_12_MDAT = "10101010101010101010101010101010";
localparam  Test_case_12_i00 = "01100110011001100110011001100110";
localparam  Test_case_12_i01 = "01100110011001100110011001100110";
localparam  Test_case_12_i02 = "01100110011001100110011001100110";
localparam  Test_case_12_i03 = "00000000000000000000000000000000";
localparam  Test_case_12_i04 = "01000100010001000100010001000100";
localparam  Test_case_12_i05 = "00000000000000000000000000000000";
localparam  Test_case_12_i06 = "01010101010101010101010101010101";

//Thirteenth test case
localparam Test_case_13_MDAT = "11001100110011001100110011001100";
localparam  Test_case_13_i00 = "00010001000100010001000100010001";
localparam  Test_case_13_i01 = "01110111011101110111011101110111";
localparam  Test_case_13_i02 = "01110111011101110111011101110111";
localparam  Test_case_13_i03 = "01000100010001000100010001000100";
localparam  Test_case_13_i04 = "10101010101010101010101010101010";
localparam  Test_case_13_i05 = "01000100010001000100010001000100";
localparam  Test_case_13_i06 = "00010001000100010001000100010001";
```

```
//Fourteenth test case
localparam Test_case_14_MDAT = "11011101110111011101110111011101";
localparam  Test_case_14_i00 = "00000000000000000000000000000000";
localparam  Test_case_14_i01 = "01110111011101110111011101110111";
localparam  Test_case_14_i02 = "01100110011001100110011001100110";
localparam  Test_case_14_i03 = "00100010001000100010001000100010";
localparam  Test_case_14_i04 = "01100110011001100110011001100110";
localparam  Test_case_14_i05 = "00100010001000100010001000100010";
localparam  Test_case_14_i06 = "01110111011101110111011101110111";


//Fifteenth test case
localparam Test_case_15_MDAT = "11111111111111111111111111111111";
localparam  Test_case_15_i00 = "11111111111111111111111111111111";
localparam  Test_case_15_i01 = "11111111111111111111111111111111";
localparam  Test_case_15_i02 = "11111111111111111111111111111111";
localparam  Test_case_15_i03 = "11111111111111111111111111111111";
localparam  Test_case_15_i04 = "11111111111111111111111111111111";
localparam  Test_case_15_i05 = "11111111111111111111111111111111";
localparam  Test_case_15_i06 = "11111111111111111111111111111111";
```

Example of Verilog test-bench code:

```
module darcTB;
// Inputs
reg [31:0] FP;
reg reset;
reg CLK;
// Outputs
wire [31:0] r00;
wire [31:0] r01;
wire [31:0] FPOUT;
// Bidirs
wire [31:0] M_ADDR;
wire [31:0] M_DATA;
wire M_RD;
wire M_WR;
wire M_RDY;
reg MWAIT;
reg [31:0] MDAT;


darc1 uut (
.r00(r00),
.r01(r01),
.FPOUT(FPOUT),
.FP(FP),
.M_ADDR(M_ADDR),
```

```
.M_DATA(M_DATA),
.M_RD(M_RD),
.M_WR(M_WR),
.M_RDY(M_RDY),
.reset(reset),
.CLK(CLK)
);
// Instantiate the Unit Under Test (UUT)

mem mymem (

.M_ADDR(M_ADDR),
.M_DATA(M_DATA),
.M_RD(M_RD),
.M_WR(M_WR),
.M_RDY(M_RDY),
.MWAIT(MWAIT),
.MDAT(MDAT)

);

localparam TCLK = 10;

//First test case
localparam Test_case_1_MDAT = "00000000000000000000000000000000";
localparam  Test_case_1_i00 = "00000000000000000000000000000000";
localparam  Test_case_1_i01 = "00000000000000000000000000000000";
localparam  Test_case_1_i02 = "00000000000000000000000000000000";
localparam  Test_case_1_i03 = "00000000000000000000000000000000";
localparam  Test_case_1_i04 = "00000000000000000000000000000000";
localparam  Test_case_1_i05 = "00000000000000000000000000000000";
localparam  Test_case_1_i06 = "00000000000000000000000000000000";
// Second test case
localparam Test_case_2_MDAT = "00010001000100010001000100010001";
localparam  Test_case_2_i00 = "10111011101110111011101110111011";
localparam  Test_case_2_i01 = "11001100110011001100110011001100";
localparam  Test_case_2_i02 = "11011101110111011101110111011101";
localparam  Test_case_2_i03 = "10011001100110011001100110011001";
localparam  Test_case_2_i04 = "11011101110111011101110111011101";
localparam  Test_case_2_i05 = "11011101110111011101110111011101";
localparam  Test_case_2_i06 = "10011001100110011001100110011001";


// Third test case
localparam Test_case_3_MDAT = "00100010001000100010001000100010";
localparam  Test_case_3_i00 = "10101010101010101010101010101010";
localparam  Test_case_3_i01 = "11111111111111111111111111111111";
localparam  Test_case_3_i02 = "01110111011101110111011101110111";
localparam  Test_case_3_i03 = "11101110111011101110111011101110";
localparam  Test_case_3_i04 = "01110111011101110111011101110111";
localparam  Test_case_3_i05 = "01110111011101110111011101110111";
```

```
localparam  Test_case_3_i06 = "10101010101010101010101010101010";

// Forth test case
localparam Test_case_4_MDAT = "00110011001100110011001100110011";
localparam  Test_case_4_i00 = "01110111011101110111011101110111";
localparam  Test_case_4_i01 = "11101110111011101110111011101110";
localparam  Test_case_4_i02 = "11111111111111111111111111111111";
localparam  Test_case_4_i03 = "10101010101010101010101010101010";
localparam  Test_case_4_i04 = "11111111111111111111111111111111";
localparam  Test_case_4_i05 = "11111111111111111111111111111111";
localparam  Test_case_4_i06 = "01110111011101110111011101110111";

// Fifth test case
localparam Test_case_5_MDAT = "01000100010001000100010001000100";
localparam  Test_case_5_i00 = "11101110111011101110111011101110";
localparam  Test_case_5_i01 = "10011001100110011001100110011001";
localparam  Test_case_5_i02 = "10001000100010001000100010001000";
localparam  Test_case_5_i03 = "11011101110111011101110111011101";
localparam  Test_case_5_i04 = "10001000100010001000100010001000";
localparam  Test_case_5_i05 = "10001000100010001000100010001000";
localparam  Test_case_5_i06 = "11001100110011001100110011001100";

//Sixth test case
localparam Test_case_6_MDAT = "01010101010101010101010101010101";
localparam  Test_case_6_i00 = "11011101110111011101110111011101";
localparam  Test_case_6_i01 = "10001000100010001000100010001000";
localparam  Test_case_6_i02 = "11001100110011001100110011001100";
localparam  Test_case_6_i03 = "10011001100110011001100110011001";
localparam  Test_case_6_i04 = "11101110111011101110111011101110";
localparam  Test_case_6_i05 = "11011101110111011101110111011101";
localparam  Test_case_6_i06 = "11001100110011001100110011001100";

//Seventh test case
localparam Test_case_7_MDAT = "01100110011001100110011001100110";
localparam  Test_case_7_i00 = "10101010101010101010101010101010";
localparam  Test_case_7_i01 = "11101110111011101110111011101110";
localparam  Test_case_7_i02 = "01110111011101110111011101110111";
localparam  Test_case_7_i03 = "11111111111111111111111111111111";
localparam  Test_case_7_i04 = "11101110111011101110111011101110";
localparam  Test_case_7_i05 = "11011101110111011101110111011101";
localparam  Test_case_7_i06 = "10101010101010101010101010101010";

//Eighth test case
localparam Test_case_8_MDAT = "01110111011101110111011101110111";
localparam  Test_case_8_i00 = "00100010001000100010001000100010";
localparam  Test_case_8_i01 = "11111111111111111111111111111111";
localparam  Test_case_8_i02 = "01000100010001000100010001000100";
localparam  Test_case_8_i03 = "01110111011101110111011101110111";
localparam  Test_case_8_i04 = "11011101110111011101110111011101";
localparam  Test_case_8_i05 = "11011101110111011101110111011101";
localparam  Test_case_8_i06 = "00100010001000100010001000100010";
```

```
//Ninth test case
localparam Test_case_9_MDAT = "10001000100010001000100010001000";
localparam  Test_case_9_i00 = "01000100010001000100010001000100";
localparam  Test_case_9_i01 = "00100010001000100010001000100010";
localparam  Test_case_9_i02 = "01010101010101010101010101010101";
localparam  Test_case_9_i03 = "00010001000100010001000100010001";
localparam  Test_case_9_i04 = "00100010001000100010001000100010";
localparam  Test_case_9_i05 = "00010001000100010001000100010001";
localparam  Test_case_9_i06 = "01000100010001000100010001000100";

//Tenth test case
localparam Test_case_10_MDAT = "10011001100110011001100110011001";
localparam  Test_case_10_i00 = "00000000000000000000000000000000";
localparam  Test_case_10_i01 = "00010001000100010001000100010001";
localparam  Test_case_10_i02 = "01000100010001000100010001000100";
localparam  Test_case_10_i03 = "01010101010101010101010101010101";
localparam  Test_case_10_i04 = "10111011101110111011101110111011";
localparam  Test_case_10_i05 = "01110111011101110111011101110111";
localparam  Test_case_10_i06 = "01010101010101010101010101010101";

//Eleventh test case
localparam Test_case_11_MDAT = "10101010101010101010101010101010";
localparam  Test_case_11_i00 = "01100110011001100110011001100110";
localparam  Test_case_11_i01 = "01100110011001100110011001100110";
localparam  Test_case_11_i02 = "01110111011101110111011101110111";
localparam  Test_case_11_i03 = "00110011001100110011001100110011";
localparam  Test_case_11_i04 = "01000100010001000100010001000100";
localparam  Test_case_11_i05 = "01100110011001100110011001100110";
localparam  Test_case_11_i06 = "00110011001100110011001100110011";

//Twelfth test case
localparam Test_case_12_MDAT = "10101010101010101010101010101010";
localparam  Test_case_12_i00 = "01100110011001100110011001100110";
localparam  Test_case_12_i01 = "01100110011001100110011001100110";
localparam  Test_case_12_i02 = "01100110011001100110011001100110";
localparam  Test_case_12_i03 = "00000000000000000000000000000000";
localparam  Test_case_12_i04 = "01000100010001000100010001000100";
localparam  Test_case_12_i05 = "00000000000000000000000000000000";
localparam  Test_case_12_i06 = "01010101010101010101010101010101";

//Thirteenth test case
localparam Test_case_13_MDAT = "11001100110011001100110011001100";
localparam  Test_case_13_i00 = "00010001000100010001000100010001";
localparam  Test_case_13_i01 = "01110111011101110111011101110111";
localparam  Test_case_13_i02 = "01110111011101110111011101110111";
localparam  Test_case_13_i03 = "01000100010001000100010001000100";
localparam  Test_case_13_i04 = "10101010101010101010101010101010";
localparam  Test_case_13_i05 = "01000100010001000100010001000100";
localparam  Test_case_13_i06 = "00010001000100010001000100010001";
```

```
//Fourteenth test case
localparam Test_case_14_MDAT = "11011101110111011101110111011101";
localparam  Test_case_14_i00 = "00000000000000000000000000000000";
localparam  Test_case_14_i01 = "01110111011101110111011101110111";
localparam  Test_case_14_i02 = "01100110011001100110011001100110";
localparam  Test_case_14_i03 = "00100010001000100010001000100010";
localparam  Test_case_14_i04 = "01100110011001100110011001100110";
localparam  Test_case_14_i05 = "00100010001000100010001000100010";
localparam  Test_case_14_i06 = "01110111011101110111011101110111";


//Fifteenth test case
localparam Test_case_15_MDAT = "11111111111111111111111111111111";
localparam  Test_case_15_i00 = "11111111111111111111111111111111";
localparam  Test_case_15_i01 = "11111111111111111111111111111111";
localparam  Test_case_15_i02 = "11111111111111111111111111111111";
localparam  Test_case_15_i03 = "11111111111111111111111111111111";
localparam  Test_case_15_i04 = "11111111111111111111111111111111";
localparam  Test_case_15_i05 = "11111111111111111111111111111111";
localparam  Test_case_15_i06 = "11111111111111111111111111111111";


initial begin

$dumpfile ("switching.vcd");
//  --- TEST CASE 1 ---------------

$display("%dns TEST 1",$stime);
FP = 0;
reset = 1;
CLK = 0;
MWAIT =1;
MDAT =32'h12345678;

// Wait 100 ns for global reset to finish

#(TCLK*2);
reset=0;
#(TCLK);
FP =32'h00000000;
MDAT = Test_case_1_MDAT;
#(TCLK*13);

//-- END of test 1 --------------



//  --- TEST CASE 2 ---------------

$display("%dns TEST 2",$stime);
```

```
FP = 0;
reset = 1;
CLK = 0;
MWAIT =1;
MDAT =32'h12345678;

// Wait 100 ns for global reset to finish

#(TCLK*2);
reset=0;
#(TCLK);
FP =32'h00000000;
MDAT = Test_case_2_MDAT;
#(TCLK*13);

//-- END of test 2 --------------



//  --- TEST CASE 3 ---------------

$display("%dns TEST 3",$stime);
FP = 0;
reset = 1;
CLK = 0;
MWAIT =1;
MDAT =32'h12345678;

// Wait 100 ns for global reset to finish

#(TCLK*2);
reset=0;
#(TCLK);
FP =32'h00000000;
MDAT = Test_case_3_MDAT;
#(TCLK*13);

//-- END of test 3 --------------



//  --- TEST CASE 4 ---------------

$display("%dns TEST 4",$stime);
FP = 0;
reset = 1;
CLK = 0;
MWAIT =1;
MDAT =32'h12345678;

// Wait 100 ns for global reset to finish
```

```
#(TCLK*2);
reset=0;
#(TCLK);
FP =32'h00000000;
MDAT = Test_case_4_MDAT;
#(TCLK*13);

//-- END of test 4 --------------



//  --- TEST CASE 5 ---------------

$display("%dns TEST 5",$stime);
FP = 0;
reset = 1;
CLK = 0;
MWAIT =1;
MDAT =32'h12345678;

// Wait 100 ns for global reset to finish

#(TCLK*2);
reset=0;
#(TCLK);
FP =32'h00000000;
MDAT = Test_case_5_MDAT;
#(TCLK*13);

//-- END of test 5 --------------



//  --- TEST CASE 6 ---------------

$display("%dns TEST 6",$stime);
FP = 0;
reset = 1;
CLK = 0;
MWAIT =1;
MDAT =32'h12345678;

// Wait 100 ns for global reset to finish

#(TCLK*2);
reset=0;
#(TCLK);
FP =32'h00000000;
MDAT = Test_case_6_MDAT;
#(TCLK*13);
```

```
//-- END of test 6 -------------



//  --- TEST CASE 7 ---------------

$display("%dns TEST 7",$stime);
FP = 0;
reset = 1;
CLK = 0;
MWAIT =1;
MDAT =32'h12345678;

// Wait 100 ns for global reset to finish

#(TCLK*2);
reset=0;
#(TCLK);
FP =32'h00000000;
MDAT = Test_case_7_MDAT;
#(TCLK*13);

//-- END of test 7 -------------



//  --- TEST CASE 8 ---------------

$display("%dns TEST 8",$stime);
FP = 0;
reset = 1;
CLK = 0;
MWAIT =1;
MDAT =32'h12345678;

// Wait 100 ns for global reset to finish

#(TCLK*2);
reset=0;
#(TCLK);
FP =32'h00000000;
MDAT = Test_case_8_MDAT;
#(TCLK*13);

//-- END of test 8 -------------



//  --- TEST CASE 9 ---------------
```

```
$display("%dns TEST 9",$stime);
FP = 0;
reset = 1;
CLK = 0;
MWAIT =1;
MDAT =32'h12345678;

// Wait 100 ns for global reset to finish

#(TCLK*2);
reset=0;
#(TCLK);
FP =32'h00000000;
MDAT = Test_case_9_MDAT;
#(TCLK*13);

//-- END of test 9 --------------



//  --- TEST CASE 10 ---------------

$display("%dns TEST 10",$stime);
FP = 0;
reset = 1;
CLK = 0;
MWAIT =1;
MDAT =32'h12345678;

// Wait 100 ns for global reset to finish

#(TCLK*2);
reset=0;
#(TCLK);
FP =32'h00000000;
MDAT = Test_case_10_MDAT;
#(TCLK*13);

//-- END of test 10 --------------



//  --- TEST CASE 11 ---------------

$display("%dns TEST 11",$stime);
FP = 0;
reset = 1;
CLK = 0;
MWAIT =1;
MDAT =32'h12345678;
```

```
// Wait 100 ns for global reset to finish

#(TCLK*2);
reset=0;
#(TCLK);
FP =32'h00000000;
MDAT = Test_case_11_MDAT;
#(TCLK*13);

//-- END of test 11 --------------



//  --- TEST CASE 12 ---------------

$display("%dns TEST 12",$stime);
FP = 0;
reset = 1;
CLK = 0;
MWAIT =1;
MDAT =32'h12345678;

// Wait 100 ns for global reset to finish

#(TCLK*2);
reset=0;
#(TCLK);
FP =32'h00000000;
MDAT = Test_case_12_MDAT;
#(TCLK*13);

//-- END of test 12 --------------



//  --- TEST CASE 13 ---------------

$display("%dns TEST 13",$stime);
FP = 0;
reset = 1;
CLK = 0;
MWAIT =1;
MDAT =32'h12345678;

// Wait 100 ns for global reset to finish

#(TCLK*2);
reset=0;
#(TCLK);
FP =32'h00000000;
MDAT = Test_case_13_MDAT;
```

```
#(TCLK*13);

//-- END of test 13 --------------



//  --- TEST CASE 14 ---------------

$display("%dns TEST 14",$stime);
FP = 0;
reset = 1;
CLK = 0;
MWAIT =1;
MDAT =32'h12345678;

// Wait 100 ns for global reset to finish

#(TCLK*2);
reset=0;
#(TCLK);
FP =32'h00000000;
MDAT = Test_case_14_MDAT;
#(TCLK*13);

//-- END of test 14 --------------



//  --- TEST CASE 15 ---------------

$display("%dns TEST 15",$stime);
FP = 0;
reset = 1;
CLK = 0;
MWAIT =1;
MDAT =32'h12345678;

// Wait 100 ns for global reset to finish

#(TCLK*2);
reset=0;
#(TCLK);
FP =32'h00000000;
MDAT = Test_case_15_MDAT;
#(TCLK*13);

//-- END of test 15 --------------



$stop();
```

```
end
always
begin
#(TCLK/2)  CLK =  ! CLK;
$dumpvars;
end
endmodule
```

Appendix C

Total Area

```
============================================================
Generated by:           Encounter(R) RTL Compiler v11.20-s017_1
Generated on:           Aug 14 2014  03:34:19 pm
Module:                 darc1
Technology library:     uk65lscllmvbbr_108c125_wc
Operating conditions:   uk65lscllmvbbr_108c125_wc (balanced_tree)
Wireload mode:          top
Area mode:              timing library
============================================================

Instance  Cells  Cell Area  Net Area  Total Area  Wireload
---------------------------------------------------------------
darc1       135       1668         0        1668       wl0 (D)

(D) = wireload is default in technology library
```

Appendix D

Power report

```
============================================================
Generated by:            Encounter(R) RTL Compiler v11.20-s017_1
Generated on:            Aug 14 2014  03:34:19 pm
Module:                  darc1
Technology library:      uk65lscllmvbbr_108c125_wc
Operating conditions:    uk65lscllmvbbr_108c125_wc (balanced_tree)
Wireload mode:           top
Area mode:               timing library
============================================================

Leakage    Dynamic      Total
Instance Cells Power(nW)  Power(nW)   Power(nW)
-------------------------------------------------
darc1      135  1557.027 3910317.601 3911874.629
```

Appendix E

Timing report

```
============================================================
Generated by:           Encounter(R) RTL Compiler v11.20-s017_1
Generated on:           Aug 14 2014  03:34:19 pm
Module:                 darc1
Technology library:     uk65lscllmvbbr_108c125_wc
Operating conditions:   uk65lscllmvbbr_108c125_wc (balanced_tree)
Wireload mode:          top
Area mode:              timing library
============================================================

Pin            Type       Fanout Load Slew Delay Arrival
(fF)  (ps)  (ps)   (ps)
-----------------------------------------------------------------
(clock my_clock)   launch                                  0 R
Mstate_reg[1]/CK                            100             0 R
Mstate_reg[1]/Q    SDFQSM2RA         3  3.9   63  +220     220 R
g384/B                                            +0     220
g384/Z             OR3M2R           32 53.5  660  +443     664 R
FPout_reg[6]/SE    SDFQSM2RA                      +0     664
FPout_reg[6]/CK    setup                    100  +332     995 R
- - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - -
(clock my_clock)   capture                              1400 R
uncertainty                   -200    1200 R
-----------------------------------------------------------------
Cost Group   : 'my_clock' (path_group 'my_clock')
Timing slack :     205ps
Start-point  : Mstate_reg[1]/CK
End-point    : FPout_reg[6]/SE
```

List of Abbreviations

ALU	**A**rithmetic **L**ogic **U**nit
ASIC	**A**pplications **S**pecific **I**ntegrated **C**ircuit
CISC	**C**omplex **I**nstruction **S**et **C**omputing
CoDA	**Co**-processor **D**ominant **A**rchitecture
EPRSC	**E**ngineering and **P**hysical **S**cience **R**esearch **C**ouncil
FIFO	**F**irst **I**n **F**irst **O**out
FP	**F**rame **P**oint
FPGA	**F**ield **P**rpgrammable **G**ate **A**array
FSM	**F**inite **S**tate **M**achine
GPRF	**G**eneral **P**urpose **R**egister **F**ile
HDL	**H**ardware **D**escription **L**anguage
IC	**I**ntegrated **C**ircuit
IP	**I**ntellectual **P**roperty
LUT	**L**ook **U**p **T**able
NMOS	**N**-type **M**etal **O**xide **S**emiconductor
NOS	**N**ext **O**n **S**tack
PMOS	**P**-type **M**etal **O**xide **S**emiconductor
RB	**R**egister **B**ank
RF	**R**egister **F**ile
RISC	**R**educed **I**nstruction **S**et **C**omputing
RTL	**R**egister **T**ransfer **L**evel
SP	**S**tack **P**oint
TCL	**T**ool **C**ommand **L**anguage
TOP	**T**op **O**f **S**tack
VCD	**V**alue **C**hange **D**ump
VHDL	**V**HSIC **H**ardware **D**escription **L**anguage
VLIW	**V**ery **L**arge **I**nstruction **W**ord
VLSI	**V**ery **L**arge **S**cale **I**ntegration

Bibliography

[1] URL: https://benchmarksgame.alioth.debian.org/u64q/performance.php?test=fasta.

[2] M. B. P. C. M. G. B. H. R. M. Wolfgang Arden. "More than Moore". In: *White Paper* (2013).

[3] C. Bailey. "Optimization Technique for Stack Based Architectures". PhD thesis. 1996.

[4] H. S. S. Chris Crispin Bailey. "Towards Scalable Parallelism with Stack Machines". In: *HIPEAC 2012 TUTORIAL*. 2012.

[5] T. M. By Friedrich L. Bauer. 1952. URL: https://www.cs.umd.edu/users/oleary/cggg/bauer.pdf.

[6] *Binarytrees benchmarks*. URL: https://benchmarksgame.alioth.debian.org/u64q/performance.php?test=binarytrees.

[7] *Box and Whisker chart*. URL: http://www.bbc.co.uk/schools/gcsebitesize/maths/statistics/representingdata3hirev6.shtml.

[8] M. J. L. M. H. G.-Y. W. D. Brooks. "The Accelerator Store framework for high-performance, low-power accelerator-based systems". In: *the IEEE Computer Society* (2010).

[9] *Calculate MIPS*. URL: https://students.cs.byu.edu/~cs224ta/labs/L03-blinky/MIPS.php.

[10] B. V. I. T. M. Conte. "A Power Model for Register Sharing Structures". In: *Dystributed Embedded Systems* (2008), p. 132.

[11] *Core Benchmarks*. URL: https://benchmarksgame.alioth.debian.org/u64q/performance.php?test=binarytrees.

[12] *CUDA Parallel Computing*. URL: http://www.nvidia.co.uk/object/cuda-parallel-computing-uk.html.

[13] L. H. and. D. Patterson. *Computer architecture a Quantitative Approach*. p.93. Morgan Kaufmann, 1990.

[14] D. R. K. a. D. R. W. Shukri A. Wakid. *Scimark*. IEEE software. 1999.

[15] C. S. a. E. T. O. S. Universe. *Backtracking Algorithms*. The Ohio State Universe. URL: http://web.cse.ohio-state.edu/~gurari/course/cis680/cis680Ch19.html.

[16] Syed Mahfuzul Aziz Etienne Sicard. "Introducing 65 nm technology in Microwind 3". In: *Microwind Application Note* (2006).

[17] *fannkuch Benchmarks*. URL: https://benchmarksgame.alioth.debian.org/u64q/performance.php?test=fannkuch.

[18] Jeanne Ferrante and Karl J. Ottenstein. "The Program Dependence Graph and Its Use in Optimization". In: *ACM Transactions on Programming Languages and Systems* 9.3 (1987), pp. 319–349.

[19] J. Fisher. *In the paper that innaugurate the term "instruction-level parallelism*. Tech. rep. 1991.

[20] S. Furber. *ARM System Architecture*. Addison Wesley Longman, 1996.

[21] . T. M. Sampson J. Venkatesh G. G.-H. N. G. S. . S. S. "Efficient complex operators for irregular codes," in High Performance Computer Architecture". In: IEEE 17th International Symposium, San Antonio, TX, 2011.

[22] P. Gargini. *International Technology Roadmap for Semiconductor*. URL: http://www.itrs.net.

[23] D. M. H. Neil H. *CMOS VLSI DESIGN A circuits and Systems Perfective*. Ed. by New York: Addison Wesley. 2011.

[24] C. L. Hamblin. "An addressless coding scheme based on mathematical notation". In: *First Australian Conference on Computing and Data Processing*. 1957.

[25] D. A. P. John L. Hennessy. *Computer Architecture A Quantitative Approach*. p.3. Morgan Kaufmann, 2001.

[26] D. A. P. John L. Hennessy. *Computer Organization and Design: The Hardware and Software Interface*. Morgan Kaufmann, 2010.

[27] Jhon L Hennessy and David Patterson. *The Role of high-level languages and Compilers.* Ed. by Morgan. p.11. Computer Architecture A Quantitative approach, 1997.

[28] John L Hennessy and David A. *The Major Hurdle of Pipelining- Pipelinine Hazards*. 257-284. Computer Architecture A Quantitative Approach, 1990.

[29] Paul Horowiths. *State diagram as degin tools*. Ed. by Winfield. p.512. The Art of electronics, 2010.

[30] M. Horowitz. *Computing's Energy Problem*. 2014. URL: http://eecs.oregonstate.edu/research/vlsi/teaching/ECE471_WIN15/mark_horowitz_ISSCC_2014.pdf..

[31] M. Horowitz. "Computing's energy problem (and what we can do about it)". In: *Solid-State Circuits Conference Digest of Technical Papers (ISSCC), 2014 IEEE International, San Francisco* (2014).

[32] *INMOS, transputer Architecture*. 1987. URL: http://www.transputer.net/tn/06/tn06.html.

[33] Intel. *FPGA Solution*. 2016. URL: https://www-ssl.intel.com/content/www/uk/en/fpga/solutions.html.

[34] B. Ivey. *Low Power Design Guide*. Tech. rep. Microchip, 2011.

[35] Philips J.Koopman Jr. "A Preliminary Exploration of Optimized Stack Code Generation". In: *Journal of Forth Applications and Research* (1994), pp. 241–251.

[36] P. Koopman. *Stack Computer The new wave*. Mountain view press, 1989.

[37] Rocket Science L. D. Jasio. *This is not rocket Science*. Lulu Enterproses, 2015.

[38] C. E. LaForest. "Second-Generation Stack Computer". PhD thesis. University of Waterloo, Canada, 2007.

[39] J. Lemieux. *Introduction to ARM thumb*. 2003. URL: http://www.embedded.com/electronics-blogs/beginner-s-corner/4024632/Introduction-to-ARM-thumb.

[40] *Low Power VLSI chip design: Circuit design techniques*. URL: http://www.eeherald.com/section/design-guide/Low-Power-VLSI-Design.html.

[41] G. V. J. S. N. G. S. G. V. B. o. Lugo-Martine. "Conservation Cores: Reducing the Energy of Mature Computations". In: ASPLOS 10, 2010.

[42] T. Martin. *The Insider's guide to the Philips ARM 7 Based Microcontroller*. Hitex(UK) Ltd., Coventry, 2005.

[43] Burroughs MCP. URL: https://en.wikipedia.org/wiki/Burroughs_MCP.

[44] Chuck Moor. *C18 colorForth Compiler*. 2001. URL: http://www.complang.tuwien.ac.at/anton/euroforth/ef01/moore01a.pdf.

[45] Samuel K. Moore. IEEE Spectrum. 2016. URL: http://spectrum.ieee.org/semiconductors/processors/breaking-the-multicore-bottleneck.

[46] Prof. Dr.-Ing. Christian Märtin. *Post-Dennard Scaling and the final Years of Moore's Law, Consequences for the Evolution of Multicore-Architectures*. Tech. rep. Hochschule Augsburg University of Applied Sciences Faculty of Computer Science September, 2014.

[47] *nbody Benchmark*. URL: https://benchmarksgame.alioth.debian.org/u64q/performance.php?test=nbody.

[48] Writer P. I. Philips. "Gazing into the void Many Core. ARM". In: 2012.

[49] A. S. Preeti Ranjan. *Power consumption state machine*. p.124-125. London, Springer, 2010.

[50] G. B. B. A. Research. "Options for embedded systems. Constraints, challenges, and approaches". In: HPEC. 2001.

[51] *RTX2010*. 1988. URL: https://en.wikipedia.org/wiki/RTX2010.

[52] Jikes RVM. *Compilers*. 2010. URL: http://www.jikesrvm.org/UserGuide/Compilers/index.htmlVisitedon27/08/2016.

[53] J. S. F. a. S. P. W. Michale R. "The History of the Microprocessor". In: *Bell Labs Technical Journal* (1997), pp. 29–56.

[54] V. G. C.-H. H. a. Sankaralingam. "Dynamically Specialized Datapaths for Energy Efficient Computing". In: 2011, IEEE International Symposium on High Performance Computer Architecture.

[55] D. Seal. *Architecture Reference Manual*. Addison-Wesley, 2000.

[56] Bailey C Shannon M. "Global Stack Allocation-Register Allocation for Stack Machines". In: the proceeding of Euroforth, 2006.

[57] Kennet L. Short. *Finite State Machine*. Ed. by Perason Education. p.380. VHDL FOR ENGINEERS.

[58] Hendrawan Soelernan and Kaushik Roy. "Ultra-Low Power Digital Subthreshold Logic Circuits". In: *Low Power Electronics and Design* (1999).

[59] *spectral benchmark*. URL: https://benchmarksgame.alioth.debian.org/u64q/performance.php?test=spectral.

[60] J. R. A. A. Girish Aramanekoppa Subbarao. "Low-power Microprocessor based on Stack". MA thesis. Lund University, 2015.

[61] M. B. T. Steven Swanson. “GreenDroid:Exploring the Next Evolution in Smartphone Application Processors”. In: *IEEE Communication Magazine* (2001), pp. 112–119.

[62] M. B. Taylor. “A LANDSCAPE OF THE NEW DARK SILICON DESIGN REGIME”. In: *IEEE Computer Society* (2013), pp. 8–19.

[63] M. B. Taylor. “Is dark silicon useful? Harnessing the four horsemen of the coming dark silicon apocalypse”. In: *In Design Automation Conference (DAC), 2012 49th ACME DAC IEEE, San Francisco* (2012).

[64] *The History of the HP 3000*. URL: http://www.robelle.com/smugbook/classic.html.

[65] A. Turing. “A Study in Light and Shadow”. In: *Xlibris Corporation* (2003).

[66] R. Schreiber V. Kathail S. Aditya and B. Rau. “PICO: automatically designing custom computers”. In: *Computer IEEE* (2002), pp. 39–47.

[67] J. S. N. G.-H. S. K. V. M. B. T. S. S. Ganesh Venkatesh. “QsCores: Trading Dark SIlicon for Scalable Energy Efficiency with Quasi-Specific Cores”. In: Micro 11, 2011.

[68] Ethan J. Warner and Deepak Ganapathy. “DEFINING THERMAL DESIGN POWER BASED ON REAL-WORLD USAGE MODELS”. In: *ITHERM* (2008).

[69] Wikibooks. 2015. URL: https://en.wikibooks.org/wiki/Microprocessor_Design/Register_File.

[70] R. William. *Computer system arthictecture a networking approach*. p. 535. Essex, Addison Wesley, 2000.

[71] N. G.-H. S. R. S. S. Qiaoshi Zheng. “Exploring Energy Scalability in Coprocessor-Dominated”. In: *in Embedded Computing Systems* (2013).