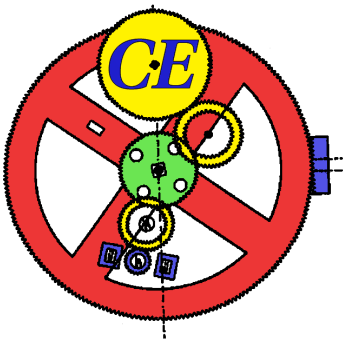


Computer Engineering
Mekelweg 4,
2628 CD Delft
The Netherlands
http://ce.et.tudelft.nl/

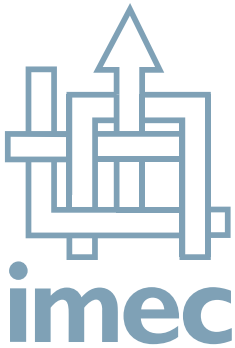
2006

MSc THESIS

Power and Performance Optimization for ADRES

Frank Bouwens

CE-MS-2006-12



Abstract

Reconfigurable computational architectures are envisioned to deliver power efficient, high performance, flexible platforms for embedded systems design. One such architecture, the *Architecture for Dynamically Reconfigurable Embedded Systems* (ADRES) is being developed by the IMEC DESICS department. ADRES is a template configurable by designers requirements. The architecture consists of a VLIW engine tightly coupled with a Course Grain Reconfigurable Array (CGRA) targeting multi-media and mobile devices. This thesis project delivers (1) a novel toolflow for power simulations, synthesis & performance evaluation of a variety of ADRES architectures for energy-delay trade-off. The toolflow utilizes an innovative instruction-set simulator based approach for capturing switching activity of synthesis-invariant components in addition to regular VHDL RTL simulations. The FFT, IDCT and MPEG2 benchmarks are utilized for evaluation of different architectures. Furthermore, (2) optimization techniques are proposed utilizing clock gating, operand isolation, memory segmentation and architectural modifications. Clock gating reduced power by 10 - 21%. Combined with operand isolation a reduction of 39 - 53% is obtained. Segmenting a memory of 256wx128b into 2 parts reduces memory power by 20% with 27% area increase. Architecture modifications like sharing register files among four functional units reducing power with 14 - 16% compared to a reference 4x4 architecture with 64 - 80mW at 100MHz and 619nJ - 1944uJ. On top of the optimizations (3) an extensive power analysis of different architectures has been performed. The proposed architecture with the evaluated optimizations including pipelining consumes 63.87 - 82mW at 312MHz and 307nJ - 702.7uJ when synthesized with 90nm Synopsys TSMC library (tcbn90ghptc).

TU Delft
Delft University of Technology

Faculty of Electrical Engineering, Mathematics and Computer Science

Power and Performance Optimization for ADRES

Analysis, optimization and synthesis of a coarse-grained reconfigurable array

---

THESIS

submitted in partial fulfillment of the
requirements for the degree of

MASTER OF SCIENCE

in

COMPUTER ENGINEERING

by

Frank Bouwens
born in Oostburg, The Netherlands

Computer Engineering
Department of Electrical Engineering
Faculty of Electrical Engineering, Mathematics and Computer Science
Delft University of Technology

Power and Performance Optimization for ADRES

by Frank Bouwens

Abstract

Reconfigurable computational architectures are envisioned to deliver power efficient, high performance, flexible platforms for embedded systems design. One such architecture, the *Architecture for Dynamically Reconfigurable Embedded Systems* (ADRES) is being developed by the IMEC DESICS department. ADRES is a template configurable by designers requirements. The architecture consists of a VLIW engine tightly coupled with a Course Grain Reconfigurable Array (CGRA) targeting multi-media and mobile devices. This thesis project delivers (1) a novel toolflow for power simulations, synthesis & performance evaluation of a variety of ADRES architectures for energy-delay trade-off. The toolflow utilizes an innovative instruction-set simulator based approach for capturing switching activity of synthesis-invariant components in addition to regular VHDL RTL simulations. The FFT, IDCT and MPEG2 benchmarks are utilized for evaluation of different architectures. Furthermore, (2) optimization techniques are proposed utilizing clock gating, operand isolation, memory segmentation and architectural modifications. Clock gating reduced power by 10 - 21%. Combined with operand isolation a reduction of 39 - 53% is obtained. Segmenting a memory of 256wx128b into 2 parts reduces memory power by 20% with 27% area increase. Architecture modifications like sharing register files among four functional units reducing power with 14 - 16% compared to a reference 4x4 architecture with 64 - 80mW at 100MHz and 619nJ - 1944uJ. On top of the optimizations (3) an extensive power analysis of different architectures has been performed. The proposed architecture with the evaluated optimizations including pipelining consumes 63.87 - 82mW at 312MHz and 307nJ - 702.7uJ when synthesized with 90nm Synopsys TSMC library (tcbn90ghptc).

Laboratory : Computer Engineering
Codenumber : CE-MS-2006-12

Committee Members :

Advisor:	Mladen Berekovic, CE, TU Delft & IMEC/DESICS
Advisor:	Georgi Gaydadjiev, CE, TU Delft
Chairperson:	Stamatis Vassiliadis, CE, TU Delft
Member:	Patrick Dewilde, CAS, TU Delft

Contents

List of Figures **ix**

List of Tables **xi**

Acknowledgements **xiii**

1 Introduction **1**
1.1 Background . . . 1
1.2 Scope of Document . . . 2
1.3 Document Structure . . . 3

2 The ADRES Architecture **5**
2.1 Top Level Processor Architecture . . . 6
2.2 ADRES Architecture Template . . . 7
2.2.1 XML Architectural Description . . . 7
2.2.2 ADRES Data Path . . . 9
2.2.3 Functional Units . . . 11
2.2.4 Registers Files . . . 15
2.2.5 Configuration Memories . . . 18
2.2.6 Modulo Scheduling . . . 19
2.3 Instruction Cache . . . 20
2.4 Data Memory Interface . . . 22
2.5 ADRES Base Architecture Selection . . . 23
2.5.1 Exploration Options . . . 23
2.5.2 Instances Selection . . . 25
2.6 Summary . . . 28

3 Exploration Methodology **33**
3.1 Tool Flow . . . 33
3.2 Optimizations . . . 35
3.3 Final Steps . . . 36
3.4 Summary . . . 36

4 Synthesis and Simulation Flow **37**
4.1 Synthesis Flow . . . 38
4.2 Simulation Flow . . . 40
4.2.1 Capturing Activity . . . 40
4.2.2 Power Estimation . . . 43
4.3 Summary . . . 44

5 Optimization Techniques **47**
5.1 Power Components . . . 47
5.2 Dynamic Optimizations . . . 50
5.2.1 Operand Isolation . . . 50
5.2.2 Clock gating . . . 52
5.3 Static Optimizations . . . 53
5.3.1 Pipelining . . . 54
5.3.2 Architectural Modifications . . . 54
5.3.3 Memory Segmentation . . . 64
5.4 Summary . . . 67

6 Detailed Architecture Results **69**
6.1 Library Selection . . . 69
6.1.1 Configuration Memories . . . 69
6.1.2 Register Files . . . 70
6.1.3 Functional Units . . . 74
6.2 Verification Environment . . . 76
6.2.1 Reference Architectures . . . 76
6.2.2 Benchmark Applications . . . 78
6.3 Clock Gating Results . . . 80
6.4 Operand Isolation Results . . . 80
6.5 Milestone Architectures . . . 83
6.5.1 Results . . . 83
6.6 Total ADRES Processor Power . . . 90
6.7 Comparison with LISATeK VLIW Architecture . . . 91
6.8 Summary . . . 92

7 Conclusions and Future Work **95**
7.1 Conclusions . . . 95
7.2 Future Work . . . 95

Bibliography **100**

A XML Architectural File **101**

B Instruction Set Architecture **105**

C Interconnection Options **107**

D Architectural Explorations **109**

E DRF Library Selection **111**

F FU Library Selection **117**

G Milestone Results **121**

List of Figures

1.1 Power efficiency vs. performance and flexibility of various architectures . . 1

2.1 Processor overview of ADRES 6
2.2 Overview of ADRES architecture 9
2.3 Data path example of an ADRES Instance 10
2.4 Example of Store Operation of a Byte 10
2.5 Operation Code Table 12
2.6 Functional unit instruction fields 13
2.7 Functional unit pipelined datapath in VLIW part 14
2.8 Configuration Field for CGA register file 16
2.9 Register Rotating 17
2.10 Forwarding logic in the Global Data Register File 18
2.11 Modulo Scheduling example on a 2x2 array 20
2.12 ADRESv1 Instruction Cache 21
2.13 ADRESv1 Data Memory Interface 22
2.14 Interconnection Options for Architectural Experiments 24
2.15 Area of First Architectural Exploration 26
2.16 Leakage Power of First Architectural Exploration @ 100MHz 26
2.17 IDCT Power consumption of First Architectural Exploration @ 100MHz . 27
2.18 FFT Power consumption of First Architectural Exploration @ 100MHz . . 27
2.19 Energy consumption of First Architectural Exploration @ 100MHz 28
2.20 IDCT Performance of First Architectural Exploration @ 100MHz 28
2.21 FFT Performance of First Architectural Exploration @ 100MHz 29
2.22 Area vs. Performance of First Architectural Exploration @ 100MHz . . . 29
2.23 Power vs. Frequency of First Architectural Exploration @ 100MHz 30
2.24 IDCT Energy-Delay of First Architectural Exploration @ 100MHz 30
2.25 FFT Energy-Delay of First Architectural Exploration @ 100MHz 30
2.26 ADRES Base Architecture 31

3.1 Global overview of Tool Flow 34

4.1 Simple representation of Tool Flow in Figure 3.1 37
4.2 Synthesis flow 39
4.3 Simulation flow 41
4.4 Capturing Activity 42
4.5 Transforming Esterel Toggle Data for PrimePower compatibility 42
4.6 Calculate Power 44

5.1 Percentages Power Components 4x4_reg_con_all 48
5.2 Power and Area distribution base architecture *4x4_reg_con_all* @ 100MHz 49
5.3 Example of External Optimizations 50
5.4 Operand Isolation 51
5.5 Difference Between Traditional and Clock Gated Register 53

5.6 Distributing the Local Data Register Files . . . 56
5.7 Interconnection Topologies . . . 57
5.8 Area of Second Architectural Exploration . . . 58
5.9 Leakage Power of Second Architectural Exploration @ 100MHz . . . 59
5.10 IDCT Power consumption of Second Architectural Exploration @ 100MHz 59
5.11 FFT Power consumption of Second Architectural Exploration @ 100MHz 60
5.12 Energy consumption of Second Architectural Exploration @ 100MHz . . . 60
5.13 IDCT Performance of Second Architectural Exploration @ 100MHz . . . 61
5.14 FFT Performance of Second Architectural Exploration @ 100MHz . . . 61
5.15 Area vs. Performance of Second Architectural Exploration @ 100MHz . . 62
5.16 Power vs. Frequency of Second Architectural Exploration @ 100MHz . . . 62
5.17 IDCT Energy-Delay of Second Architectural Exploration @ 100MHz . . . 63
5.18 FFT Energy-Delay of Second Architectural Exploration @ 100MHz . . . 63
5.19 Memory Segmentation to save Power . . . 65
5.20 Result of Memory Segmentations with MPEG2 configuration file at 250 MHz . . . 66

6.1 Register File Type selection for Configuration Memories . . . 71
6.2 CGA DRF differences between Synposis vs. Artisan for when accessing first port only . . . 72
6.3 CGA DRF differences between Synposis vs. Artisan . . . 72
6.4 Write power reduction with clock gating for VLIW DRF . . . 73
6.5 Read power reduction with clock gating for VLIW DRF . . . 73
6.6 Leakage power reduction with clock gating for VLIW DRF . . . 74
6.7 Power Results of Instructions for non-pipelined Functional Unit in CGA Section . . . 75
6.8 Glitch Power of a non-pipelined ALU in the CGA FU . . . 76
6.9 Power Results of Instructions for pipelined CGA FU . . . 77
6.10 ADRESv0 version for proof-of-concept . . . 78
6.11 ADRESv1 version . . . 79
6.12 Milestones Area . . . 84
6.13 Milestone IDCT Leakage . . . 85
6.14 Milestone IDCT Power . . . 86
6.15 Milestone IDCT Energy . . . 86
6.16 Milestone IDCT Performance . . . 87
6.17 Milestone IDCT Area vs. Performance . . . 87
6.18 Milestone IDCT Power vs. Frequency . . . 88
6.19 Milestones Energy-Delay . . . 89
6.20 Overview of ADRES Processor Power Consumption with IDCT . . . 91
6.21 4x4_reg_con_shared_morphosys_64G_4L ADRES Core Layout Views . . . 92

B.1 Operation Code Table . . . 105
B.2 Instruction Set Architecture ADRESv1 . . . 106

C.1 Interconnection Options for Architectural Experiments . . . 107

D.1 Distributing the Local Data Register Files 109
D.2 Interconnection Topologies 110

E.1 CGA DRF differences between Synposis vs. Artisan for when accessing all ports 111
E.2 CGA DRF differences between Synposis vs. Artisan for when accessing first port only 111
E.3 CGA DRF differences between Synposis vs. Artisan 112
E.4 VLIW DRF differences between Synposis vs. Artisan for when accessing all ports 113
E.5 VLIW DRF differences between Synposis vs. Artisan for when accessing first port only 114
E.6 VLIW DRF differences between Synposis vs. Artisan 115
E.7 VLIW DRF differences between no optimizations and clock gating with Synposis libraries when accessing first port only 116

F.1 Power Results of Instructions for non-pipelined Functional Unit in CGA Section 117
F.2 Power Results of Instructions for non-pipelined ALU in FU CGA Section 118
F.3 Power Results of Instructions for pipelined Functional Unit in CGA Section 119
F.4 Power Results of Instructions for pipelined ALU in FU CGA Section . . . 120

G.1 Milestones Area 121
G.2 Milestones Leakage 122
G.3 Milestones Power 123
G.4 Milestones Energy 124
G.5 Milestones Performance 125
G.6 Milestones Area vs. Performance 126
G.7 Milestones Power vs. Frequency 127
G.8 Milestones Energy-Delay 128

List of Tables

2.1 Functional units operation types 11
2.2 Pipeline stage for VLIW FU with highest latency 15
2.3 Data Memory Interface execution stages 23
2.4 Architectural Exploration Selection Options 25

4.1 Differences between Esterel and ModelSim for ADRESv0 43

5.1 Renaming Architectures of Second Exploration 55
5.2 Differences between 4x4_reg_con_all and arch_8 for IDCT and FFT 64
5.3 Reducing Register File Size arch_8 64

6.1 CGA FU Area Results between with and without Operand Isolation with Synopsys Libraries 76
6.2 MPEG2 Frame Power Results 79
6.3 Clock Gating improvements for ADRESv0 80
6.4 Clock Gating improvements for ADRESv1 81
6.5 Operand Isolation + Clock Gating Improvements for ADRESv0 82
6.6 Operand Isolation + Clock Gating Improvements for ADRESv1 82
6.7 Milestones Frequencies 83
6.8 Comparing base with final architecture 90
6.9 ADRES processor *4x4_reg_con_shared_morphosys_64G_4L* at 312MHz . . . 90
6.10 FFT Figures of VLIW instance from [39] and final 4x4 ADRES instance . 92

F.1 CGA FU Area Results between Artisan and Synopsys Libraries 117
F.2 CGA FU Area Results between without and with Operand Isolation with Synopsys Libraries 119

Acknowledgements

This thesis project was dynamic and interesting at which I have worked on with great pleasure at the DESICS division at IMEC in Leuven. Mladen Berekovic motivated me to explore new areas of research and professionally guided me during my thesis and writing. He was one of the reasons why I did not give up in the most difficult times for which I am very grateful. I would like to thank Andreas Kanstein for his excellent work at modifying the Esterel simulator for me and enduring patience for all my questions. He basically made it possible this thesis exists. I would also like to thank Georgi Gaydadjiev for his professional assistance during the thesis period and writing and his understanding in difficult periods. Special thanks also go to Bjorn de Sutter, Allam Osman, Merlijn Aurich, Veerle de Rudder and for those I did not mention I would like to extend my thanks to the entire ADRES team for their help during the coarse of the project.

My biggest gratitude goes to my mother and father for their love and support during all my life as well as in this thesis period. They showed interest in my project and motivated me to go forward. My mother left this world after a courageous battle with her illness on April, 3^{rd} 2006, which made this period even more difficult for me and my father. She showed me that nothing in the world is more important than life itself and for that alone I have the upmost respect for her. I will love and remember her till the end of days as the mother I am proud of to be her son. I have always been proud of my father and to him, my friends and colleagues I can only say: *In difficult times like this the true nature of a person and its friends comes to light.* Thank you all for your support.

Frank Bouwens
Delft, The Netherlands
August 10, 2006

1 Introduction

1.1 Background

The requirements for high performance and low power consumption are becoming more stringent every day when designing devices; especially for multi-mode multimedia. New chip architectures should be able to execute multiple applications delivering high performance, while maintaining low power consumption, area, non-recurring engineering costs and shorter time-to-market. Figure 1.1 depicts several architectures with their power efficiency and performance values.

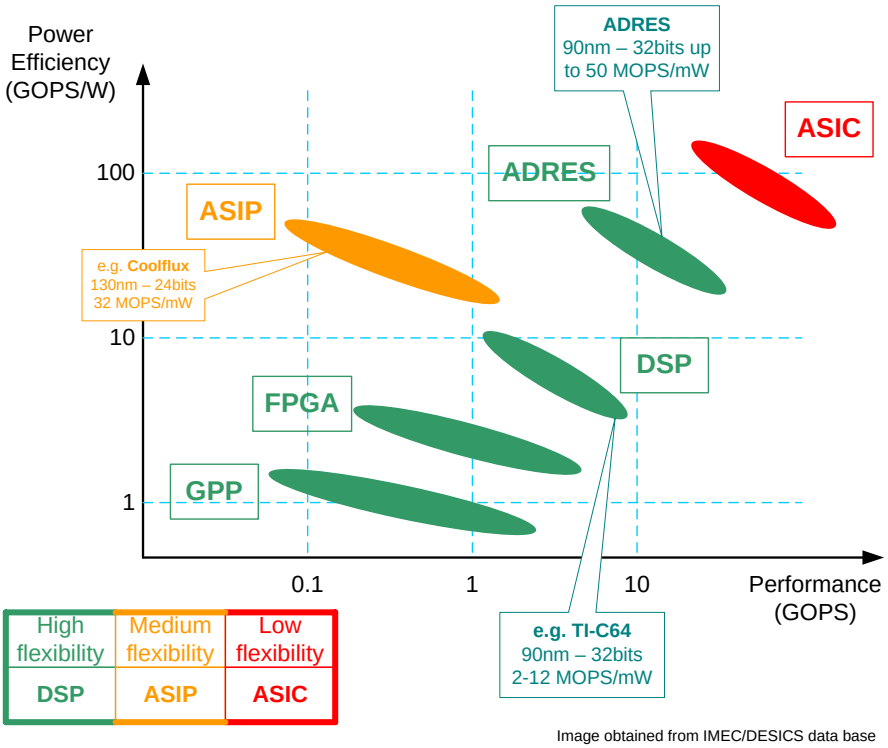


Figure 1.1: Power efficiency vs. performance and flexibility of various architectures

A General Purpose Processor (GPP) is very flexible for configurations and usually targets computers were performance has a higher priority than power consumption itself. The power efficiency is therefore low and unsuitable for embedded systems. On the other hand, Application Specific IC (ASIC) has very high performance and power efficiency, but has low flexibility in configuration. An Application Specific Instruction set Processor (ASIP) has the same performance as a GPP, but with a higher power efficiency. Digital

Signal Processors (DSP) are an intermediate solution between GPP and ASIC targeting signal processing applications e.g. FFT and IDCT. DSPs focus on power efficiency and performance with good flexibility. Field Programmable Gate Arrays (FPGA) are more flexible due to their reconfigurable possibilities, but are slow compared to DSPs. Additionally, FPGAs consume a vast amount of energy making them unsuitable for mobile devices as noted by Binfeng Mei [27]. Unlike the FPGAs and other medium of highly flexible architectures, Coarse-Grained Reconfigurable Arrays (CGRA) have high performance and power efficiency. ADRES is a CGRA which targets 50MOPS/mW with 90nm technology reducing power to the minimum and is still flexible due to its reconfigurable possibilities.

IMEC develops an *Architecture for Dynamically Reconfigurable Embedded Systems* (ADRES) [28] consisting of a Very Large Instruction Word (VLIW) processor with a reconfigurable array tightly coupled with it. The VLIW controls the array and speeds-up the program's execution with Instruction Level Parallelism (ILP), while the array does the same with Loop Level Parallelism (LLP). The ADRES architecture is a template allowing designers to specify the specific interconnections, type and the number of Functional Units (FU) and register files (RF). Compiling applications for ADRES is performed with *Dynamically Reconfigurable Embedded System Compiler* (DRESC) re-targeting regular ANSI-C code for the architecture of choice. Each application can be tested at symbolic, instruction and clock cycles accurate level.

The target of this thesis is to estimate the performance and power consumption based on benchmark applications executed on a basis ADRES core instance and optimize the architecture balancing performance and power consumption. A simulation, synthesis and analysis tool chain flow is setup for performance and power evaluating the decisions. Based on preliminary findings the architecture is optimized in both power and/or performance where this is required. Synthesizing an ADRES instance is performed with 90nm TSMC technology [20]. The most appropriate library from Artisan or Synopsys of this technology is selected based on performance, power and area.

1.2 Scope of Document

The ADRES processor consists of the CGRA, data memory and instruction cache. Each component can be improved in both power and performance, however, this thesis only focusses on the core architecture for optimizations. A non-pipelined version of ADRES is utilized during the coarse of this thesis project for evaluation and comparison. A pipelined version was only available in the last stages of the thesis and is implemented in the final optimized architecture. The data memory and instruction cache were not available during the synthesis procedure. Any data noted in this thesis about these two components are based on information found in data sheets.

Optimizations applied to ADRES instances are for both performance and power of which the latter has higher priority. Static and dynamic optimizations are utilized and either verified with the aid of the synthesis and simulation tool chain or by data sheets.

The logic synthesis of a chip to determine a.o. power consumption is done by syn-

thesizing the HDL code with a the synthesis tool Physical Compiler of the Synopsys tool chain [19] using 90nm TSMC libraries. The final architecture is placed and routed to obtain capacitance and resistance values to improve accuracy of the simulation results.

Power estimations are done by PrimePower annotating switching activity of the ADRES nets on the synthesized design. For accurate power estimations a gate level simulation of the ADRES architecture has to be performed, however, the simulations did not work. Of the non-pipelined version of ADRES there was only one instance working correctly at RT level simulations. Since different instances should be compared a different approach had to be followed. A cycle-true instruction set simulator (ISS) [12] is modified to obtain switching activity of the nets similar to simulations at register transfer (RT) level by the clock cycle accurate simulator ModelSim [18]. The activity values of the simulations are annotated on the gate level netlist of an ADRES instance. Chapter 4 will prove the reliability of the instruction cycle accurate methodology compared to RT level simulations.

For validation and verification of an ADRES instance we used an Inverse Discrete Cosine Transformation (IDCT), a Fast Fourier Transformation (FFT) and a multi-media application (MPEG2) of which the first two are primarily used for rapid testing, selection of architectures and debugging. MPEG2 itself is used when significant optimizations (called milestones) are applied to the base architecture. The benchmark applications results of the final architecture are utilized for comparison with the selected base architecture.

1.3 Document Structure

To get a better understanding of the ADRES architectural template, its modules and possibilities are explained in Chapter 2. Modifying the DRESC compiler is out of the scope of this thesis, however, a small explanation of the scheduler for ADRES is provided. The instructions cache and data memory included in the pipelined version of ADRES are also explained. Using the non-pipelined ADRESv0 several architectures are generated to empirically select an appropriate base ADRES architecture based on characteristics such as power, performance and energy.

Chapter 3 describes the exploration methodology of this thesis based on the possibilities of the selected architecture, the benchmark applications and the available designer tools.

High and low level optimization techniques are described in Chapter 5 and evaluated in terms of power, performance, energy and area. The optimizations are later implemented in the final design.

In Chapter 4, a flow is created to synthesize each architecture and determine their physical characteristics and power consumptions. This requires a synthesis and simulation flow for evaluation utilizing the available programs and 90nm TSMC libraries.

The results of the optimizations are combined in the final design in Chapter 6 to depict the optimization improvements and compare with the base ADRES architecture selected in Chapter 2. In addition, there are two different vendors of 90nm TSMC technology, which are explored for their influence on performance and power. The most appropriate library of Artisan or Synopsys is selected as described in Section 6.1. Finally,

the results of the final architecture are provided and possible improvements for future investigation.

2 The ADRES Architecture

According to Amdahl's law overall application speedup can be achieved by enhancing the parallel fraction in the code as best as possible as noted by Hennessy et al [10] and in the speedup formula below. This speedup can be achieved by looking at the part in the code that can be parallelized *($Fraction_{enhanced}$)* and requires most of the time during execution. A widely held rule of thumb according to Hennessy et al. [10] is that 90% of the execution time is spent on 10% of the code, which are generally loop iterations. Accelerating the parallelized fraction could increase the overall speedup significantly. If the $Speedup_{enhanced}$ is equal to the number of processors in the architecture the overall speedup would theoretically become $1/(1 - Fraction_{enhanced})$ with an infinite number of processors. However, this is a fallacy, since adding more processors would introduce additional communication overhead until no more enhancement is possible.

$$Speedup_{overall} = \frac{1}{(1-Fraction_{enhanced})+\left(\frac{Fraction_{enhanced}}{Speedup_{enhanced}}\right)}$$

Acceleration can be done a.o. by instruction (ILP) and loop-level parallelism (LLP) or better called as software pipelining. The *Architecture for Dynamically Reconfigurable Embedded Systems* (ADRES) [28] provides support for ILP and LLP acceleration with a VLIW processor and a coarse-grained reconfigurable array, respectively. The pipelined VLIW is comparable with the Trimedia architecture of which an extensive explanation is given by Hennessy et al. [10] and Van de Waerdt [43]. This pipelined processor has five operation slots of 32-bit instructions operating in parallel. The Trimedia architecture is an example of the ILP success in enhancing performance and power (0.649 mW/MHz at 1V for TM3270) in the embedded and mobile markets.

ADRES basically extends the VLIW with a coarse-grained array (CGA) of functional units. This array enhances performance by focussing on LLP, while the VLIW part utilizes ILP and controls the array. The regular VLIW processors use a large global data register file, which can become expensive to build. As noted by Bingfeng Mei [27], for N functional units (FU) the power consumption and area of the register file increases by N^3 and delay is increased by $N^{2/3}$. By clustering the register file these problems are avoided. The FUs in the array are connected to small register files based on the clustering principles. By using the array for loops expensive accesses to the large global register files is avoided by using the smaller register files locally. These features combined ensure a high-speed, low-power architecture as shown in this thesis.

This chapter first describes the overall top level processor architecture followed by the description of the ADRES template in detail with all its components and interconnections. The instruction cache and data memory interface in Sections 2.3 and 2.4 are shortly described for completeness of the discussion, but are not a target for optimizations. Following, an appropriate architecture out of 14 different exploration architectures

has to be selected as a basis for further optimizations as described in Section 2.5. The exploration architectures are verified by the benchmark applications Inverse Discrete Cosine Transformation (IDCT) and Fast Fourier Transformation (FFT) of which power and performance figures are obtained.

2.1 Top Level Processor Architecture

The ADRES processor consists out of the ADRES core, instructions cache (ICache), data memory (DMEM) and an Advanced High-Speed Bus (AHB) [14] interface as depicted in Figure 2.1. The processor has the ability to operate stand alone or as part of a System-on-Chip design.

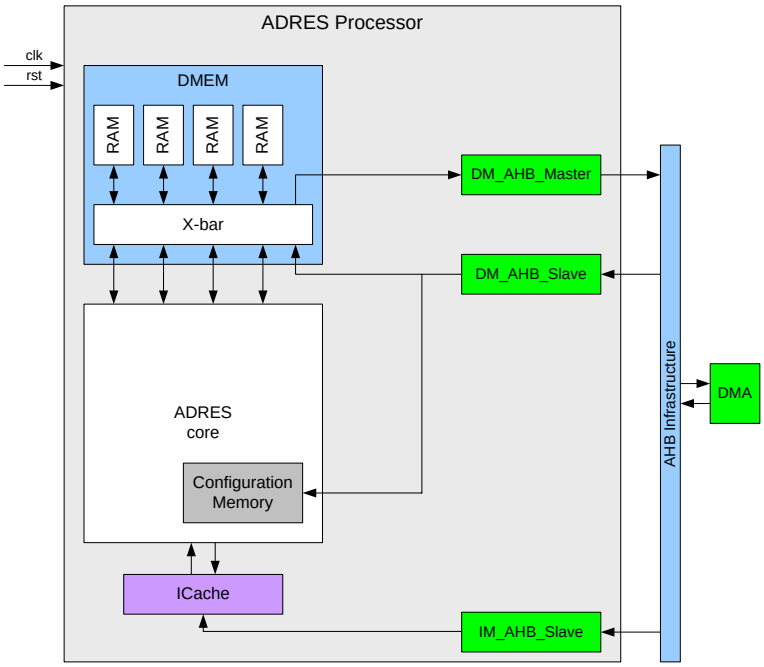


Figure 2.1: Processor overview of ADRES

Attached to the ADRES core are the data memory and instruction cache, which are both scalable in size. The synchronous DMEM is utilized by the core as a scratch pad and this also functions as a shared communication medium between the VLIW and Coarse Grained Array (CGA) modes of operation of ADRES. Both DMEM and ICache contents can be written externally through their respective AHB Slave interfaces (DM_AHB_Slave and IM_AHB_Slave). Reading the DMEM contents externally is also done by the AHB Slave interface. The AHB Master interface (DM_AHB_Master) accesses global AHB addresses by memory load instructions. The Direct Memory Access (DMA) controller makes it possible to access global data memory without interrupting the processor. The configuration memories (CM) used in the core contain the configurations for the array during CGA mode. The CMs have their contents loaded via the AHB Slave interface as well. The next sections describe the components of the processor in more detail.

2.2 ADRES Architecture Template

The coarse-grained reconfigurable architecture ADRES is not an actual fixed IP block, but rather a template described in a XML based modeling language that can be utilized by an engineer to created its own design. The ADRES architecture can have an arbitrary dimension with any shape making it very versatile as noted by Kwok [24] and Bingfeng Mei [27]. The XML architecture file is utilized by the compiler for proper scheduling in and is the basis to create a top-level VHDL architecture file as will be explained in Chapters 3 and 4.

2.2.1 XML Architectural Description

The XML architectural file consists of three major sections: *resources*, *connections* and *behaviour* of which an elaborate example is depicted in Appendix A.

The *resources* section in the XML document describes among others the FUs, register files and Transistion Nodes (TRN) to construct the ADRES core of which their simple forms are depicted in Figure 2.2(b).

- The functional units (<FU>) can have their inputs, outputs and instruction groups altered to create different FUs in the array. The configuration of the FUs is very flexible, however, only one FU is allowed to branch and execute the CGA command to enter CGA mode.

- The register files (<RF>) can either be predicate or data register files and also for the VLIW and/or CGA section. The global register files *vliw_int_rf* and *vliw_pred* are compulsory and have their number of ports, port width and sizes defined. The *nonrotsize* setting defines the size of the fixed size of the register file of which the usage will be explained in Section 2.2.4. The local *ireg_X* and *pred_X* register files (where X can be any number) are distributed on the array and are arbitrary in the architecture. These are preferably rotating due to modulo scheduling [27].

- The constant values (<CONST>) contain immediate values for the application in CGA mode and come from the configuration memories (CM) in Figure 2.2(b).

- The Transition Nodes (<TRN>) are utilized for creating connections between the functional units, register files and constant values. These are multiplexors and can also obtain a delay value creating registers at the output when this value is larger than zero. The size of the TRN depends on the connections connected to it. The compulsory ports *loop_start* and *loop_stop* are required to start and stop CGA mode of ADRES. The loop_start signal goes from the control unit to the FUs, while the loop_stop signals from the FUs in the array go to the VLIW control unit (VLIW_CU).

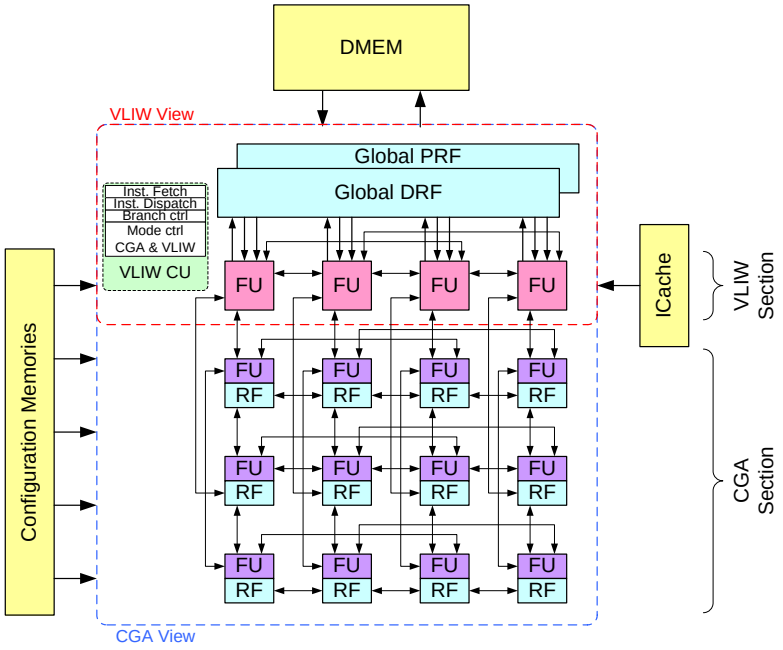
The *connections* section describes how the resources are connected with each other. Each connection has a source (*src*) and destination (*dst*). As an example, the connections section in Appendix A is depicted in in Listings 2.1.

Listing 2.1: Connections Example

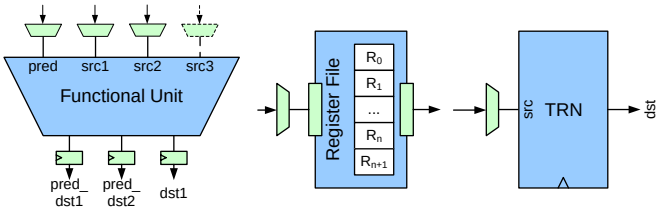
```
<connection>
    <connect> <!-- Connection 1 -->
        <src entity = "const_0" />
        <dst entity = "fu_0" port = "src1"/>
    </connect>
    <connect> <!-- Connection 2 -->
        <src entity = "fu_0" port = "dst1" />
        <dst entity = "ireg_0" port = "in1" />
    </connect>
    <connect> <!-- Connection 3 -->
        <src entity = "ireg_0" port = "out1" />
        <dst entity = "fu_0" port = "src1" />
    </connect>
    <connect> <!-- Connection 4 -->
        <src entity = "ireg_0" port = "out2" />
        <dst entity = "fu_0" port = "src2" />
    </connect>
    <connect> <!-- Connection 5 -->
        <src entity = "fu_0" port = "dst1" />
        <dst entity = "outireg1_0" />
    </connect>
    <connect> <!-- Connection 6 -->
        <src entity = "outireg1_0" />
        <dst entity = "fu_1" port = "src2" />
    </connect>
</connection>
```

Connection 1 links the constant value *const_0* to the first source port of *fu_0*. The output of fu_0 is connected to the input of local RF *ireg_0* with connection 2, while connections 3 and 4 link the RF read ports to the source ports of the FU. Connection 5 connects the output of *fu_0* to the TRN *outireg1_0*. Its output can be linked with the input of *fu_1* as created by connection 6. This procedure is iterated until the designer has connected all necessary resources with each other.

The *behaviour* section defines the instruction sets available for the architecture and the behavior of the functional units. The *vliw_section* describes which components and signals are for the VLIW section and operation. The FUs in *vliw_fus* are designated to operate in both VLIW mode and in CGA mode. The intrinsics for the FUs are defined in *intr_op* as will be placed in the instruction sets. The *op_section* contain all the instruction sets of the ADRES architecture. Each instruction set contains its own instructions with the same delay. For example, the *logic* set only requires one delay cycle in a FU, while the *ldmem* group requires 4 or 6 delay cycles depending if the data memory queue is utilized or not (Section 2.4).



(a) ADRES Instance example of a 4x4 array



(b) ADRES XML components

Figure 2.2: Overview of ADRES architecture

2.2.2 ADRES Data Path

The ADRES architecture is a tightly-coupled architecture between two views: VLIW and CGA. Note the difference between CGA view and CGA section, since the CGA view is a combination of the functional units of both VLIW and CGA section. The VLIW control unit directs the switching between VLIW and CGA modes. It also starts and stops the CGA loops when initiated by a VLIW FU, accesses the instruction cache for the next instruction for the VLIW section in the respective mode and calculates the next Program Counter (PC) (if there are no branches made by a VLIW FU).

In the example depicted in Figure 2.2(a) the VLIW section has 4 FUs and the CGA section has 4 by 3 FUs, however the size of the ADRES architecture instance can be arbitrary as noted in Section 2.2.1. The FUs in the VLIW view communicate through the a multi-port global Data Register File (DRF), while those in the CGA view communicate through the available interconnections.

An example of a data path in ADRES is depicted in Figure 2.3. The explicit pipeline

registers ensure timing of the data path. The local Data and Predicate Register Files (PRF) are use for storage of variables and removal of branches in pipeline-able loops. As part of the template's flexibility the local register files do not have to be present or could be shared among several FUs.

When the data path is operating in CGA mode the available FUs, local RFs and TRNs have to be configured by the configuration memories. These hold the instructions and addresses for the FUs and RFs, respectively, and routing information for the multiplexors to select the appropriate input. A different configuration is loaded on each instruction cycle during CGA mode. In VLIW mode the configuration memories are kept idle.

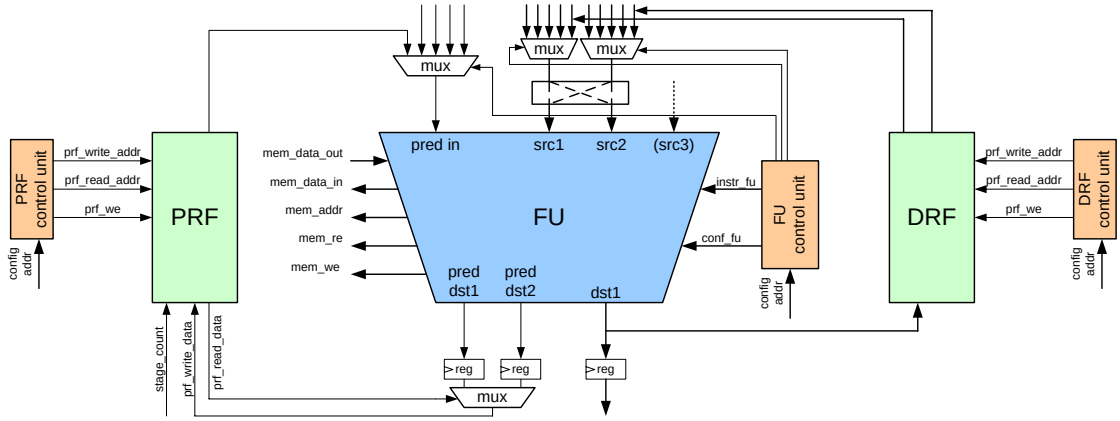


Figure 2.3: Data path example of an ADRES Instance

All functional units have 1 write port and 3 read ports at most. The third source port is only present when the FU is configured as a store unit for external access. An example of an store operation is depicted in Figure 2.4.

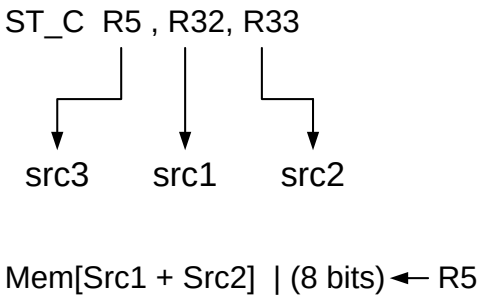


Figure 2.4: Example of Store Operation of a Byte

Source 1 of the FU holds to the base address of the memory located in register R32. The offset is either an immediate value or obtained from a register in source 2. In the example register R33 is used instead of an immediate value. Source 3 contains the value to be stored as is located in register R5. When the FU does not have store capabilities it only has 2 source port, which results in a 2:1 ratio for read and write ports. The

instruction set architecture (ISA) operates in a register-to-register way requiring the register file connected to the FU to have a 2:1 ratio for read and write ports. The source 3 port in a FU requires an additional read port in a register file or it can be shared with another register file read port.

2.2.3 Functional Units

There are two types of functional units in the ADRES architecture: one for the VLIW and one for the CGA section as depicted in Figure 2.2(a). The main difference between them is in the ISA support where the VLIW FU has additional capabilities to perform e.g. branch operations and subroutine call operations. Additionally, the pipelined ADRESv1 has different pipeline schemes for both types of which the VLIW FU has more pipeline stages than a CGA FU. With the non-pipelined ADRESv0 the VLIW FUs can operate in both modes, however, the VLIW FUs of ADRESv1 do not support different pipeline latencies. Therefore, the FUs of ADRESv1 in the XML architecture file (Section 2.2.1) designated to be VLIW FUs are duplicated, creating a CGA and VLIW FU. More details on this will be provided later in this section. In both versions of ADRES the CGA FUs can only operate in CGA mode and are idle in VLIW mode.

Instruction Set Architecture The ISA is a generic RISC instruction-set desired for IMPACT's intermediate code, LCode [17] of which a small explanation is provided in Section 3.1. During the coarse of the thesis a non-pipelined functional unit of ADRESv0 evolved to a pipelined version ADRESv1. These had different instruction sets, but only the latest version will be presented here. The ADRESv1 version has 8 different instruction sets as depicted in Table 2.1. The instructions in the instruction sets are depicted in Figure 2.5. According to Hennessy et al. [10] an instruction set holds consecutive instructions with no register data dependencies, which mean they can be executed in parallel, if hardware resources are available and dependencies through memory are preserved.

Table 2.1: Functional units operation types

Operation Type	Description
ARITH_1	Basic arithmetic operations
ARITH_2	Regular compare instructions
ARITH_X1	First intrinsic operations field
ARITH_X2	Second intrinsic operations field
LOGIC	Shift and boolean operations
PRED	Predicate compare instructions
LDST	Load and Store operation from memory to DRF and vice versa
SPECIAL	Branch and Jump, Control Operation mode, halt and special register instructions

The ARITH_1 set performs basic operations e.g. MOV, NOP and multiplication instructions. The ARITH_2 set contains the comparison instructions of which the predicated versions are located in the PRED set. The LOGIC instruction set contains modulo, logical shift and boolean operations. The SPECIAL set holds instructions that can only

be executed in VLIW mode e.g. branches and special VLIW register file instructions. A control instructions to switch into CGA mode is also in this group, hence it is not supported in the CGA FUs. The LDST set contains the load and store instructions. FU specific instructions called *intrinsics* are placed in the ARITH_X1 and ARITH_X2 fields. Intrinsics collapse several lines of code into a single instruction [9]. They are very effective at speeding up computations and consume less power compared to the individual instructions, since additional decoding operations and operand fetching are avoided. The intrinsics are implemented in the ALU and only require the ANSI-C source code to be adjusted.

Instruction Set

Instructions	ARITH_1 100	ARITH_2 101	ARITH_X1 110	ARITH_X2 111	LOGIC 001	PRED 011	LDST 010	SPECIAL (*) 000
0000	MOV	GT	SUBABS	CLIP1	LSL	PRED_GT	LD_UC	SETLO_0
0001	NOP	GT_U	INNERSUM	CLIP2		PRED_GT_U	LD_C	JMP_BR_0
0010	SHRMB	GE	AVGU4	MIN	LSR	PRED_GE	LD_UC2	SETHI_0
0011	RPHI (*)	GE_U	ADD2	MIN_U		PRED_GE_U	LD_C2H	MV2SR
0100	SHLMB	EQ	SUB2	MAX	ASR	PRED_EQ	LD_C2	SETLO_1
0101	PACK2		AVG_E	MAX_U			LD_I	JMPL_BRL_0
0110	PHI (*)	NE	SH_RND		OR	PRED_NE	LD_I2	SETHI_1
0111	SPACK2		ADD		AND			HALT
1000	MUL	LT	SUB		XOR	PRED_LT	ST_C	SETLO_2
1001	MUL_U	LT_U			NOR	PRED_LT_U	ST_C2	JMP_BR_1
1010	SPACKU4	LE			NAND	PRED_LE	ST_C2H	SETHI_2
1011	MUL_SU	LE_U			NXOR	PRED_LE_U	ST_C2R	MVFSR
1100						PRED_SET	ST_I	SETLO_3
1101						PRED_CLEAR	ST_I2	JMPL_BRL_1
1110								SETHI_3
1111					MODULO			CGA

(*) Not supported in CGA mode

Figure 2.5: Operation Code Table

Instruction Fields An example of the instruction fields for the VLIW and CGA FUs are depicted in Figure 2.6. A complete overview of all the possibilities for the VLIW instruction field and the complete ISA can be found in Appendix B. The VLIW instruction field is fixed at 32-bits wide where the CGA configuration field can be variable as noted later. There are three bits for the operation types (or instruction set as previously called) and 4 bits for the 16 instructions in the group as depicted in Figure 2.5. In the VLIW section the *Pred*, *Dest* and *Src* fields point to register addresses for operation. The CGA configuration field does not have a destination value. Instead, the output of an operation is routed to its destination by the configuration line for the register file as will be explained in Section 2.2.4. The sources for the CGA FU are selected by the port selector field in the CGA configuration field. Multiple connections can be connected to the FU source as is described by the *connections* section in the XML architecture file (Section 2.2.1). These connections are routed by a multiplexor of which its sizes depends on the number of connections to it.

The *Pred* field saves the output of an operation in one of the 32-bit predicate registers. In VLIW mode any instruction can be predicated with the predicate registers, however

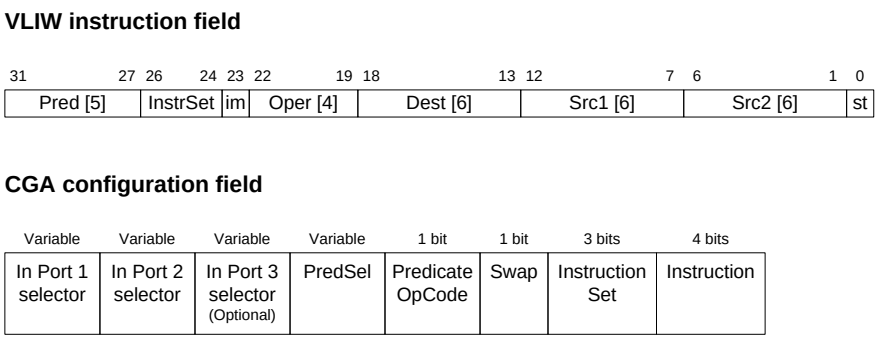


Figure 2.6: Functional unit instruction fields

in CGA mode the predicate input for a FU is selected by the *PredSel* field. To ensure the predicate outputs are generated at the *pred_dst1* and *pred_dst2* outputs in Figure 2.3 the Predicate Opcode bit has to be set in the CGA configuration field. The predicate instructions also have the ability to remove epilogue and prologue when going into CGA mode as will be explained in Section 2.2.6. A predicate output is routed as well by the configuration line for the register file.

The *immediate* bit replaces one or more *src* values in the instruction field by an immediate value. This bit is not available in the CGA instruction field, however, immediate values from the constant memories as depicted in Figure 2.2(b) are routed by the multiplexors to their FU's source inputs. When the source multiplexors to the CGA FU have different inputs the compiler might not be able to route the instruction operands to the correct source with non-commutative operations ($a-b \neq b-a$). By setting the *swap* bit the source inputs are swapped and the operands are routed to the correct source input. The stop bit (st) in the VLIW field is utilized to decompress the instruction line in the ICache after NOP compression as will be explained in Section 2.3.

Pipeline Stages This section primarily applies to the pipelined version of the FUs in ADRESv1, however, when removing all of the registers between the pipeline stages the non-pipelined version is recreated. The entire process of fetching, decoding, execution and write back has a latency of a single clock cycle in ADRESv0. This makes the relatively slow data memories a significant bottleneck in performance, which is the main reason to apply pipelining. The FUs have different pipeline schemes for VLIW and CGA mode of which the differences will be explained later.

The VLIW FU data path depicted in Figure 2.7 is similar to that of the MIPS architecture [10] of which the amount of pipeline stages varies between 5 and 10 depending on the instruction executed and the functional unit type. A small description of all the pipeline stages for a FU in the VLIW section is noted in Table 2.2. The FUs in the VLIW have 5 - 10 stages depending on the instruction, while those in the CGA section only have 3 - 8 stages. The CGA FU have a simple data-flow without the fetch2 and decode stage requiring the compiler to explicitly create the operands for the CGA FU.

The instructions in the ICache are compressed by NOP compression to pack the VLIW instructions in the cache and prevent switching activity. In the CGA FUs there

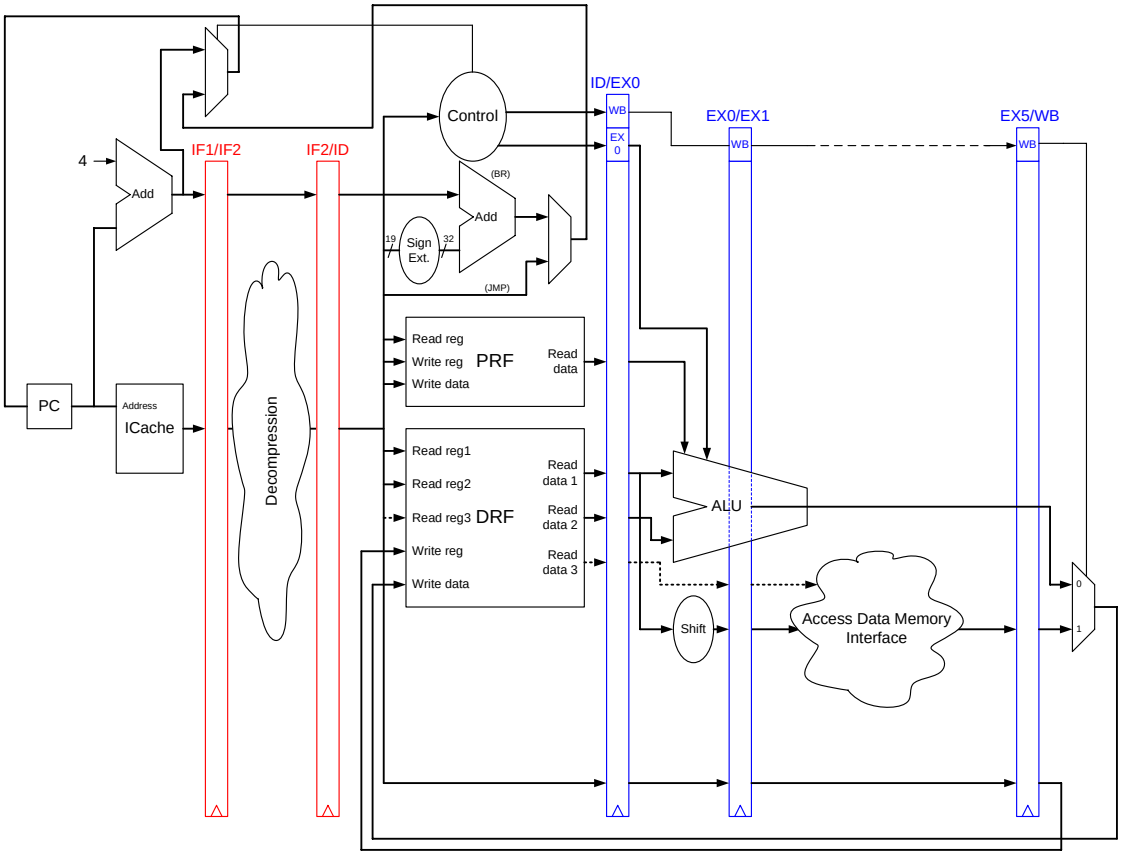


Figure 2.7: Functional unit pipelined datapath in VLIW part

is no compression, hence the Fetch2 stage can be omitted. Fetching values from registers in the instruction decoding (ID) stage differs as well. This is done automatically every clock cycle in CGA mode omitting the ID stage entirely for the CGA FU. The compiler places the operand explicitly during compilation at the correct source input of the FU.

Pipeline Stage Operations The operations of the pipeline stages apply to the VLIW FU as depicted in Figure 2.7. It starts by fetching $N \times 32$-bit instruction words in the instruction cache in the instruction fetch1 (IF1) stage. Where N is the amount of VLIW FUs. In the second fetch (IF2) stage the instruction line is decompressed by placing NOP instructions after the instruction in the line that has its stop bit set. Since this is done in the instruction cache more information can be found in Section 2.3. The VLIW FUs' execution stages become idle when obtaining a NOP instruction.

Instruction decoding in the *ID* stage obtains predicate and data results to be used for calculation. This stage additionally calculates the next program counter value. When a jump or branch instruction is decoded the appropriate new program counter (PC) value is selected by the multiplexor and send to the VLIW CU (Figure 2.2(a)).

The execution stages vary from 1 to 6 cycles depending on the instruction as noted in Table 2.2. These delays are the same as noted in the XML architecture file, since the

Table 2.2: Pipeline stage for VLIW FU with highest latency

Stage		Description
Fetch1	(IF1)	Read instruction cache (n x 32-bits + 32-bits immediate)
Fetch2	(IF2)	Decompress fetched instructions in cache
Decode & Read	(ID)	Decode n instructions in parallel and read data and predicate register files. Adjust PC in case a JMP or Branch instruction is fetched
Execute0	(EX0)	Arithmetic operations e.g. ADD, SUB and intrinsics. First stage of MUL and address modifications for Load operations
Execute1	(EX1)	Last stage of MUL operation or sending address to the corresponding queue in the DMEM interface
Execute2	(EX2)	Send most timing critical request to memory
Execute3	(EX3)	Reading data from memory for the Load instruction
Execute4	(EX4)	Send data to the reorder buffer
Execute5	(EX5)	Send data back from DMEM interface to the ADRES core
Write Back	(WB)	Write the results to the data register file

compiler requires these value for proper scheduling. A regular instruction with a delay of one is executed in the first stage followed by an immediate storage in the register file in the write back (WB) stage. Multiplications require two stages. A load operation normally requires 6 stages, but this can be reduced to 4 if the memory queue is bypassed omitting Execute stages 1 and 4 in Table 2.2. Reducing the memory queue latency can only be done when no data memory conflicts can occur. The store instruction requires a single clock cycle, since only the address and data have to be placed on the busses. While the processor continues with other operations the data memory interface takes care of the data storage.

The problem with such variable pipeline stages is the in-order completion and structural hazards when e.g. a load instruction returns its value while an arithmetic operation returns its result. By default the load instruction has higher priority over the multiplication operation followed by other instruction as depicted in Figure 2.5. The scheduler is also expected to avoid a situation like this during compilation.

Functional Unit Forwarding The forwarding logic in [10] avoids data dependency hazards and is implemented in the functional unit. With ADRES, however, it is implemented in the global data register file or the compiler uses a routing resource from the FU output to input in the CGA section. This forwarding mechanism avoids an additional clock cycle if the data written in the Write Back stage to the register file is required immediately for another instruction.

2.2.4 Registers Files

There are two different register files responsible for the correct operation of ADRES: data and predicate register files. These can be subdivided for the VLIW and the CGA sections giving global and local register files, respectively. The register files can have

a fixed and register rotating section. Register rotating is a register renaming operation used during software pipelining as explained in more detail later in this section. Utilizing the XML architecture file as basis to generate the VHDL architecture file, the XML file describes how the functional units are connected to the register files as well as their sizes (Section 2.2.1). The depth of the global register files can be arbitrarily, but for compiler friendly operation and optimal performance in array mode a fixed and rotating part for register rotating should be available. Only during CGA mode both sections are accessible, while in VLIW mode only the fixed part can be accessed. The sizes of the local register files can also be arbitrary and can be either rotating or fixed. Rotating is preferable for compiler friendliness and is an essential part for modulo scheduling [30]. Section 2.2.6 will explain the basics of modulo scheduling. Unlike the global register files the local register files are not required in the architecture for correct operation and can even be replaced by busses reducing area and power consumption.

The 32-bit wide data register files store data and can serve as communication medium for functional units. The 1-bit predicate register files are used for predicate functions and remove branches from pipeline-able loops [31], but are also essential to remove feedback operations, prologue and epilogue as noted by Bingfeng Mei et al. [29]. The latter two occur when a loop starts and stops and the array is not completely occupied as will be explained in Section 2.2.6.

Instruction Fields The local DRFs and PRFs have to be controlled during CGA mode. Just as with the functional units this is done by a register file configuration field as depicted in Figure 2.8. The width of the register file configuration word depends on the number of write and read ports, multiplexor size and the size of the register file. The fields contain the selection bits for the register files input multiplexors and the read and write addresses for the local register files in CGA mode. This omits the decoded stage in the CGA FU as noted in Section 2.2.3.

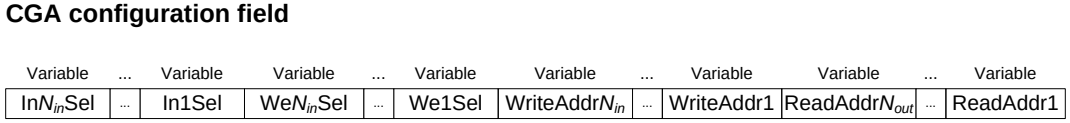


Figure 2.8: Configuration Field for CGA register file

The configuration field contains the read ($ReadAddrN_{out}$) and write addresses ($WriteAddrN_{in}$) used to fetch the operands for the FU. Multiple connections can be connected to a register file write port of which the correct one is selected by the $InN_{in}Sel$ signal. Writing the values to the register file requires a write enable of which the appropriate one is selected by the WeN_{in} signal.

Register Rotating The registers can be overwritten by other loops while they are still used with modulo scheduling. A register renaming type called *register rotating* [27] is utilized to ease the allocation of registers with software pipelining. This method is hardware based of which a simple schematic is depicted in Figure 2.9. This register file has a rotating and fixed section of which the fixed section always comes first.

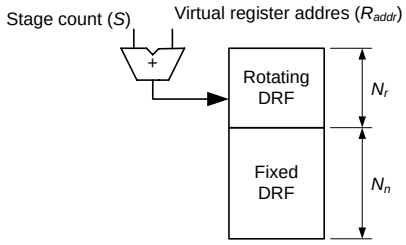


Figure 2.9: Register Rotating

The address for the rotating register (R_{rot_addr}) is calculated based on number of rotating register (N_r) and non-rotating registers (N_n), a virtual address in the source code ($N_n \leq R_{addr} < N_r$) and the stage count (S). The stage count is incremented each time an iteration of a loop completes ensuring the life-time of a variable remains intact. The range of the stage counter depends on the number of iterations in a loop. The physical, rotating address is calculated as:

$$R_{rot\_addr} = Nn + (R_{addr} + S)\ mod(Nr)$$

For example, if the generated code writes to register 13 in stage 2 and it wants to read that same data in stage 3 it will read from register 12 to access the same physical rotating register address. Therefore, it appears that the register is rotating downwards until the top of the non-rotating registers is reached.

Global Data Register File Forwarding Logic The data forwarding in ADRESv1 is relatively simple compared to the more complex one in Hennessy et al. [10]. Forwarding is only done implicitly in the global DRF or externally from the output of a functional unit to its inputs. The latter is visible for the DRESC compiler, which is treated as a resource for routing. The forwarding logic in the global DRF compares the write with the read addresses and forwards the data to the appropriate read port when there is a match as is depicted in Figure 2.10. The disadvantage is the amount of comparators created due to the multiple write and read ports. With a ratio of 2:1 for read vs. write ports and N write ports the number of comparators will be $\#Read \cdot \#Write\ ports = 2N \cdot N = 2N^2$ comparators. For example, with 8 write ports there would be 128 comparators. To avoid this large amount of comparators the number of write ports should be as low as possible. This could be done by sharing the ports among several FUs, but could reduce performance, since only one FU can access the write port in one clock cycle.

This simple bypassing logic avoids an additional cycle when a register being written and read in one clock cycle. An improvement on register forwarding is suggested by Sami et al. [37] and [38] preventing writing/reading of short-lived variables to/from the register files. This prevents unnecessary switching activity in the register files with an expected reduction of power by 20 - 35% with IDCT. The compiler reads in a trace file to determine which variables are short-lived for only a few clock cycles and generate write and read inhibit bits for the register file. These inhibit bits are used during runtime of the application and have to be stored somewhere during application execution. The inhibit

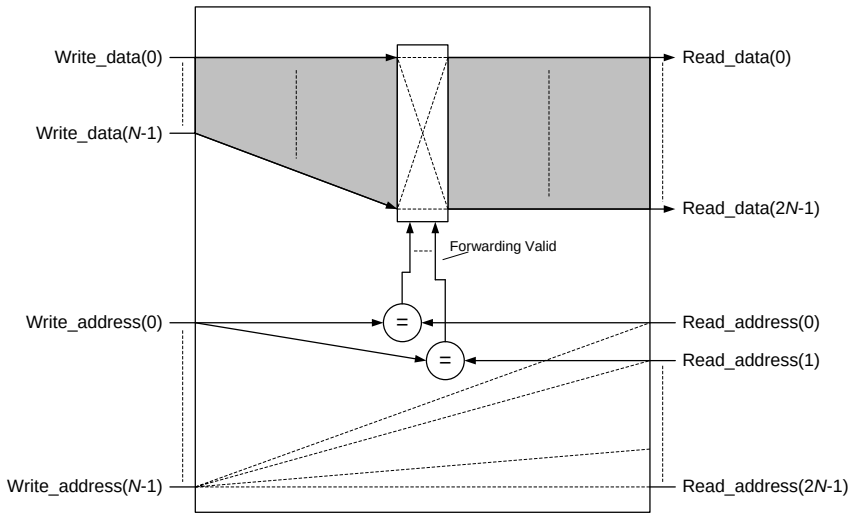


Figure 2.10: Forwarding logic in the Global Data Register File

bits can either be (1) placed in unused instruction encoding bits in the ISA or (2) create a specific location in the FU instruction field or (3) create dedicated hardware. The first option does not guarantee unused locations are available in every instruction, as is the case for most instructions of ADRES, while the second option does not require complex encoding logic. The third option must check for write-after-write [10] dependencies creating more logic. Therefore, the second option is a good optimization possibility to be implemented in the future.

2.2.5 Configuration Memories

ADRES relies heavily on the configuration memories in the array and for the data and the instruction memories. The configuration memories (CM) in the CGA section are distributed providing memories for every functional unit and register file. These memories hold instructions for the functional units, routing information for multiplexors and addresses for the register files. In CGA mode the memories are active every clock cycle and the address changes constantly increasing power. The number of lines of the configuration memories depends on the total amount of Initiation Intervals (II) of all the loops in the programs combined. However, if a program only runs in VLIW mode the configuration memories can be left empty and remain idle during the entire program.

If a different program has to be executed, of which its configurations are not available in the CMs, the configuration data are loaded by the Configuration Memory Interface (not depicted in Figure 2.1) when ADRES is idle or running in VLIW mode. It receives 32-bits words externally through the AHB Slave and forwards the data to a configuration memory when a configuration line is ready. The configuration line width depends on the size of the CGA configuration fields as depicted in Figures 2.6 and 2.8.

2.2.6 Modulo Scheduling

Modulo scheduling is a form of *software pipelining* [10] which is entirely done by the compiler. This section gives a small explanation how this works.

According to Hennessy et al. [10] software pipelining reorganizes loops such that several instructions of different iterations of the original loop are executed simultaneously. This provides high parallelism among instructions without loop unrolling, while preventing inter and intra dependencies in a single loop iteration. Modulo scheduling is basically the same as software pipelining, but now each new iteration is scheduled on a regular peace [36], [44] determined by the *Initiation Interval* (II). Just as with software pipelining this prevents resource usage conflicts and dependency violations. The number of stages in an iteration is called *stage count* a.k.a. *schedule length* in the ADRES architecture.

As with every software pipelining algorithm the pipeline/array has to be filled up. Filling up the array occurs in the first few iterations of the loop, which is called prologue. The prologue is followed by the steady state (or kernel) when the array is filled with a maximum number of operations. After the steady state the last instruction have to be finished until the array is empty. These last few instructions created the epilogue. For maximum usage of the array the epilogue and prologue should be reduced to the minimum increasing performance. An illustration with a 2x2 array is depicted in Figure 2.11 with courtesy to Binfeng Mei [27].

There are 3 stages and assuming FU3 is the only LD/ST unit and n1 and n2 are memory operations. With this setup there is a constraint on resources increasing the II, since n1 and n2 have to wait for each other. So, after 2 different configurations of the FU a new loop can start until all iterations are completed. This would be different if e.g. FU1 would also be a LD/ST unit creating more resources and making the II = 1. When all the iterations are finished the epilogue is still remaining. The length is stored by the *schedule length* in the ADRES architecture. When this value becomes zero it signals that the epilogue is finished and the VLIW CU in ADRES has to switch back from CGA to VLIW mode.

The constraint on resources based on the components in the XML architectural file gives a *Resource Minimal Initiation Interval* (ResMII), while direct or indirect dependencies with the same operation in a previous iteration gives a *Recurrence Minimal Initiation Interval* (RecMII). Combining these gives a *Minimal Initiation Interval* (MII) that the scheduler will start from to successfully place the operations on the array. If a scheduling is not with the given MII this is simply increased and the compiler start over. To address these Place&Route problems a *Modulo Routing Resource Graph* (MRRG) [27] is created combining features of software pipelining and routing resource methodologies utilized by FPGA P&R.

Modifications of the compiler is beyond the scope of this thesis, but it does support intrinsics only requiring adjustments in the ANSI-C source code. By compacting different instructions the scheduling should become easier and higher performance and less power consumption can be obtained. Intrinsics are not available in the benchmark applications neither were they implemented in the applications.

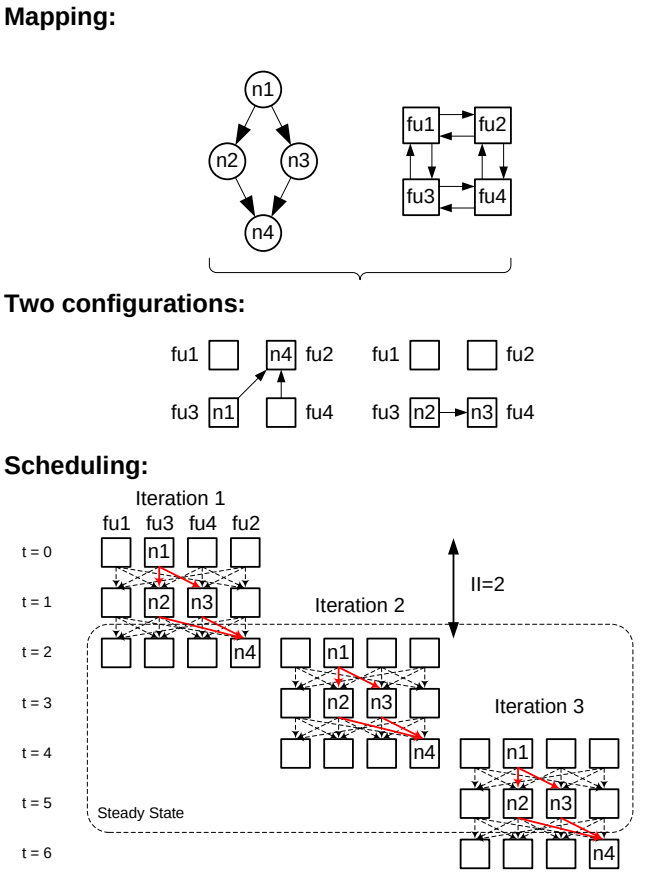


Figure 2.11: Modulo Scheduling example on a 2x2 array

2.3 Instruction Cache

The ADRES instruction cache provides the 32-bits instructions to the functional units in the VLIW section. The cache is direct mapped for its simplicity and performance, but at the expense of miss rates and inefficiency. As depicted in Figure 2.12 it has 4 memory blocks with 512 words by $N\times$ 32-bits for each memory where N is the amount of functional units in the VLIW section. In addition to this it has 2 TAG memories with 512 words by 17 bits due to the TAG address.

The first stage of the instruction cache is the same as depicted in Figure 2.7, which fetches the instructions. The proper line is selected by the 17 MSBs of the program counter while bits 3 and 2 select the correct bank holding the instructions. After comparing the tags the instructions are either send to the fetch2 stage or the ADRES core is stalled depending if the instruction is in cache or not. In the latter case it fetches one entire line from the external L2 memory after which the core and decompression resumes. Due to the instruction compression (explained later) it might be necessary to check two lines for presence of the instructions. If the program counter points to the end of the instruction line TAG0 checks the current line, while TAG1 checks the next line. If one of the TAGs miss a stall is created to fetch the instruction line.

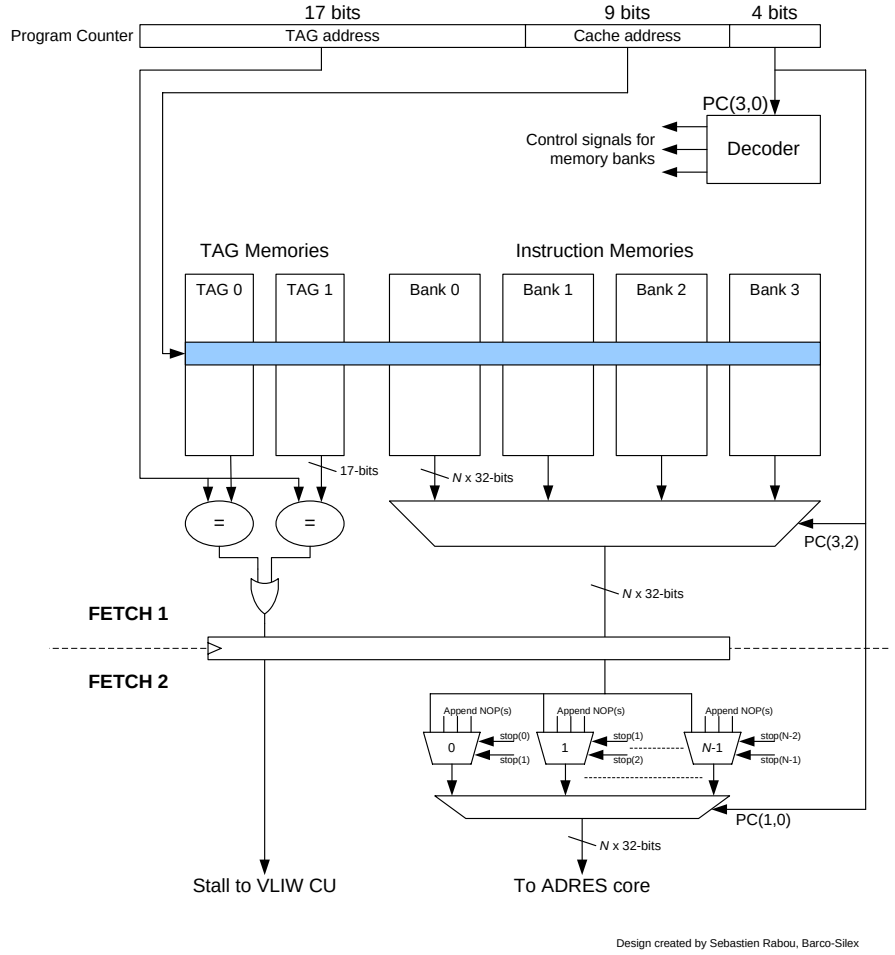


Figure 2.12: ADRESv1 Instruction Cache

Compression is performed by the compiler making sure no NOP instructions are between two valid instructions to avoid clutter. It then removes all NOP instruction and sets the stop bit (Figure 2.6) in the last instruction of the issue slot. Fetching is the reverse procedure with a few exceptions. Since there are no NOP instructions in a fetched instruction issue slot also instructions from the next slot can be included as well. By using the stop bit in combination with multiplexors NOP instructions are concatenated to the valid instruction(s) discarding the invalid ones. Depending on the amount of NOP instructions added the program counter is increased and fetches the next line. A regular increase can only be a maximum of N, but with only 2 selection bits (bits 1 and 0 of PC), this is currently 4 limiting the size of the available functional units in the VLIW section. Unfortunately, decompression requires a significant amount of multiplexors increasing area and static power, however, the estimated reduction in switching activity of signals between the ADRES core and instruction cache is between 30% - 40% [3], [6] and [23].

2.4 Data Memory Interface

Besides a L1 instruction cache ADRES also has a L1 data memory that functions as a scratchpad for applications and communication medium between the VLIW and CGA sections of ADRES and with the external environment. Due to the register-to-register architecture special Load and Store instructions are required to access the data memory that can only be executed by Load/Store (LS) units. As depicted in Figure 2.13 the data memory interface consists of an address decoder for every LS-unit, interconnection networks (X-bar), bypass logic, queue (FIFO), re-order buffers and pipeline registers. The memory banks are externally connected to the interface. For simplicity the connection to the external environment (DMEM AHB Master) is omitted, but operates in similar way.

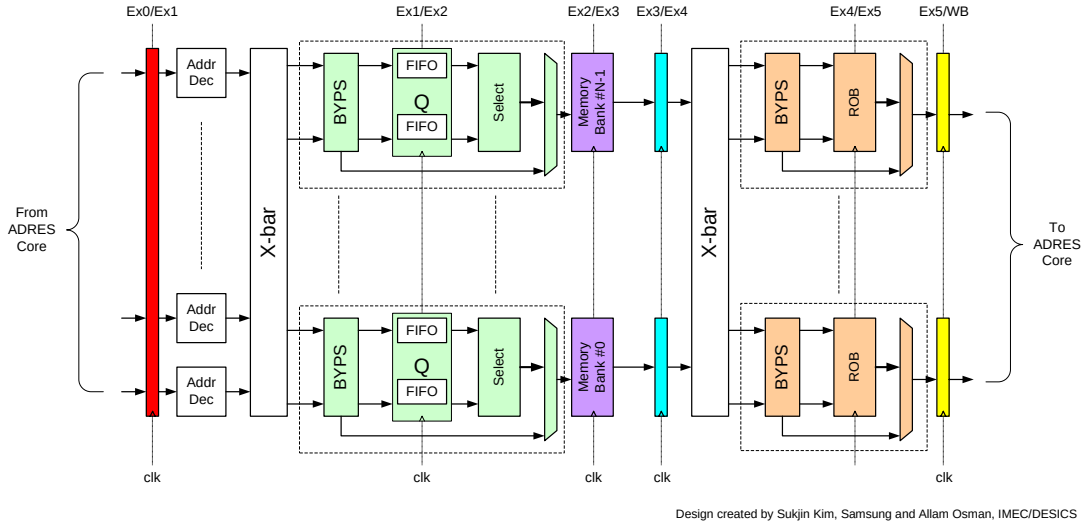


Figure 2.13: ADRESv1 Data Memory Interface

Any request to memory is rerouted through the input X-bar to the bypass queue logic first. This logic can bypass the queue when this is empty or when a low latency mode is used (explained later). The memories have single read and write ports requiring a FIFO to ensure that request to the same memory bank are placed in this FIFO and adds a time tag. When the queue is full the ADRES core or AHB slave port is stalled until some requests in the queue are handled. Based on the first-come-first-served principle the most time critical request is forwarded to memory. In case of a read operation the data is retrieved from memory and routed through the interconnection network to the bypass logic and re-order buffer (ROB). The ROB rearranges read data according to the time tag added in the memory queue and must have the same size as the maximum latency minus 1 to be able to handle all requests. With the bypass logic it is possible to bypass the ROB when the data reaches its deadline and should get priority.

The memory access can be done with low or high latency (latency is number of clock cycles), which bypasses the memory queue and ROB or not. For clarity, the execution stages in Table 2.2 are iterated for the data memory interface only in Table 2.3 with

high latency. With low latency execution stages 1 and 4 can be omitted, but this is only possible when no memory bank access conflicts can occur. If that happens the ADRES core is stalled until the conflict is resolved.

Table 2.3: Data Memory Interface execution stages

Ex. Stage	Instruction type	
	Store	Load
Ex1	Send address and data to queue	
Ex2	Send most timing critical request to memory	
Ex3	Write memory	Read memory
Ex4		Send data to the reorder buffer
Ex5		Send data back from DMEM to ADRES core

2.5 ADRES Base Architecture Selection

For this thesis power and performance optimizations have to be applied to an architecture of choice. The versatility of ADRES, however, does not make the selection unambiguous, since different interconnection schemes can influence the results significantly. Research has been conducted by Binfeng Mei et al. [31] to select the most appropriate architecture in terms of performance per unit area compared to a fully interconnected network. Although no immediate architecture could be selected as the most optimal, one could conclude that variation in interconnections could have a significant impact on performance and area. The architecture selection, therefore, is based on empirical results obtained by synthesis and simulation of 14 different 4x4 non-pipelined architectures ranging from a simple MESH to a completely interconnected network. These 14 different architectures are based on interconnection options as will be explained in the next section. The empirical results with a frequency 100MHz are depicted in Figures from 2.15 to 2.23. For verification we use the IDCT and FFT applications from Chapter 4 to obtain power and performance figures. A single functional unit with registers is also synthesized for simple comparison.

2.5.1 Exploration Options

The 14 different architectures are based on 7 different architectural featured options as noted in Table 2.4. The schematic representation of each option is depicted in Figure 2.14. The *Morphosys* option was not evaluated for this exploration, however, it is explored in Chapter 5 for optimizations.

For proper routing during Modulo Scheduling in Section 2.2.6 the Mesh architecture is the minimum requirement as interconnection. The *mesh_plus* and MorphoSys (not used for the exploration) interconnections are based on Mesh, which just have more interconnections. The *reg_con1* and *reg_con2* options are connected diagonally with their neighboring functional units and register files. The advantage with this interconnection is more routing, but also the possibility of sharing of data among FUs connecting directly to the register files. The *extra_con* option is created to bypass the functional unit to ease

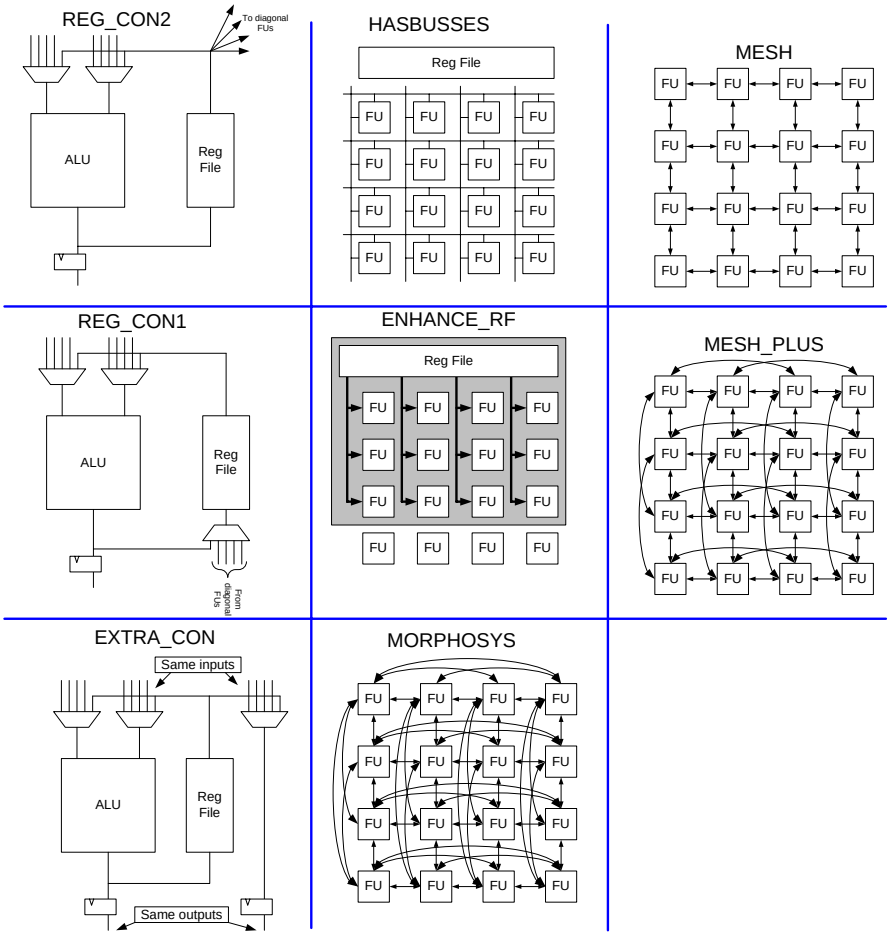


Figure 2.14: Interconnection Options for Architectural Experiments

routing for the DRESC compiler. The *enhance_rf* option has shared read and write data ports from the global DRF to all functional units in the same column except the last row as is also used by Kwok et al. [24]. Although this is beneficial for communication it also accesses the global DRF quite often increasing power consumption. Splitting up the power-hungry DRF into smaller, local DRFs was one of the mean features of power reduction as explained in the introduction of this chapter. For completeness, however, the option is accounted for. The option *has_busses* determines if predicate and data busses are implemented of 1 and 32 bits wide, respectively. Connecting to the busses is done via multiplexors and not via tri-state connections increasing area and power consumption. On the other side, if switching activity is avoided in other components by these 'high-speed' connections, eventually power reduces.

For selection of the basis architecture the physical properties of register files and functional units should be consistent. The local and global register files have 16 and 64 words, respectively. The data bus width between data register files, external memories and functional units is 32-bits. Each functional unit is capable of regular additions and multiplications, however, due to the Esterel simulator requirements, utilizing DRESC2.0,

Table 2.4: Architectural Exploration Selection Options

Architecture	mesh	mesh plus	reg con1	reg con2	extra con	enhance RF	has busses
mesh	X						
mesh_plus	X	X					
xtra_con	X	X		X			
reg_con1	X	X	X				
reg_con2	X	X		X			
reg_con_all	X	X	X	X			
enh_rf	X	X				X	
busses	X	X					X
arch_1	X	X		X	X	X	
arch_2	X	X	X		X	X	
arch_3	X	X		X		X	
arch_4	X	X	X			X	
all	X	X	X	X	X	X	X
ref	X	X	X		X	X	X

all inputs of the VLIW FUs have to be linked with the global DRF with a minimum of three source ports. To reduce waste of space of the global DRF only the FUs in the VLIW section have load/store capabilities, but these will also be utilized in CGA mode. This gives the global DRF 12 read and 4 write ports, the global PRF 4 read and write ports, the local DRFs 2 read and 1 write ports and for the local PRF 1 read and write port. The configuration memories have 128 words to provide enough space for the benchmark applications. Their bus sizes are variable and no memories are merged. We use the Artisan 90nm general purpose libraries for standard cells and memories, since these are most reliable as proved in Section 6.1. Although we base our findings on the non-pipelined version of ADRES also data and instruction memories should be included. However, we assume ideal external memories that only consume energy when we access them. Since the amount of accesses to data memory is the same for all architectures we can discard these results. For instruction memories this is a different case, since the amount of instructions and cycles varies. A simple checkup showed the energy consumptions of the instruction memories for different architectures do not differ that much, discarding these memories as well for the exploration.

2.5.2 Instances Selection

The results after synthesizing the 14 different architectures and obtaining power figures of the IDCT and FFT benchmarks are noted in this section.

Leakage power consumption and area are interlinked with each other as depicted in Figure 2.15 and 2.16. Most area is accounted for by the configuration memories, global DRF and the FUs in the CGA section and are similar for all architectures with a few exceptions. The Synopsys synthesis tool is not always consistent in its synthesis results creating a smaller global DRF for *4x4_ref* as the others while the same parameters are

utilized.

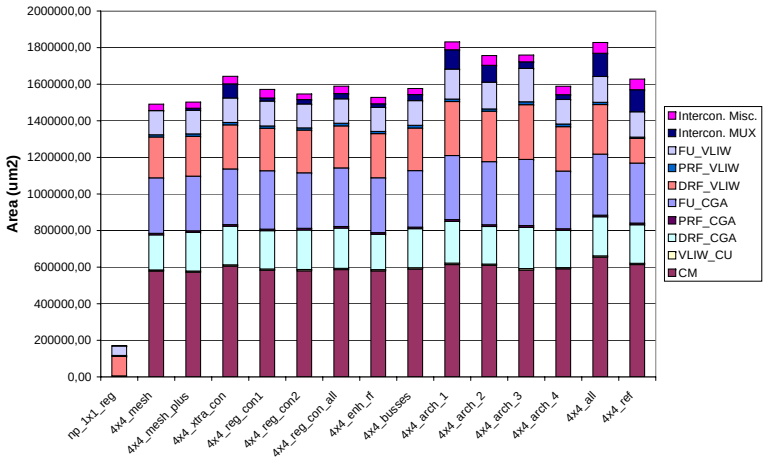


Figure 2.15: Area of First Architectural Exploration

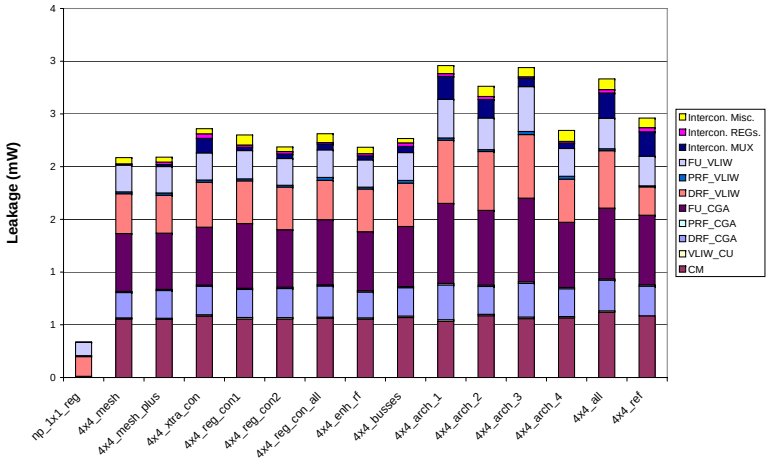


Figure 2.16: Leakage Power of First Architectural Exploration @ 100MHz

Executing the IDCT and FFT applications at a frequency of 100MHz provide performance, power and energy figures that all have to be looked at. When looking at power alone (Figures 2.17 and 2.18) the most simplest one, the 1x1, would be selected immediately, but the energy (power×time product) is of more significance as depicted in Figure 2.19, since this determines how often you can run the application on one battery life. Additionally, the ratios of the results are also important as depicted in Figures 2.20, 2.21, 2.22 and 2.23.

Figure 2.15 for area and Figure 2.21 for FFT performance show that the fully interconnected architecture *4x4_all* is not an efficient way for operation. The *4x4_mesh*, on the other hand, requires less area, but has the highest energy consumption for both applications. The *np_1x1_reg* architecture has a relatively high MIPS/mW but has a

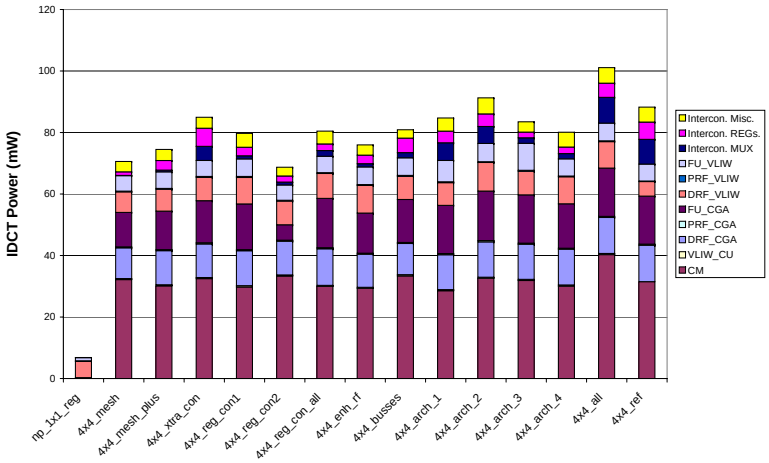


Figure 2.17: IDCT Power consumption of First Architectural Exploration @ 100MHz

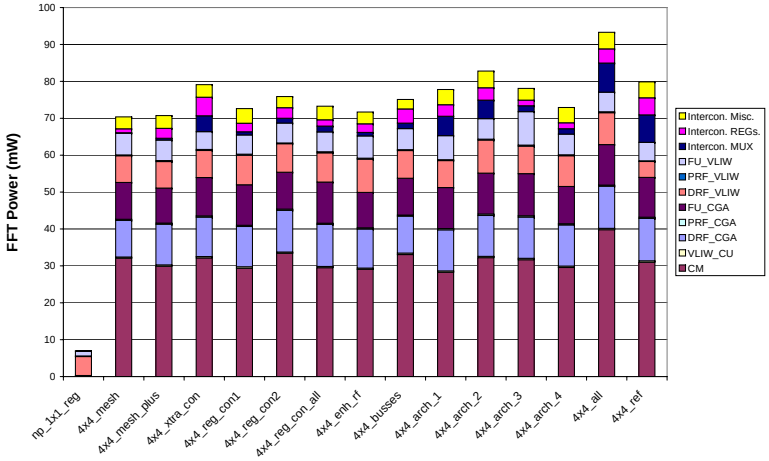


Figure 2.18: FFT Power consumption of First Architectural Exploration @ 100MHz

drastic reduction in millions of instructions per second (MIPS) compared to the others. The results also point out the bypass feature in *4x4_xtra_con* only has some advantage for IDCT, but certainly not for FFT. This is typically an application that access data registers files quite often, which is shown in Figure 2.21 by *4x4_enh_rf* and *4x4_reg_con_all* having a higher MIPS/mW factor compared to *4x4_xtra_con* due to the better accesses to register files. Busses in the design does also not give much advantages, since it requires more multiplexors, larger configuration memories and the data networks increase power.

Of all these architectures there is one that has the least amount of energy consumption, an average area of 1.6 mm^2 and one of the lowest values for area vs. performance ratio: *4x4_reg_con_all*. Figures 2.24 and 2.25 depict the energy-delay charts where 4x4_reg_con_all has the least amount energy with relatively low delay. Although performance for FFT is not one of the highest it is for IDCT. Additionally, MPEG2 utilizes IDCT giving it more 'weight' than FFT. This *4x4_reg_con_all* interconnection architec-

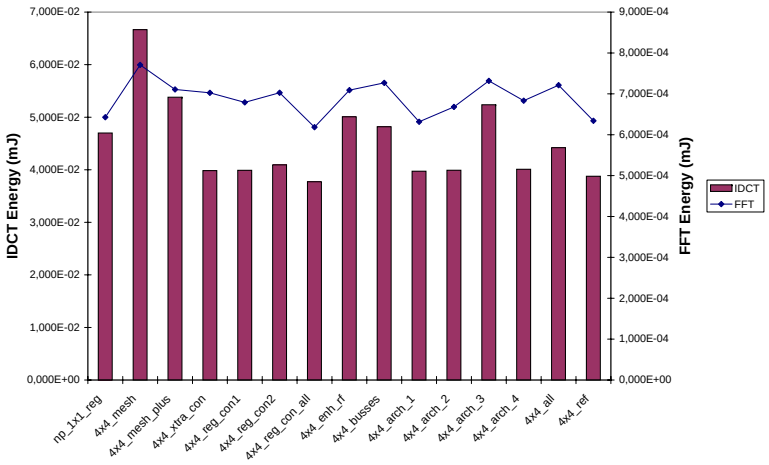


Figure 2.19: Energy consumption of First Architectural Exploration @ 100MHz

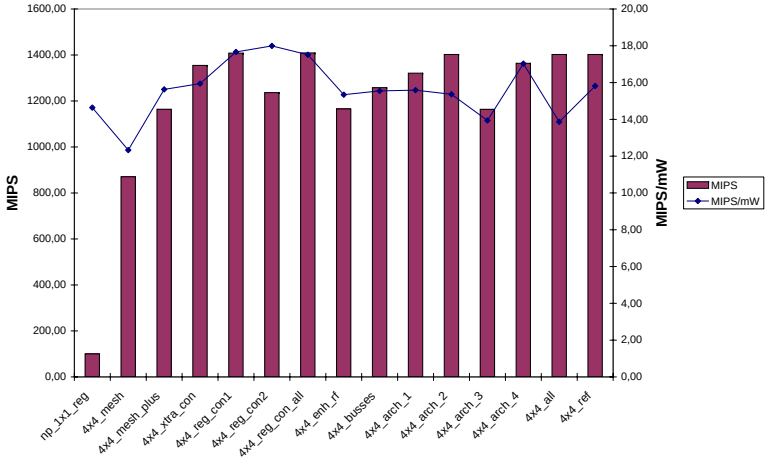


Figure 2.20: IDCT Performance of First Architectural Exploration @ 100MHz

ture as depicted in Figure 2.26 has connections between the FUs and register files directed through multiplexors. This architecture will function as a basis for optimizations in this thesis.

2.6 Summary

The ADRES architecture is a highly flexible coarse-grained reconfigurable array capable of applications acceleration with instruction level parallelism on a VLIW like section and loop-level parallelism on a coarse-grained reconfigurable array of ADRES. The combination of the pipelined ADRES core with a data memory interface used as a scratchpad and instruction cache for the VLIW section provides a high-performance, low energy design that can be tuned for specific designer requirements. The DRESC compiler compiles an application and maps it on the array. To obtain high performance the ADRES architec-

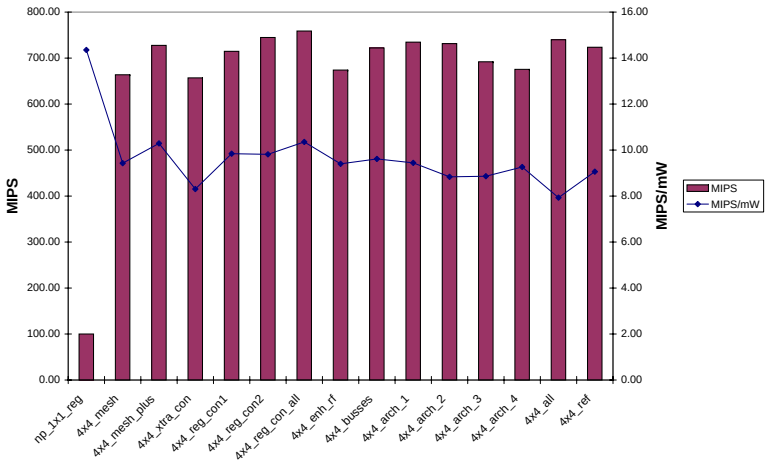


Figure 2.21: FFT Performance of First Architectural Exploration @ 100MHz

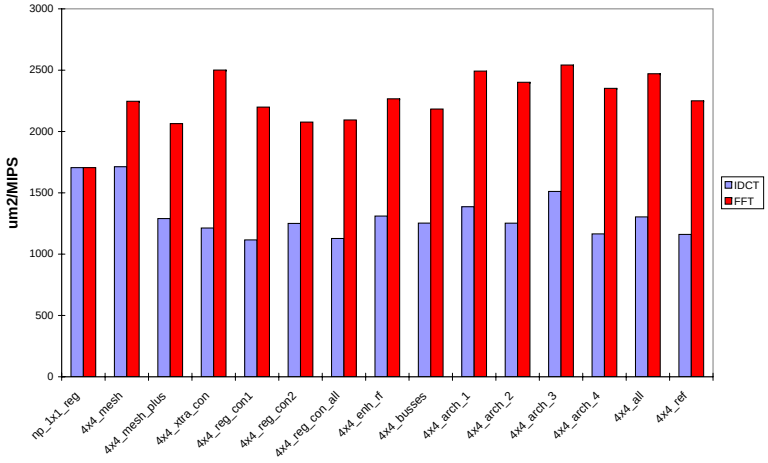


Figure 2.22: Area vs. Performance of First Architectural Exploration @ 100MHz

ture speeds up the parallel fraction of the application and results in overall speedup of the application.

The high flexibility of ADRES makes the selection for an architecture as a reference cumbersome. Out of 14 different 4x4 non-pipelined architectures the most optimal interconnection scheme is selected based on area, power, energy and performance results. The selected architecture consists of a regular *mesh* and *mesh plus* interconnection topology and diagonal connections through multiplexors between functional units and (local and global) register files. Power and performance optimizations are to be applied to the selected base architecture.

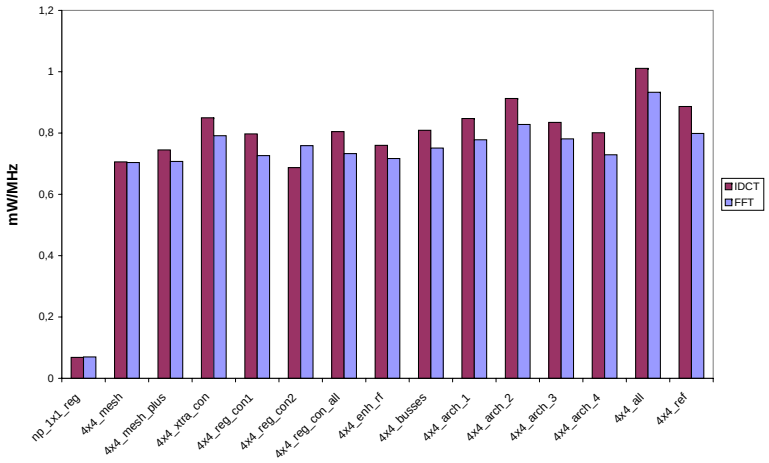


Figure 2.23: Power vs. Frequency of First Architectural Exploration @ 100MHz

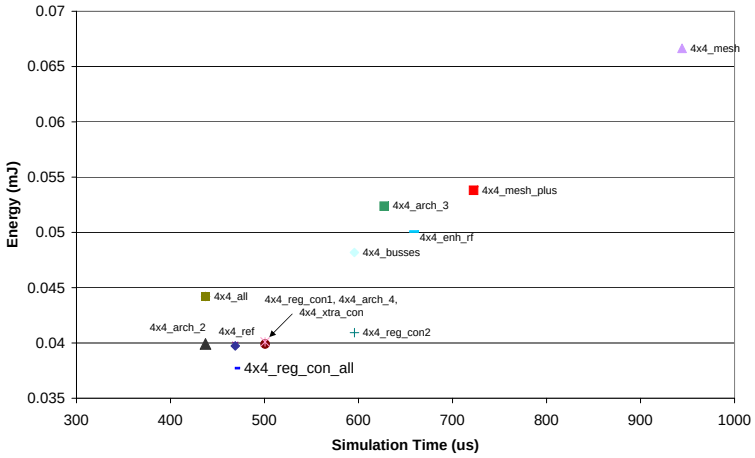


Figure 2.24: IDCT Energy-Delay of First Architectural Exploration @ 100MHz

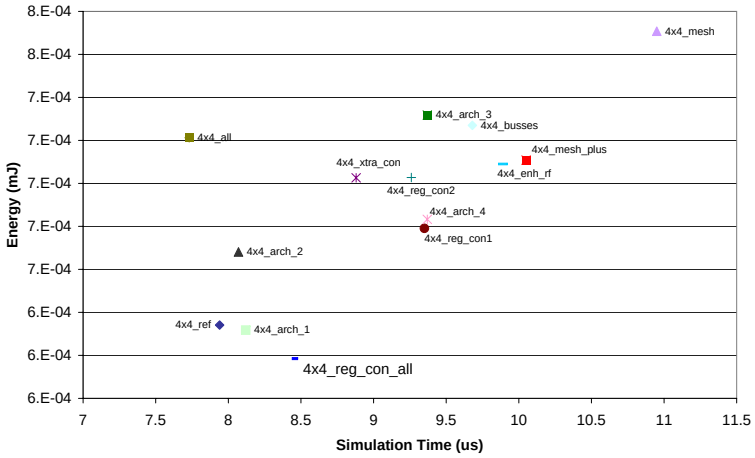


Figure 2.25: FFT Energy-Delay of First Architectural Exploration @ 100MHz

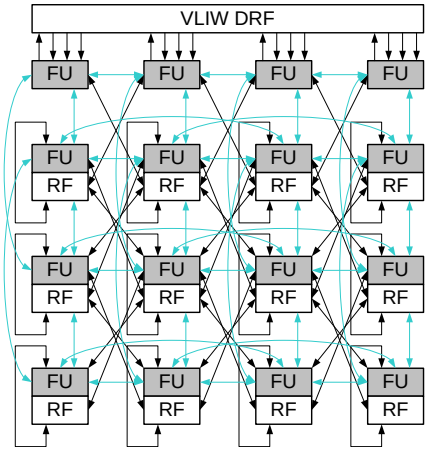


Figure 2.26: ADRES Base Architecture

3 Exploration Methodology

This thesis focuses on power and performance optimizations for ADRES as explained in Chapter 2, however, exploration of the architecture was only one of the first steps for the project. Before the optimizations can be applied other issues e.g. analysis, simulation, synthesis and power estimation flows have to be set up. We use DRESC, Esterel, ModelSim and Synopsys tools for these flows. Working with these tools proved to be quite challenging, since ADRES and DRESC were constantly in development that influenced the compilation and simulation results. Also the actual hardware description code of the processor and the tool's capabilities had to be explored, which were both significant issues during the coarse of the thesis.

This chapter describes all the steps made during the thesis and which decisions are made. First, the tool flow is set up globally for analysis and creation of ADRES instances. The next section depicts the possibilities of optimizations for ADRES and if they can be implemented. Finally, all previous steps come together in the latest version of ADRESv1 with instruction cache and data memory interface with the benchmark applications FFT, IDCT and MPEG2 simulated on them.

3.1 Tool Flow

To characterize the benchmark applications on the selected ADRES instances the tool flow, as depicted in Figure 3.1, is needed. It can be split up in basically three parts: 1) Compilation and Assembly resulting in performance figures, 2) Simulation obtaining power figures and 3) Synthesis resulting in physical characteristics e.g. area.

The application in the *Compile and Assemble* part is in ANSI-C code optimized for ADRES, e.g. transformation of loops for efficient CGA usage. If this is not the case the application has to be rewritten of which more details are described by Bennet et al. [27]. The optimized code is forwarded to IMPACT developed by the University of Illinios [17] targeting instruction level parallelism (ILP) processors such as VLIW and superscalar processors. In the compilation flow IMPACT transforms the source code to intermediate code and performs additional optimizations for ILP. The DRESC compiler in Figure 3.1 then schedules for ILP, allocates registers, analyses and performs modulo scheduling as described earlier. The resulting optimized DRE files are ready to be assembled and transformed to binary files. These files are also used to create a high level simulator to obtain basic performance parameters such as instructions per cycle (IPC).

The ADRES instance is synthesized in the *Synthesize* part to obtain the front-end of a chip as a gate-level design. First, a XML2VHDL parser is utilized to create the top-level VHDL architecture file. The VHDL descriptions of all individual components in this file such as the functional units, register files and control units are already available and are linked during compilation. The VHDL files are synthesized to a gate-level design

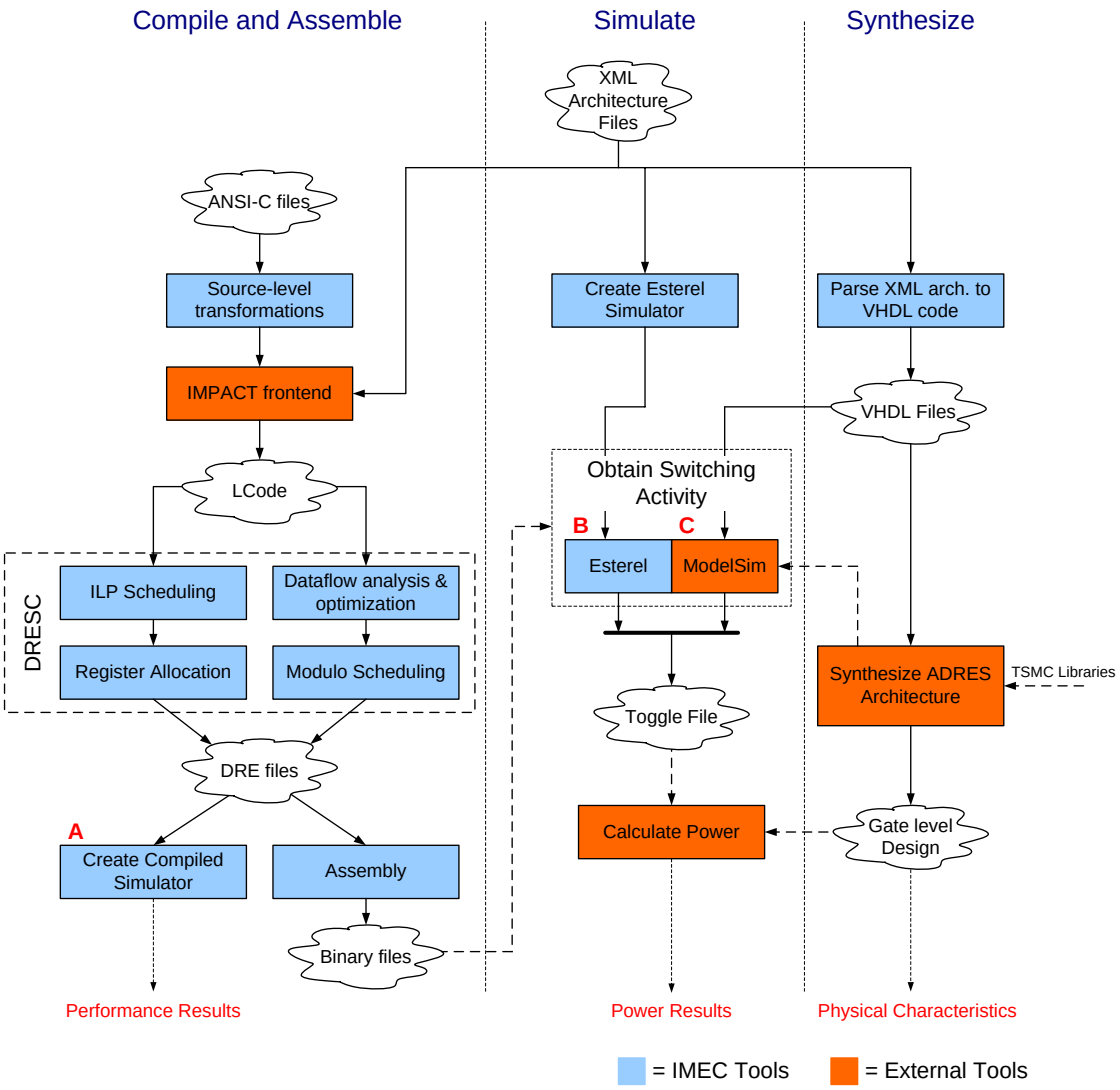


Figure 3.1: Global overview of Tool Flow

based on state-of-the-art 90nm TSMC [20] libraries. From the gate level design, physical characteristics e.g. area, capacitance and resistor values are extracted.

The *Simulate* section brings the synthesize and compile parts together. In the ADRES environment there are three simulators with different levels of accuracy and speed. The compiled simulator (marked as A in Figure 3.1) is a high-level simulator that validates the code with symbolic accuracy based on the XML architecture files. The performance results are accurate and it is also fast making it ideal for the large amount of architecture explorations in this thesis. One level lower is the Esterel simulator (marked as B) created by the Applied Mathematics Center, Ecole des Mines de Paris, and INRIA in Sophia-Antipolis [12] of which a language primer is available [2]. It is a synchronous language dedicated to control-dominated reactive systems and describes

the entire ADRES instance as a state machine. It can verify the ADRES instance at an intermediate level as an instruction set simulator and uses the XML architecture file to generate verification files for VHDL simulations.

Clock cycle accurate simulations are done by the ModelSim simulator, which verifies the benchmarks on the ADRES architecture instance on RT level of VHDL code and obtains switching activity of the nets. For maximum accuracy, gate level simulations were the target at the beginning of this thesis, which did not work. When modifying the top-level VHDL architecture file of ADRESv0 the RTL simulations failed as well. This made the regular flow of RTL simulations unsuitable for architecture explorations. To resolve this issue the Esterel simulator was adjusted by Andreas Kanstein to do the same as ModelSim during RTL simulation, which was obtaining switching activity of the nets. This was possible since the VHDL architecture file is generated out of the XML architecture file, which have strong relations with each other. In addition the Esterel simulator provided fast results compared to the ModelSim simulator.

The switching activities obtained after simulations are annotated on the gate level design created in the synthesize part. The toggling file and the gate level design are used by the power tool *PrimePower* to estimate power. The Esterel methodology has an offset of 11% in CGA mode compared to the ModelSim simulator as will be proved in Section 4.2.1.

3.2 Optimizations

Power and performance optimizations have numerous possibilities in a hardware design, however, not all of them could be applied during the coarse of this thesis. After reviewing all the original VHDL code of ADRES it became clear the code for the FUs was certainly not optimized to be properly be synthesized as an ASIC. Adjusting the entire FU to be proper for pipelining and ASIC synthesis was carried out by IMEC employees.

Dynamic optimizations like *operand isolation* focusses on the data path of the functional unit and has been hard coded in the design. Another dynamic optimization is *clock gating* to avoid unnecessary switching activity in nets and logic gates caused by the clock input. This technique targets the registers in the ADRES instance.

Static optimizations are implemented during design phase of an architecture. Techniques like *splitting up* the configuration memories, *pipelining* and top-level *architectural modifications* are selected. Pipelining is implemented by the IMEC employees, but an explanation is provided in Section 5.3.1. Architectural modifications consists of an exploration similar as in Section 2.5, but now focussing on sharing register files and reduction of the register sizes.

Besides design modifications the library used during synthesis has a significant impact on performance and power. In Section 6.1 the appropriate library (Artisan or Synopsys) based on 90nm TSMC technology is selected for both power and performance. During the thesis only the Artisan standard cell libraries are utilized, since they are more reliable and slightly better in power as explained in Section 6.1. ADRESv1, however, requires faster logic like the Synopsys standard cell libraries to obtain maximum performance results.

3.3 Final Steps

All the optimizations techniques for power and performance are implemented in the final design based on the latest version of ADRES available at that moment. The benchmark applications FFT, IDCT and MPEG2 are used to obtain final power results of ADRES and are compared to the base architecture. Additionally, an energy-delay chart is created with variable ADRES instance dimensions: 2x2, 4x4 and 8x8. In order to show the contribution of CGA the chart is compared to 2x1, 4x1 and 8x1 VLIW instances of ADRES.

There are other optimizations possible for ADRES noted in the conclusions and future work of this thesis, which were not implemented during this thesis period. These optimizations are expected to reduce power consumption of ADRES even further, which have to be investigated in the future.

3.4 Summary

The tool flow utilized in this thesis consists of compiling benchmark applications, synthesis of an ADRES instance and simulation of the application on the ADRES architecture. DRESC compiles the benchmark applications FFT, IDCT and MPEG2 onto the CGA and are verified by the simulation flow. A simulation can be done by either the compiled simulator, the instruction set simulator Esterel or the clock cycle-true simulator ModelSim. The ModelSim simulator was the regular flow of VHDL simulation and obtaining switching activity of the ADRES architecture, but modifying the ADRESv0 architecture made the RTL simulations fail. This regular method made architecture explorations impossible. The Esterel simulator is adjusted to obtain switching activities in the ADRES architecture similar to the methodology utilized by ModelSim. This technique is faster than ModelSim and was ideal for architecture explorations. The captured switching activities are applied to the synthesized gate level design of ADRES after which power estimations are obtained.

The optimizations are either dynamically implemented after the design has been finished or statically during the design phase. The dynamic optimizations are operand isolation and clock gating implemented to reduce switching activity in the architecture. The static optimizations are pipelining, architectural modifications and segmentation of the configuration memories. Pipelining is implemented by IMEC employees and improves performance, but also increases power consumption. The architectural modifications changes the base ADRES architecture and investigates the impact of register file sharing on power and performance. Segmenting the configuration memories is a methodology to reduce power in the memories.

The optimization techniques are merged with the selected ADRES instance after the architectural modifications. The benchmark applications are simulated on the architecture for verification and power estimations. In addition, a comparison is made with a VLIW like ADRES instance to prove CGA contribution.

4 Synthesis and Simulation Flow

The target architectures as selected in Section 2.5 have to be synthesized and simulated to obtain power and performance figures. The flow in Figure 3.1 is depicted in Figure 4.1 in a simpler form for convenience. As noted in Section 3.1, the C-source code is first optimized for efficient CGA usage. The optimized code is transformed by IMPACT to the intermediate LCode. Finally, it is compiled with DRESC 2.x and assembled utilizing the XML architecture file resulting in binary files for simulation. The compiled simulator provides the performance results for the code and conjunctive architecture. The simulation flow validates the compiled code on an intermediate and/or hardware RT level obtaining switching activity of the nets. The switching activity is annotated on the synthesized design after power estimations are obtained on RT simulation level.

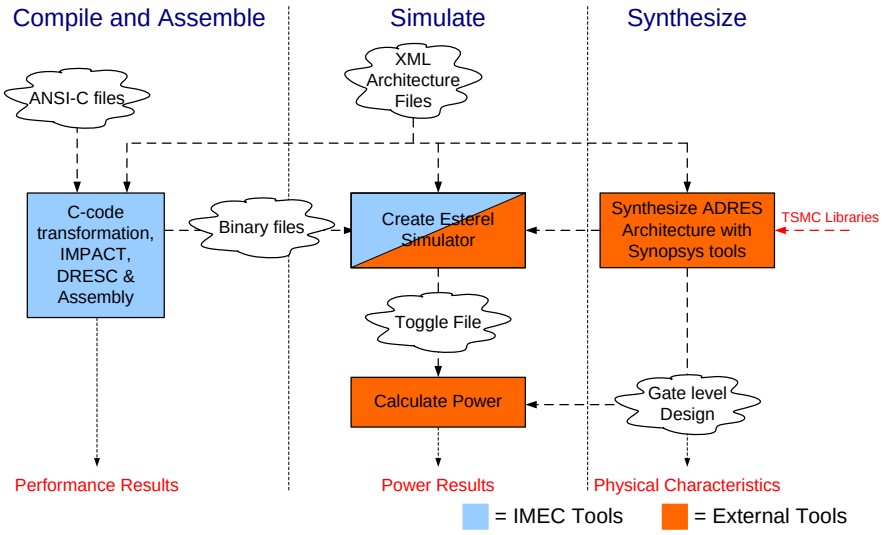


Figure 4.1: Simple representation of Tool Flow in Figure 3.1

This chapter first describes in Section 4.1 the steps made to synthesize the VHDL architecture file with *Synopsys'* [19] synthesis tools: *Design Compiler* and *Physical Compiler* into a logically synthesized (front-end) design. The simulation flow and power calculations are described in Section 4.2. The tools *Esterel* and *ModelSim* simulate the benchmark applications and create switching information of the nets as noted before. The accuracy between the Esterel and ModelSim is also investigated utilizing ModelSim as a reference. *PrimePower* of the Synopsys tool flow is utilized to calculate the power of the design on RT level. The power calculations and physical characteristics of the designs are based on the utilized Artisan and/or Synopsys libraries based on 90nm TSMC technology. Several simulations are performed on register files and (non-)pipelined functional

units selecting the most appropriate library as noted in Section 6.1. Global description of the compiler was already available in Section 3.1.

4.1 Synthesis Flow

The purpose of the synthesis flow is to transform the XML architecture file into a front-end, gate level netlist mapped to a specific technology. The flow starts by transforming the XML file into a top-level VHDL architecture file using the XML2VHDL parser as depicted in Figure 4.2.

The next steps are done by the Synopsys tools, which analyzes the VHDL code and translates it to components extracted from the generic technology (GTECH) and DesignWare library [41] during elaboration. Both libraries are technology independent of which GTECH contains basic logic gates and flipflops, while the DesignWare library contains more complex components like adders and comparators. Clock gating is implemented during synthesis and has to be set during the elaboration step. Design Compiler utilizes the features of *Power Compiler* to analyze the HDL code for the specific structure to implement clock gating as depicted in Listing 4.1. It uses the write enable signal (*we*) and address (*Address*) from the code to create the control logic as depicted in Figure 5.5(b). The clock signal is only forwarded when the WE and Address values direct the register file. The AND-based clock gating elements in that figure are implemented automatically by Power Compiler together with a latch to avoid glitches.

Listing 4.1: Example of register file VHDL code

```
WriteRegisterFile: process (clk)
begin
  if(clk'event and clk = '1') then
    if( we = '1') then
      register_file(Address) <= write_value;
    end if;
  end if;
end process WriteRegisterFile;
```

After elaboration a RTL2SAIF file is created containing all the technology independent components. It instructs the ModelSim simulator which modules' in- and outputs have to be monitored during simulation. The same method is also adapted in the Esterel simulator as will be described in Section 4.2.

During logical compilation the design is optimized and mapped to a technology specific target library that will be selected in Section 6.1. When there is no floorplan available during compilation Design Compiler creates its own, which is not optimal except for the critical path. In the physical compilation step, the synthesis tool *Physical Compiler* is capable of basic placement and routing, which is sufficient for the front-end designs of all the tested architectures in this thesis. The Physical Compiler additionally optimizes the design even further based on the wire load model for the capacitance values, layout and timing constraints. The resulting gate-level netlist contains all the resistors, capacitances and interconnection delay values. These can be back-annotated in

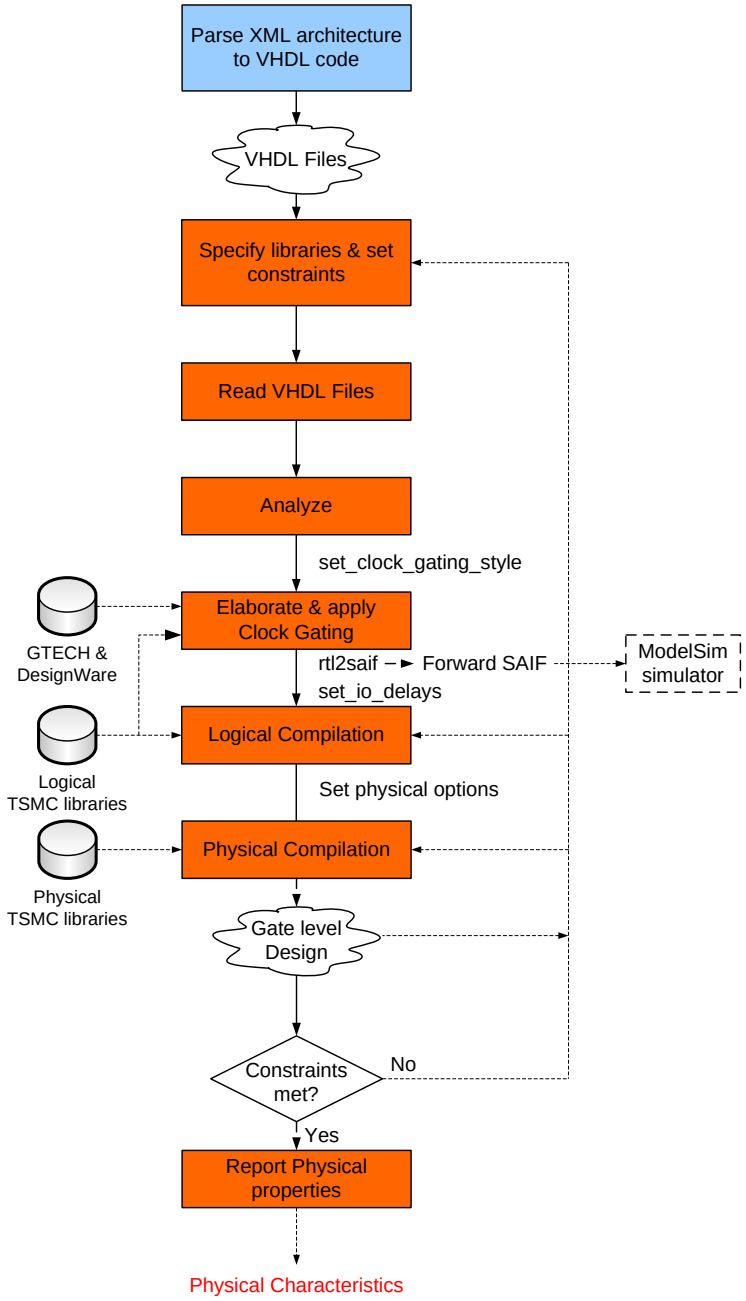


Figure 4.2: Synthesis flow

the flow if the timing constraints are not met. The timing constraints are dependent on the ADRES architecture instance, which target either 100MHz for ADRESv0 or 500MHz for ADRESv1. The critical path determines the maximum frequency of the synthesized architecture. If the design met the timing constraints or it can not be optimized any further the gate level design is utilized for power estimation as described in the next section. Physical (area, minimum clock period, etc.) attributes noted in this thesis are

as well based on the gate level design.

4.2 Simulation Flow

The simulation flow has the purpose of validating the ANSI-C source code and obtaining switching activity of the top-level nets in the VHDL architecture file to perform power estimations. There are three different simulators available to validate the applications and the architecture: compiled-code, Esterel and ModelSimv6.0a simulators as depicted in Figure 3.1. The compiled-code simulator is in the *Compile and Assemble* part, while Esterel and ModelSim are in the *Simulation* part. The simulators range from a high-speed, symbolic accurate to the low-speed, clock cycles and bit-level accurate simulators. As noted in Section 3.1 the compiled-code simulator is a high level simulator with excellent simulation speed. Esterel is a synchronous, cycle accurate simulator [12], which can create test vectors for the VHDL simulations. ModelSim is an event-based simulator as is a common approach for VHDL simulators [18]. It is the lowest level for simulation with bit-level accuracy, unlike Esterel, and can also have Z and X states in the simulation waveforms.

The Esterel and ModelSim simulators are also used to obtain switching activity from the nets to calculate power. The HDL simulations did not work properly anymore with the non-pipelined version ADRESv0 when the architecture was changed. This was actually an interlinked problem between the compiler, assembler and the ModelSim simulator as well. The tool versions (DRESC1.x) for ADRESv0 and its HDL code were not stable enough. As noted in Section 3.1 the Esterel simulator for DRESC2.x was capable to resolve this issue and could be used for the architectural explorations as described in Sections 2.5 and 5.3.2. For the pipelined version ADRESv1 the ModelSim simulator is used, since it does not have the problems as in the previous version.

As depicted in Figure 4.3 Esterel parses the modules described in the XML architecture file into the Esterel language [2], which is similar to the XML2VHDL parser. The generated code consists of concurrent state machines communicating via synchronous signals, with an interface to ANSI-C for data manipulation. The Esterel and ANSI-C code are compiled together into one single, flattened state machine completely described in C. After the compilation the Esterel simulator is obtained and is able to monitor the inputs and outputs of the Esterel components. The switching activity obtained from the nets is placed in a toggle data file, which is later transformed to PrimePower format.

The flow for the ModelSim simulator is shorter than that of Esterel and operates on VHDL files instead of Esterel code. Compiling the VHDL files places the designs in a library where the simulator will read from. After simulation and capturing the switching activities from the nets a backward SAIF file is created to be read later by PrimePower for power calculations.

4.2.1 Capturing Activity

Besides validation of the VHDL code another important feature of the Esterel and ModelSim simulators is to obtain switching activity of the nets in the architecture.

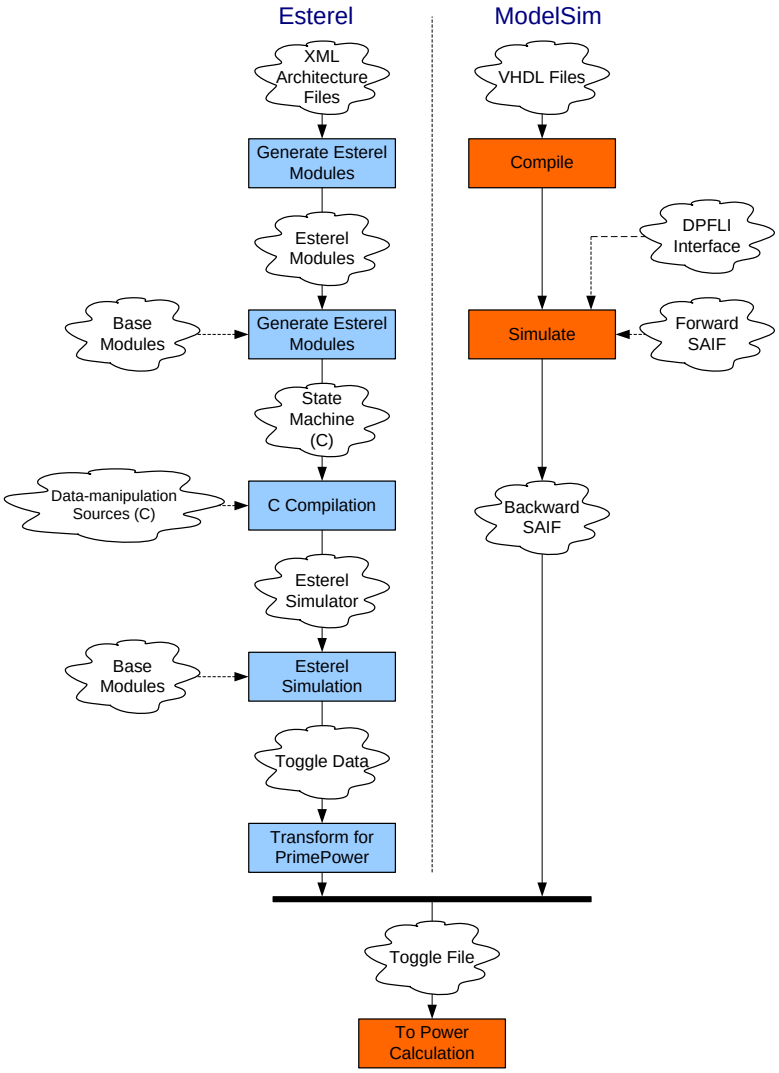


Figure 4.3: Simulation flow

ModelSim utilizes the forward SAIF created in the synthesis flow to capture the activities of the primary in- and outputs and other synthesis-invariant elements. For ModelSim to be able to read the forward SAIF file a link between Synopsys files and ModelSim is created by a DesignPower interface called DPFLI [42]. After simulation the backward annotation file as depicted in Figure 4.3 is forwarded to PrimePower for estimations. Figure 4.4 depicts an example how the activity of the nets are captured by ModelSim. The simulator monitors the inputs and outputs of the FU, DRF and TRN components as marked by the dashed lines. After simulation the obtained switching activities are saved in the Toggle File and used for power estimations by PrimePower.

The activity values after ModelSim simulation consist of the time periods in logical '0', '1' and 'X' (don't care) state. The amount of glitches and toggling from $1 \rightarrow 0$ and vice versa are also present in the backward SAIF file.

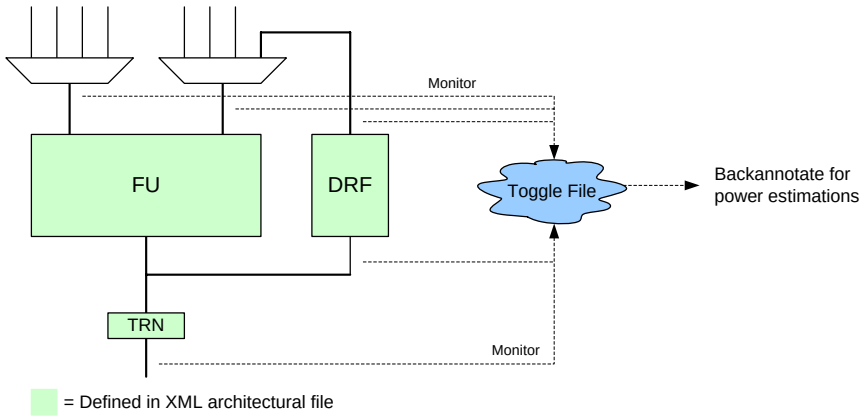


Figure 4.4: Capturing Activity

Esterel captures the switching activities of the nets during simulation the same way as depicted in Figure 4.4. After simulation the same kind of information should be created as in the backward SAIF file for PrimePower to be able to work with. However, the values during Esterel simulation can only be logically ones and zeros, while don't care states and glitches are not accounted for. Fortunately, simulations with the reference version of ADRESv0 showed these values are minimally present and can be ignored at this level.

Figure 4.5 depicts how Esterel saves the captured switching activity in the toggle data file. The values *time_0* and *time_1* contain the number of cycles the signal was set to logically zero and one, respectively. The values *T01* and *T10* contain the number of $0 \rightarrow 1$ and vice versa transitions, which are usually equivalent with a maximum difference of 1. Based on these values the *toggle rate*, *simulation period* and *static probability* can be determined for PrimePower. The static probability calculates the period the net was set to logically one. It is required for statistical propagation when a interconnected signal is not annotated.

The transformation to the *set_switching_activity* PrimePower command is required to annotate the values properly on the gate level design in PrimePower as noted as well in Figure 4.5.

Esterel: Toggle Data: (0, 1, 0->1, 1->0)
<signal_name> time_0 time_1 T01 T10
....

PrimePower: set_switching_activity -toggle_rate (T01+T10) -period (time_0+time_1)*t_{clk_per}
-static_probability (time_1/(time_0+time_1)) <signal_name>
....

Figure 4.5: Transforming Esterel Toggle Data for PrimePower compatibility

With the Esterel methodology about 40 - 50% of the signals at RT-level are annotated while the rest are outputs of individual registers. The outputs of the register files (where the registers reside) are annotated by statistical propagation obtaining the full 100% of annotation at RT-level. The signals that can not be annotated or propagated obtain a

default value, which is higher than it should be after reviewing the PrimePower reports.

The differences between the Esterel and ModelSim simulation for power estimations are noted in Table 4.1. The simulations are based on the ADRESv0 reference version with a frequency of 100MHz.

Table 4.1: Differences between Esterel and ModelSim for ADRESv0

	IDCT	
Simulator	VLIW only (mW)	CGA (mW)
ModelSim	46.5	59.16
Esterel	57.75	65.65
Difference	24.2%	10.9%

The empirical results based on an IDCT simulation on ADRESv0 show an overestimated offset of 11 - 24% when ran in CGA mode or VLIW mode only, respectively. To keep maximum accuracy for power estimations with Esterel it is best to compile and simulate the benchmarks on the CGA and not only on the VLIW section.

4.2.2 Power Estimation

Power calculation is performed by PrimePower of which the flow is depicted in Figure 4.6. The gate level design is read together with the physical TSMC libraries including the memories. For power calculation all libraries are set to *typical* (25°C, 1.0V supply voltage) and linked together.

The toggle data file is read and annotated on the synthesized ADRES architecture. PrimePower calculates the power estimated consumptions of all the components and provides detailed reports of the static and dynamic power components. Static (or leakage) power is obtained from tables in the TSMC libraries where the value depends of the input signals of a logic component. Dynamic power is calculated by the summation of switching and internal power. The internal power is based on the toggle rate annotated on the nets and calculated in a similar way as switching power. For this reason, only the formula to calculate switching power is noted below. The total switching power is calculated using the following formula.

$$P_{total\_switching} = \frac{V^2}{2} \cdot f \cdot \sum_{\forall nets} \alpha_i \cdot C_{load_i}$$

In the formula, α_i describes the average switching activity per second for net i. It is calculated by the summation of *T01* and *T10* (toggle rate) divided by the simulation period. The *T01* and *T10* values are usually the same, hence the division by 2 in the formula.

The capacitive value C_{load} can either be obtained from a Place&Route tool like *Encounter* [15] or from a wire load model described in the libraries. Encounter obtains the capacitance values after place&route, while the methodology with wire load models

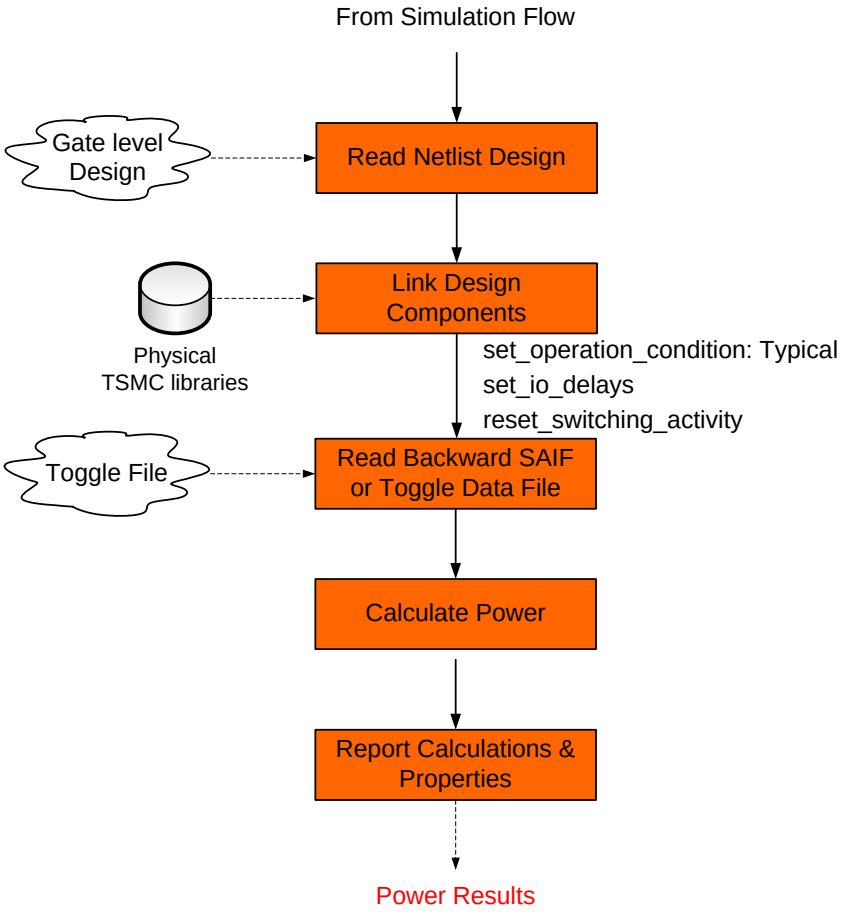


Figure 4.6: Calculate Power

obtains the values from charts. This makes the wire load models less accurate as Encounter, but it gives a simple and good overview for power. In the final version of the created ADRESv1 architecture the capacitance values are obtained from Encounter as will be presented in Chapter 6.

The supply voltage value V is set to 1V when the typical settings are utilized during power calculations as is the case in this thesis. The frequency f is determined after synthesizing the ADRES design.

4.3 Summary

For this thesis an analysis, simulation and synthesis flow was setup to validate the power and performance optimizations applied to ADRES.

The synthesis flow synthesizes VHDL code into a front-end gate level design and is utilized for power estimations. A place&route tool is also used for detailed routing (back-end), but is only applied to the final ADRES architecture. Since routing of the nets is done more accurately then with regular physical compilation the the capacitance

and resister values are more reliable. This will create more accurate results during power estimation.

The compiled-simulator of DRESC provides the performance results and simple verification. More detailed verification of the benchmarks on the ADRES architectures is either done with the Esterel simulator or ModelSimv6.0a simulator.

Switching activity of the architecture nets is obtained at RT level to estimate power by PrimePower. The Esterel simulator is utilized for the non-pipelined version ADRESv0 during the architectural explorations, while ModelSim is used for the pipelined version ADRESv1 for obtaining switching activity files. The Esterel simulator has an overestimated offset of 11% with the IDCT benchmark in CGA mode compared to ModelSim simulation making it a good alternative for power estimation.

PrimePower reads in the switching activity file obtained from the benchmark simulation and the gate level design after synthesis. Power is calculated based on these data and 90nm TSMC technology.

5 Optimization Techniques

Reducing power while maintaining performance is currently an important issue in portable multi-media devices, since battery lifetimes can not keep up with today's increasing power consumptions of processors [8]. ADRES was designed to be power efficient, however, additional optimization techniques could be applied. There are numerous possibilities to reduce power and all are based on reducing *Dynamic* and *Static* power. Dynamic power is created by signal changes on the architectural nets, while static power depends on the physical characteristics of the design creating e.g. leakage. Static power is always present whether the nets are switching or not.

$$Total\ Power = Dynamic\ Power + Static\ Power$$

Dynamic power is the largest component of total power with the assumption the circuit is operational most of the time as is case with ADRES. When a design, however, is idle for a long period of time dynamic power reduces and the gap between dynamic and static power closes. Static power can become a significant component of total power focussing power reduction techniques on this part.

Optimization techniques can either be implemented manually by adjusting the architectural HDL code or automatically by tools on register transfer or gate level. Performance relates to timing, while energy relates to power in addition to timing. In the targeted application domain power is scares and has a higher optimization priority than performance. The power optimizations could influence the performance results and have to be investigated before implementing the optimizations.

This chapter starts by describing the components of power in an ADRES instance and depicts what can be dynamically and statically optimized. These optimization types do not refer to dynamic and static power. Dynamic optimizations are applied to the design after it has been created. The dynamic optimization in this thesis consists of operand isolation and clock gating modifying the VHDL code and/or utilizing Synopsys power tools. Static optimizations are created during the design phase consisting of pipelining, architectural modifications and memory segmentation. This thesis implemented the last two static optimizations, however, pipelining was implemented in the latest ADRESv1 by IMEC employees. Each optimization will have its own improvement results where the combination of the optimizations are implemented in the final design as noted in Chapter 6. The results in this chapter depend on the Artisan 90nm library used as selected in Chapter 4.

5.1 Power Components

The dynamic power component consists of the first two terms in the formula below [4]. The two terms are *Switching* and *Internal* power and are the two largest power

components of ADRES during regular operation. Static power consists of the last two terms in the formula, which are *Leakage* and *Static* power as is, consequently, the smallest power component of ADRES.

$$P_{total} = \underbrace{\alpha \cdot \frac{1}{2} \cdot C \cdot V^2 \cdot f}_{Switching} + \underbrace{I_{short} \cdot V \cdot f}_{Internal} + \underbrace{I_{leakage} \cdot V}_{Leakage} + \underbrace{I_{static} \cdot V}_{Static}$$

Switching power is represented in the first term where C is the capacitance of the nets and connected components, f is the clock frequency, V is the supply voltage and α is the switching activity factor, which is the amount of $0 \rightarrow 1$ and $1 \rightarrow 0$ transitions in one clock period. According to Chen et al. [4] this is accounted for 70 - 90% of total power. *Internal power* represented by the second term is caused by the short-circuit current through the P and N type transistors and is between 10 and 30% of total power consumption depending on the used library. *Leakage power* occurs when gate voltages are just below the threshold value of the transistors and *Static power* is caused when the circuit is not switching. Static power is also dissipated when current leaks between the diffusion layers and the substrate of the circuit. For this reason, static power is often called leakage power. The Synopsys tools use this terminology as well and will also be used in this thesis. The leakage power accounts for 1 - 2% of total power. The power figures for ADRES are a bit different where switching power is around 35%, internal power is around 62% and the leakage power is 3% as depicted in Figure 5.1. This chart is obtained after power estimation of the selected base architecture 4x4_reg_con_all with Artisan 90nm Nominal Vt libraries (*tsmc090nvt*) as depicted in Figure 2.26. The total power consumption and area are 80.45mW and 1.59mm^2.

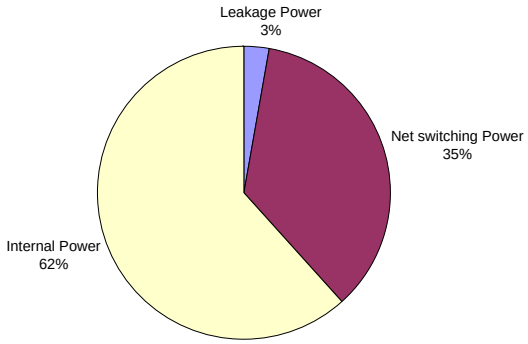
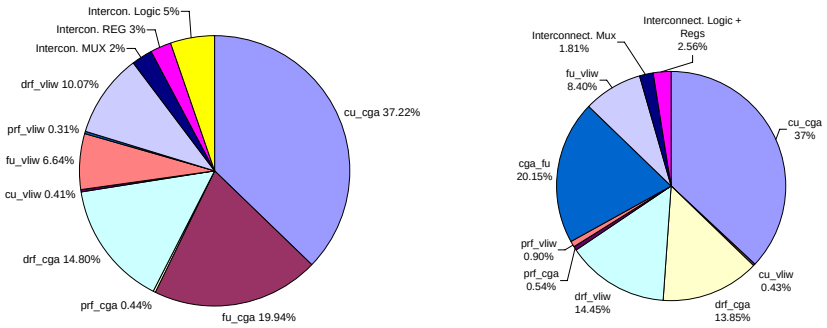


Figure 5.1: Percentages Power Components 4x4_reg_con_all

Power reduction of a design can be done by modifying one of the four terms in the formula e.g. switching power can be reduced by lowering the switching activity factor (α). The capacitance is dependent on the design area and utilized technology library. Lowering the capacitance usually requires minimizing the area by reducing the sizes of the logic components. These components become slower and performance is decreased. The same applies for voltage scaling increasing leakage, since the distance between the gate level voltage and threshold values becomes smaller. Another possibility for power reduction is to avoid the short-circuit in a circuit. Short-circuit occurs when

input signals do not arrive simultaneously introducing a glitch at the output. Delay balancing makes sure the input signals do arrive simultaneously avoiding glitches [25]. This can significantly reduce power, however, this method requires gate-level simulations of ADRES, which was not possible (Section 3.1). The following chapter will therefore only focus on reducing the switching activity.

ADRES has register files, functional units, multiplexors, interconnect networks and memories all consuming power of which the distributions of total power are represented in Figure 5.2 based on the base architecture.



(a) IDCT Power (b) Area

Figure 5.2: Power and Area distribution base architecture *4x4_reg_con_all* @ 100MHz

The power chart in Figure 5.2(a) shows that the configuration memories (cu_cga), FUs and DRFs consume most of the power in the architecture. Interesting to note is the low power consumption of less than 10% of the interconnections between modules as it was assumed in a regular design to be 10 - 30% [26]. The array is larger than the DSP processor design in [26] and not every FU of ADRES is always active. This results in relatively lower switching activity and power consumption. The VLIW control unit and predicate register files consume the least amount of power. As an optimization of the PRFs they can be replaced by predicate busses to decrease, although not significantly, area and power (Section 5.3.2).

The area chart in Figure 5.2(b) shows that the cu_cga, FUs and DRFs require most area. The area of the configuration memories have a fixed depth of 128 words during the architectural explorations, however, the register files can be altered reducing area. During the architectural modification we will investigate the effect of reducing the register files size. The FUs in the base architecture are fixed as well, however, this might change due to the optimizations implemented.

For power optimizations we focus on the configuration memories, functional units and data register files. The size of the configuration memories is rather fixed depending on the targeted applications, but by segmenting the memories power could be reduced by disabling not used parts during operation as will be explained in Section 5.3.3. The functional units can be improved by utilizing *operand isolation*, *intrinsics* and *pipelining* of which all require low-level modifications of the VHDL code. *Operand isolation* reduces switching activity in the data paths. *Intrinsics*, however, are not available in the benchmark applications FFT, IDCT and MPEG2 and are ignored as an optimization.

Pipelining improves throughput of a design. The register files are optimized with *clock gating*, which also reduces switching activity as with operand isolation.

5.2 Dynamic Optimizations

With dynamic optimizations the architecture is first created after which power and performance results are obtained [11]. Based on these results logic is modified or added to improve the results. The idea is to avoid unnecessary switching activity in components when they are in a state of idleness for a certain period of time. Preventing switching activity in components can be either done internally or externally. With internal prevention signal changes at the input do not appear at the output of the components by using e.g. blocking logic at the inputs or disabling clock inputs. Externally, the signal changes at the output are prevented to propagate further on the data path. This can be done with a multiplexor to select the proper output of the two components as depicted in Figure 5.3.

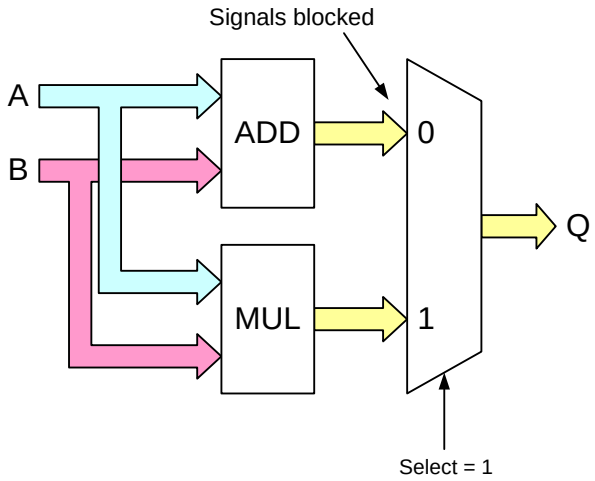


Figure 5.3: Example of External Optimizations

Two dynamic optimizations are utilized to improve ADRES power consumption: operand isolation and clock gating. Clock gating is a form of internal optimization, since the clock input is blocked to avoid state change of the connected component like a register. Operand isolation is a combination of internal and external optimizations as the outputs are analyzed to determine when the respective component is idle after which logic is added at its inputs to prevent signal propagation. The following subsection describe these techniques in more detail.

5.2.1 Operand Isolation

Operand Isolation focuses on the combinatorial logic of the data path of a design to avoid redundant computations. In an ALU there are several computational components

(shifter, multiplier, adder, etc.) capable of operating in parallel while only one the outputs is required as selected by the output multiplexor. The unused outputs are blocked by the output multiplexor, however, the inputs of all the computational components in the ALU switch consuming unnecessary power. Operand isolation directs this problem and reduces switching activity.

When focusing in more detail on the combinatorial data path of the ALU in ADRESv1 there are five different operation groups as depicted in Figure 5.4 and can also be found in Table 2.1. In this figure a *Greater-Than* (oGT) operation from the ARITH_2 set is executed. Only the data inputs (data_in_1 and data_in_2) are connected to this block, while the other inputs are kept silent by setting them to a fixed input value, hence reducing switching activity.

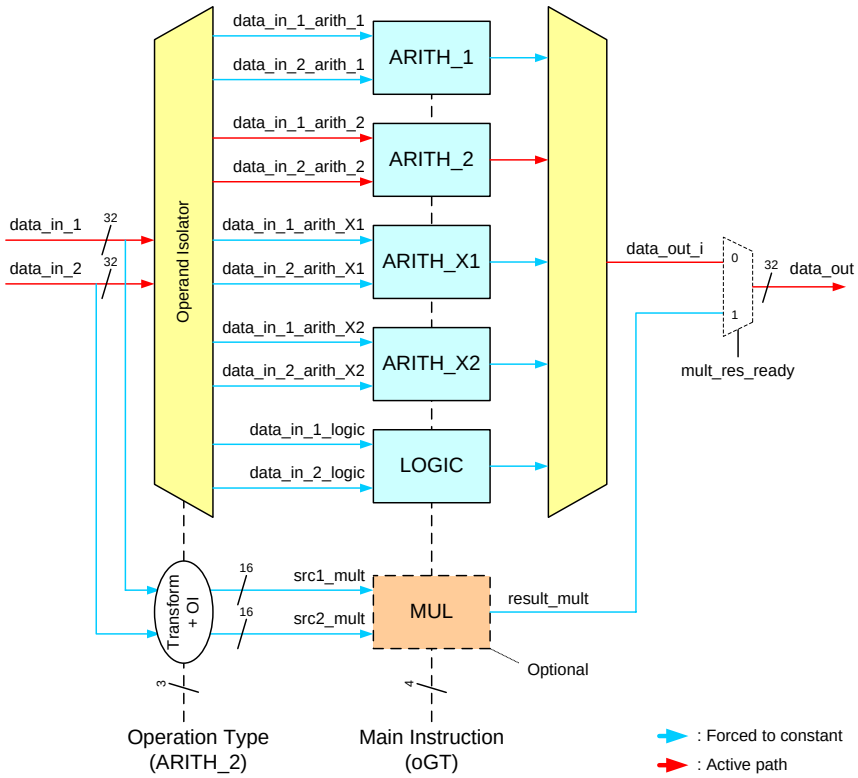


Figure 5.4: Operand Isolation

Isolation can be applied automatically utilizing Synopsys Power Compiler or manually by describing it in the VHDL code. With *automatic insertion* the tool analyzes the combinatorial data paths searching for *observability don't care* (ODC) condition sets [32]. The designer can also signal the tool where to apply operand isolation by using the *pragma* keyword in VHDL. Optionally, the tool can utilize a Switching Activity Information File (SAIF) as noted in Section 4.1 for better optimizations. Unfortunately, automatic insertion creates pre-computation logic to control the isolation logic increasing the area and delay. Since the instruction is already known at the moment the data values arrive at the inputs, this is an unnecessary calculation. *Manual insertion* re-

quires low level modification of the VHDL code. It operates in the same way as with automatic insertion, but utilizes the operation type known prior to the start of the calculations instead of the selection signal for the multiplexors. This is more efficient than the pre-computation logic utilized by automatic insertion and relatively easy to program in VHDL.

The isolation logic itself can be either done by AND/OR gates or latches. The former forces the inputs to logical zero or one, respectively, when the components are idle making it the simplest implementation. Latches keep their values and reduce power even with one clock cycle. The AND/OR implementation requires multiple clock cycles to have any power reduction as noted by Banerjee et al. [1] and Münch et al [33]. The latches have the disadvantage of being more expensive in terms of area and power overhead making the AND/OR-based implementation the most appropriate selection.

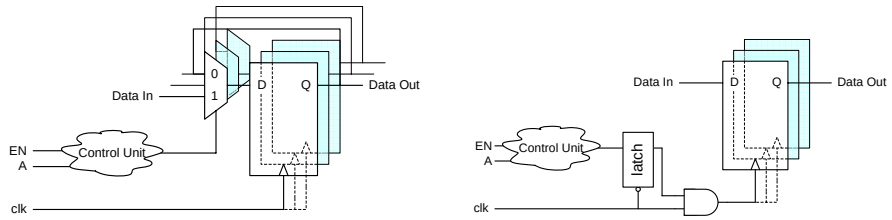
The results of applying operand isolation together with clock gating to the reference version of ADRESv0 are located in Table 6.5. Automatic implementation of operand isolation showed only a power reduction 1% in CGA mode even after applying the SAIF file during synthesis. The power results with operand isolation were even worse than without this feature. The reason was that Power Compiler was not capable of isolating all the data path input bits properly, which is avoided by manual implementation. Therefore, with the latest version of ADRESv1 the manual approach is preferred by setting the inputs of inoperable components to a fixed value. An OR-based isolation is selected as it is most power efficient according to Münch et al. [33]. With ADRESv1 power was reduced by 30% in a single function unit and 30 - 40% in overall (Section 6.4.

5.2.2 Clock gating

Unlike operand isolation minimizing power in combinatorial circuits, clock gating focuses on sequential components in a design disabling them when no value has to be written or read. The only components in ADRES controlled by a clock are the registers files and transition nodes (TRN) with a delay of 1 or more clock cycles.

An example of a register as it is implemented during synthesis is depicted in Figure 5.5(a). When the enable signal of the register is set to zero the value of the register is rerouted through the multiplexor back to its input. Although the non-zero value of the registers internal state is not changing the same value is loaded each time when the clock is changing consuming power in the multiplexor, register and clock net. Additionally, in pipelined designs clock trees with buffers are usually created to distribute the clock and control the clock skew. Since the single clock net is connected to the vast amount of buffers and registers it is highly loaded consuming a lot of power. Gating the clock avoids the unnecessary switching of the registers and reduces the load on the clock as depicted in Figure 5.5(b).

Utilizing the register file address and enable signals the clock gating control circuit controls whether the clock is forwarded or not. Note that the clock is only connected to the control circuit and not to all registers as with the traditional way. Clock gating also has a positive effect on timing and area, since the multiplexor is removed. However, the benefits are only present when applying clock gating to register banks of at least 3 bits or more [42].



(a) Traditional Register (b) Clock Gated Register

Figure 5.5: Difference Between Traditional and Clock Gated Register

Applying clock gating on the data registers as noted in Section 6.1 proved to be very effective on power with an average reduction of 50 - 80%. With the non-pipelined version of ADRES the implementation was less effective due to the single clock architecture, which was just between 6 - 8%. For ADRESv1, however, this resulted in a significant power reduction of 20 - 25% (Section 6.3). The clock tree showed a reduction of 10% in power due to the reduction in load of the design.

Clock gating can affect the testability of your design unless you add logic to enhance testability [42]. A gated register cannot be included in a scan chain, because gating the registers clock makes it uncontrollable for testing. Without the register in the scan chain, test controllability is reduced at the register output and test observability is reduced at the register input. If there are many gated registers, this can significantly reduce the fault coverage of an ADRES instance. Fortunately, Synopsys Power Compiler provides options to improve testability. Although the testability option is useful it was not the focus of this thesis and is not used.

5.3 Static Optimizations

Static optimizations are determined during the coarse of development before actual results are obtained from the design. The designer applies the optimizations not only to reduce power, but also to improve performance where dynamic optimizations primarily reduce power only. The modifications are implemented on RT level, since it has to be done manually in VHDL code.

Although several possibilities exist as noted by Hinrichs [11] only three techniques are selected that could be successfully implemented: *pipelining*, *architectural modification* and *memory segmentation*.

Pipelining is implemented to increase throughput with the same amount of energy as without pipelining. Performance can be improved even more by utilizing intrinsics, but is not utilized in the benchmark applications. These single cycle instructions are implemented to enhance performance and reduce power consumption by collapsing instructions as is done with Pentium processors with MMX technology and DSPs.

The other two optimizations are architectural optimizations based on the base architecture of Section 2.5 and segmentation of the configuration memories. The *architectural optimizations* looks at the possibilities of reducing area and power by sharing registers, while maintaining performance. Also the local register file sizes are modified with the

same objective. The *memory segmentation* option is implemented to reduce power, since the memories consume a significant percentage of total power as shown in Figure 5.2(a).

5.3.1 Pipelining

The basic idea of pipelining is to increase throughput of a design by overlapping the executing of multiple instructions each cycle (ILP). By dividing the single-cycle data path into several stages the frequency and performance are increased, however, also power. The power is even slightly higher, relatively, due to the overhead of registers and control logic [10]. Energy, however, remains the same as with the non-pipelined version, but with much higher throughput.

All the simulations of ADRESv0 are performed at 100MHz frequency as the designs were synthesized. With a minimum of 5 stages ADRESv1 should be capable of 500MHz. Unfortunately, due to the added logic, registers and delay in data memory interface this is not possible unless more pipeline stages are added. Additional pipeline stages also increase time when branches are miss predicted and the stages have to be flushed. Chapter 6 will note that the maximum clock frequency of the final architecture is about 250 - 350MHz depending on the architecture. 500MHz was the target frequency for ADRESv1, however, due to interconnection and logic delays this was not possible resulting in the lower frequencies.

Comparing the non-pipelined and pipelined power results is not fair, since the latter is optimized at architectural level which reduced in power consumption. The modifications increased performance (MIPS/mW). In addition to this, the reference architectures ADRESv0 and ADRESv1 in Section 6.2 are completely different of which the latter has more resources. Performance of ADRESv1 is in the range of 15 - 20 MIPS/mW for IDCT as will be noted in Chapter 6. To reach the target goal of 50 MOPS/mW in Figure 1.1 intrinsics have to be implemented resulting in speedups of more than 6× [9]. The benchmark applications do not use intrinsics, making it infeasible to reach the 50MOPS/mW in this thesis.

With regular pipelining power increases linearly with frequency unless a power optimization methodology e.g. operand isolation and clock gating is applied. Clock gating is normally only addressed in the register files, but this can also be applied to the pipeline registers. Individual figures are not available, but this method will be utilized in the final architecture as selected in the next section.

Although beyond the scope of this thesis, another interesting optimization for future implementation is noted by Jacobson [22] improving power with clock gating on transparent pipelining. The registers of the pipelines are transparent by default meaning they are only latched when data races have to be avoided and data values have to be separated between adjacent stages. Based on a multiply/add-accumulate unit design power reductions between 20 - 60% on top of regular clock gating are possible while maintaining performance and clock frequency despite the cost of added control logic.

5.3.2 Architectural Modifications

The base architecture selected in Section 2.5 was the most optimal one in area vs. performance, energy consumption and high performance for IDCT. The main difference

with the other architectures was the diagonal connections between functional units and local data register files. Without clock gating the power consumption of the local DRFs with 16 registers each is about 15% of total power. The architecture has a local DRF for each FU of which the DRFs are indirectly shared among diagonally, neighboring FUs. Experiments by Kwok et al. [24] also prove that the size of the local DRFs can be reduced significantly suggesting that the storage space is not utilized efficiently by the scheduler. Since the writing of that paper the scheduler has been improved and the results of Kwok et al. may not be correct for current version (DRESC2.0 and further). That is why these experiments are repeated in the context of this work.

In addition to the reduction of sizes as an architecture modification, Bingfeng Mei [27] suggests different interconnection topologies and replacing the local DRFs. Therefore, we conduct a second architectural exploration in this section based on Bingfeng Mei's architecture suggestions followed by the reduction of the register file size on the selected architecture to test Kwok's assumptions. The architectural explorations are synthesized at 100MHz as with the first exploration.

Interconnection Exploration There are two experiments suggested by Bingfeng Mei: *Distributing the local data register files* and *interconnection topologies*. In total there are 15 different architectures that will be used in the experiments. The names of the architectures are quite long and are renamed in Table 5.1 for ease of discussion with the results.

Table 5.1: Renaming Architectures of Second Exploration

Original	Renamed
4x4_mesh_plus	arch_1
4x4_mesh_plus_pred_bus	arch_1_pred_bus
4x4_reg_con_shared_2R_1W	arch_2
4x4_reg_con_shared_2R_1W_pred_bus	arch_2_pred_bus
4x4_reg_con_shared_4R_2W	arch_3
4x4_reg_con_shared_4R_2W_pred_bus	arch_3_pred_bus
4x4_reg_con_shared_8R_4W	arch_4
4x4_reg_con_shared_8R_4W_pred_bus	arch_4_pred_bus
4x4_reg_con_all	arch_5
4x4_reg_con_all_pred_bus	arch_5_pred_bus
4x4_reg_con_all_mesh	arch_6
4x4_reg_con_all_mesh_pred_bus	arch_6_pred_bus
4x4_reg_con_all_morphosys	arch_7
4x4_reg_con_all_morphosys_pred_bus	arch_7_pred_bus
4x4_reg_con_shared_2R_1W_morphosys	arch_8

The first experiment has a Mesh Plus architecture as basis while the local DRFs are distributed over the array as depicted in Figure 5.6. The idea is to determine the impact of creating a central storage location among four FUs (*reg_con_shared*) or even removing the local DRFs completely (*mesh_plus*). Notice the removal of the diagonal connections between the first two rows in architecture *reg_con_all* in Figure 5.6 compared to the basis

architecture in Figure 2.26. This is done to avoid connections from the second row FUs with the global DRF, which was not the case for the basis architecture.

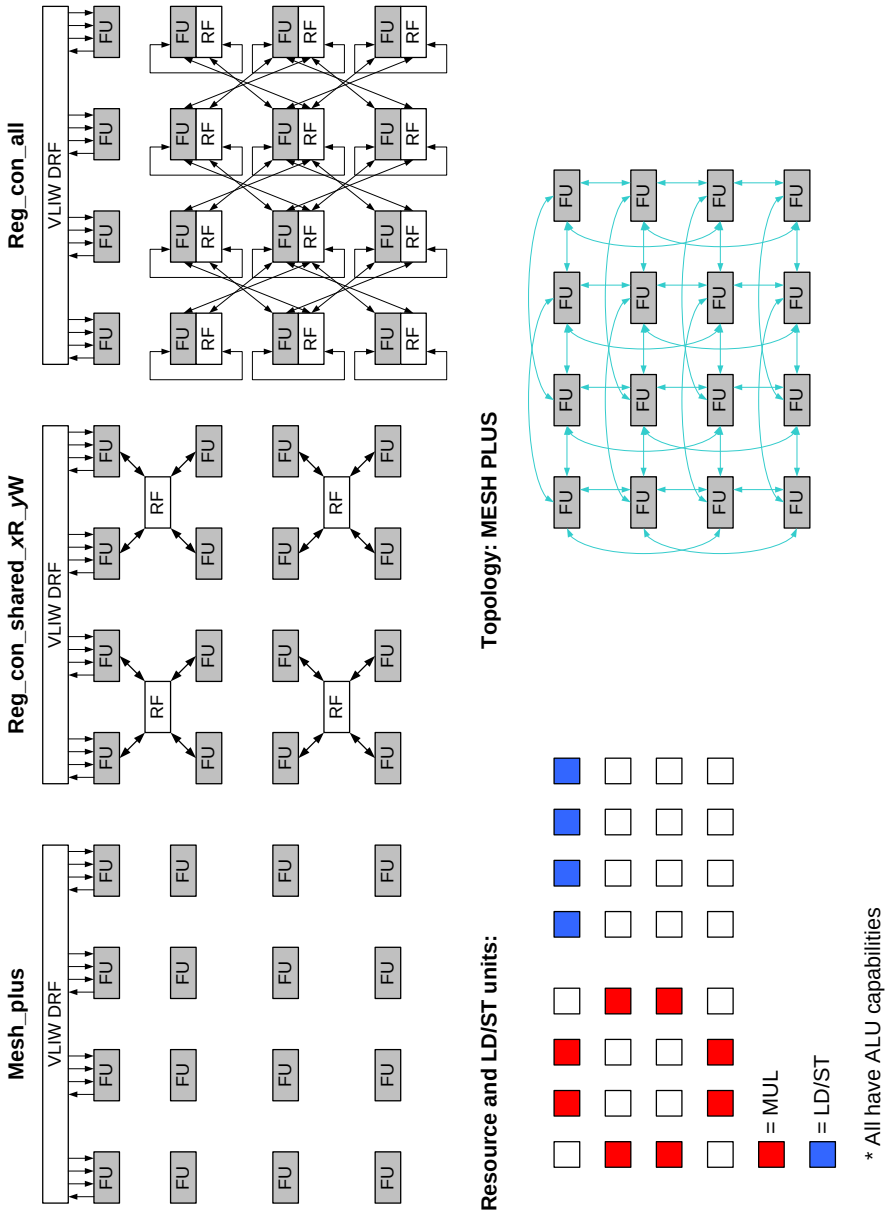


Figure 5.6: Distributing the Local Data Register Files

The second experiment has the reg_con_all architecture as basis, while the horizontal and vertical connections are modified: *Mesh*, *Mesh Plus* and *Morphosys* as depicted in Figure 5.7. The last option is not tested in Section 2.5 and is based on the Morphosys architecture as described by Singh et al. [40]. Note that the *reg_con_all* in Figure 5.6 and *mesh_plus* in Figure 5.7 are similar and will only be depicted as 4x4_reg_con_all in

the results.

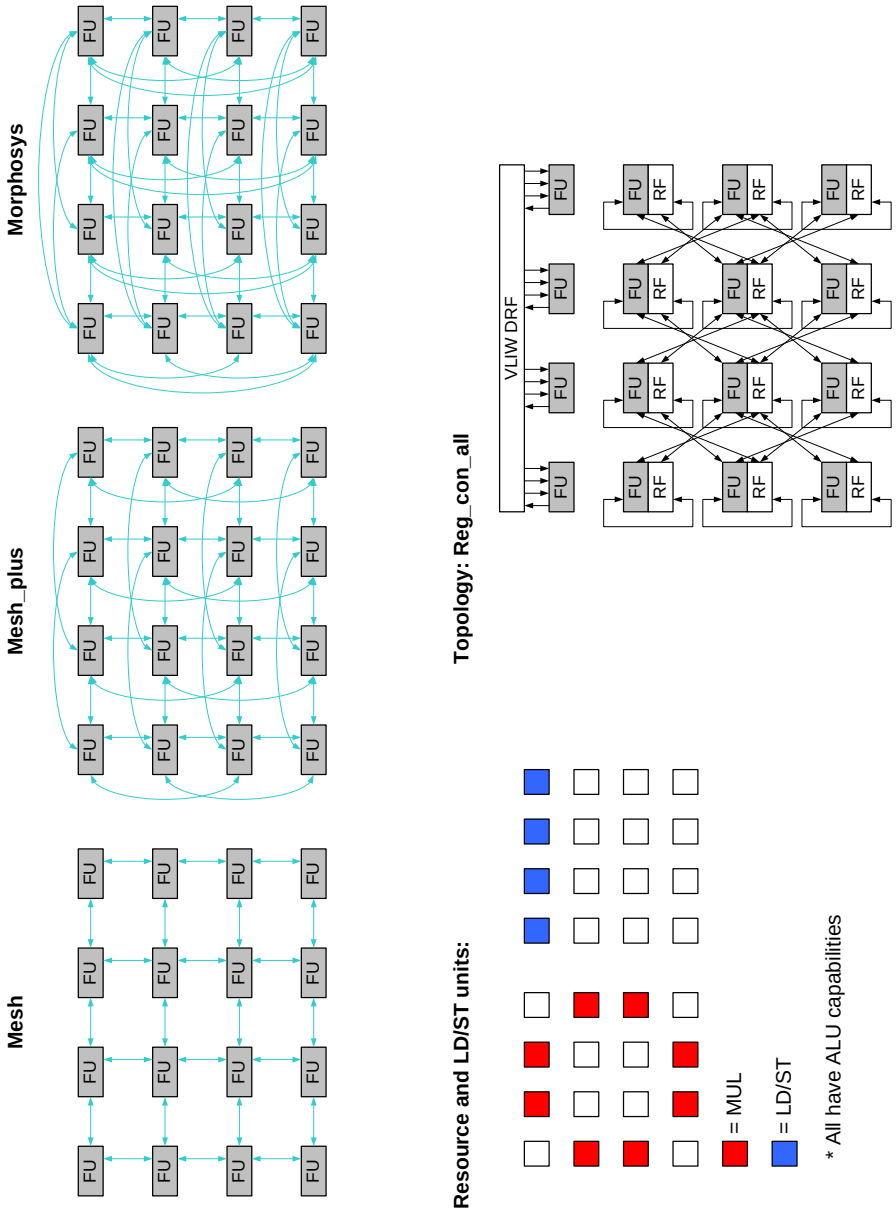


Figure 5.7: Interconnection Topologies

The architectures created in both experiments have local predicate register files or predicate busses (noted with suffix *pred_bus*). Since the predicates are only one bit wide the replacement of the registers could have a positive effect on area and consequently leakage, but do not have the capability to store values for later process. The effect of replacing the local PRFs with busses is noted later in this section.

The results of Bingfeng Mei's experiments are based on the same characteristics as

the first architectural exploration as noted in Section 2.5.1. Again the *np_1x1_reg* is depicted in the charts as a comparison, which is in terms in area and leakage most optimal. MIPS/mW is also high, but with a drastic reduction in performance compared to the other architectures.

The first ten bars (arch_1 to arch_5_pred_bus) in the charts represent the distributed register file experiment, while the others (arch_6 to arch_7_pred_bus) are part of the interconnection topology. The last bar is the select architecture with reduced register sizes as explained later.

The architecture selected by the distribution of the local DRFs and interconnection topologies experiments resulted in a power reduction of 24.5% and 27% for IDCT and FFT, respectively. Energy was reduced by 25% and 22% for IDCT and FFT, respectively.

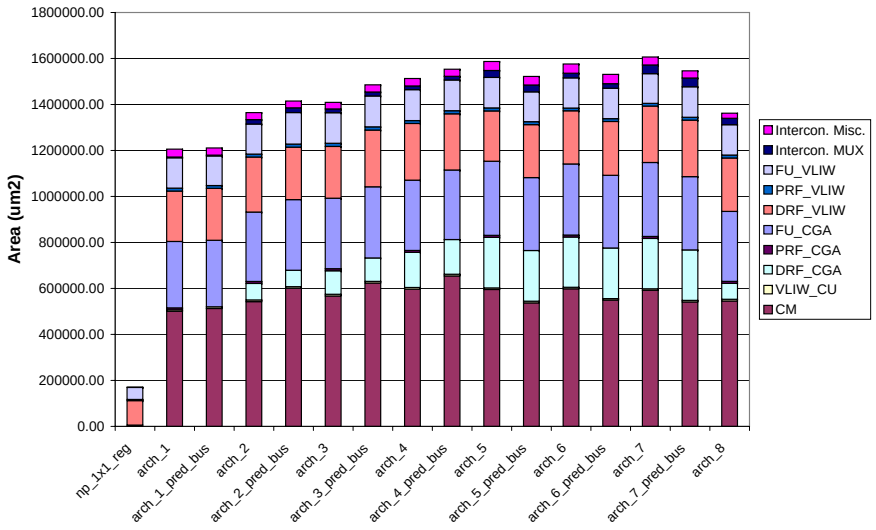


Figure 5.8: Area of Second Architectural Exploration

Distributing Local Data Register Files Removing all the local DRFs as in *arch_1* result in lower power and low leakage, however, with bad performance. When replacing the local PRFs with busses as well results in the highest energy consumption of all architectures for FFT, since this application relies heavily on registers.

Sharing the registers can be considered as an additional interconnection between the FUs with storage capabilities. By replacing four register files with one area and leakage of the DRFs is reduced. Their number of ports are also varied from 1, 2 and 4 write ports. Except for the FFT chart the shared register file with 1 write and 2 read ports (*arch_2*) is most optimal and is even better than the *arch_5* architecture with fully distributed local DRFs.

Interconnection Topology For the interconnection topology experiment (right side of charts) it becomes immediately clear that more connections is better as noted as well by Kwok et al. [24]. In general this is true and the effects of Morphosys have

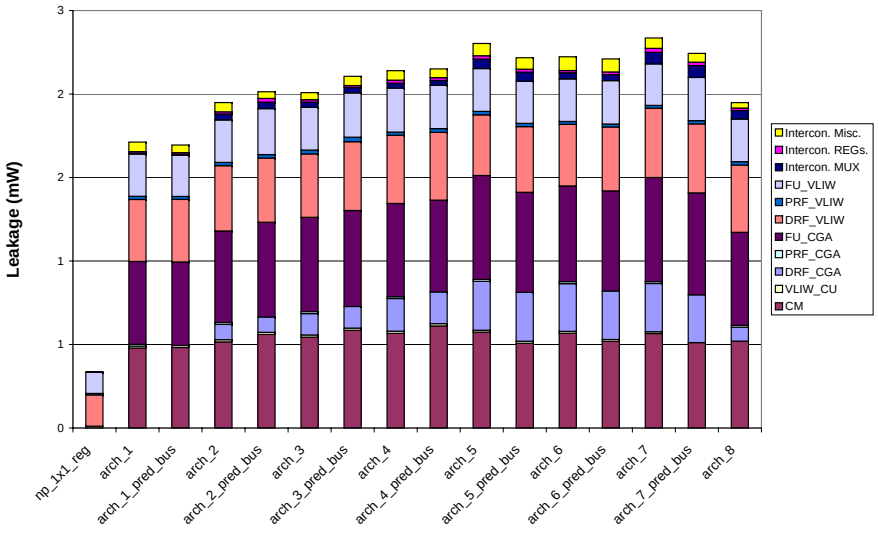


Figure 5.9: Leakage Power of Second Architectural Exploration @ 100MHz

become clear. Replacing the predicate register files with busses also have a positive effect on performance and energy making the *arch_7_pred_bus* the most optimal selection in the interconnection topology experiment.

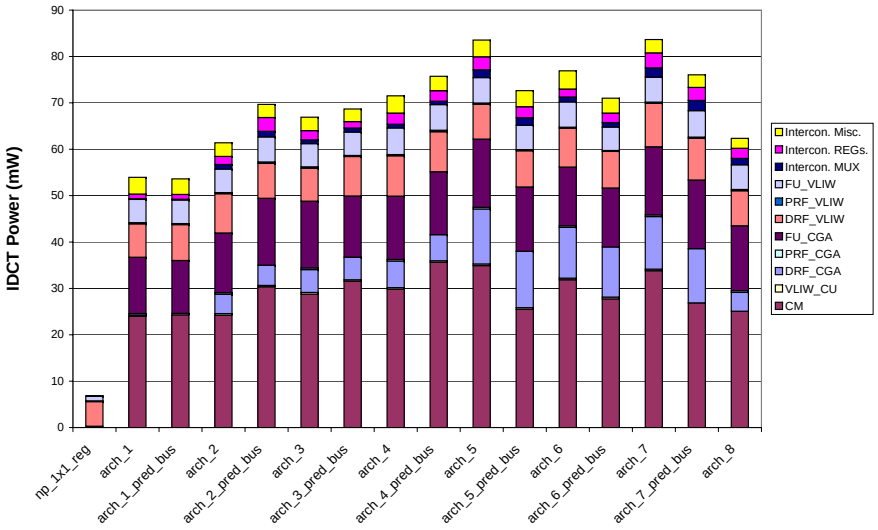


Figure 5.10: IDCT Power consumption of Second Architectural Exploration @ 100MHz

Combining the two most optimal architectures of the previous paragraphs would result in an architecture with shared data register files, Morphosys connections and predicate busses. The architecture arch_2_pred_bus, arch_3_pred_bus and arch_4_pred_bus, however, showed that the the predicate busses have a negative effect on energy and performance. Therefore, the predicate register files are kept in the final architecture. The results of the combination are depicted in the last bar in the charts marked as

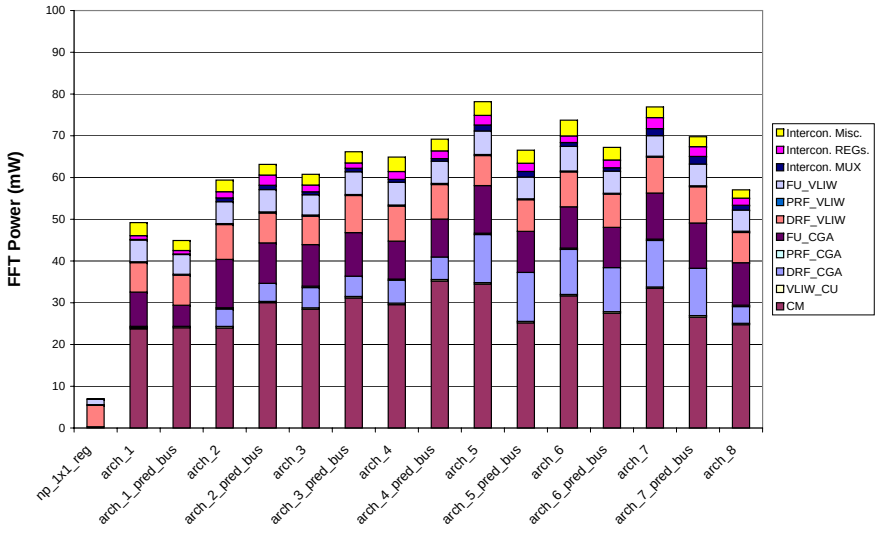


Figure 5.11: FFT Power consumption of Second Architectural Exploration @ 100MHz

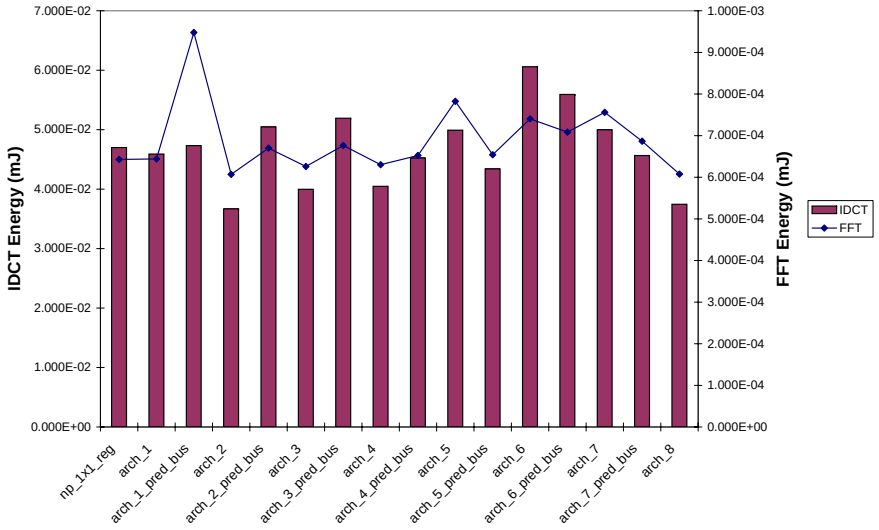


Figure 5.12: Energy consumption of Second Architectural Exploration @ 100MHz

arch_8. The architecture without the Morphosys option is minimally superior, however, the connection is sustained in the 4x4 architecture for MPEG2 simulations due to the higher load simulations and the 8x8 architecture for better routing. Figures 5.17 and 5.18 depict the energy-delay charts where arch_2 requires least amount of energy followed by arch_8. Additionally, notice the large difference between arch_1 and arch_1_pred_bus for the FFT benchmark. It clearly shows the negative effect of predicate busses for architectures without any local DRFs.

Table 5.2 shows the differences between the base architecture and arch_8. The figures of arch_8 are all improved over the base architecture. Sharing the local DRFs reduces

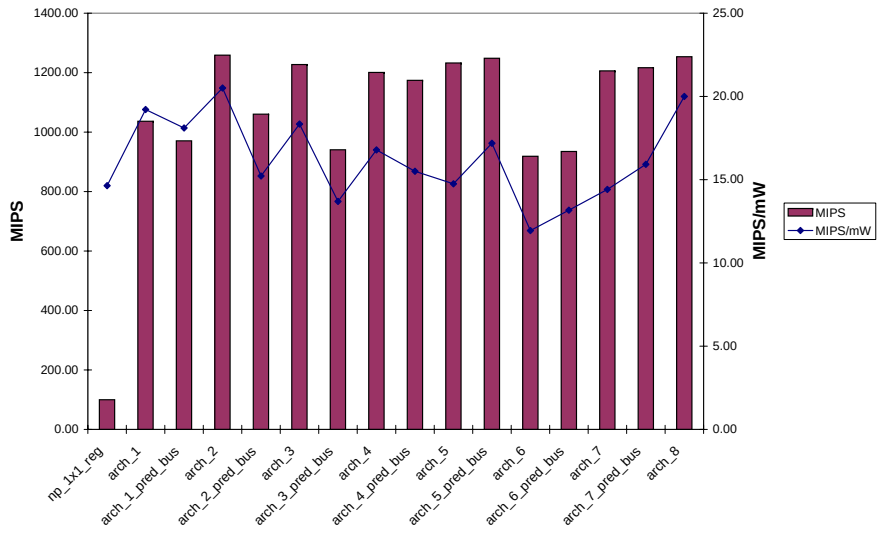


Figure 5.13: IDCT Performance of Second Architectural Exploration @ 100MHz

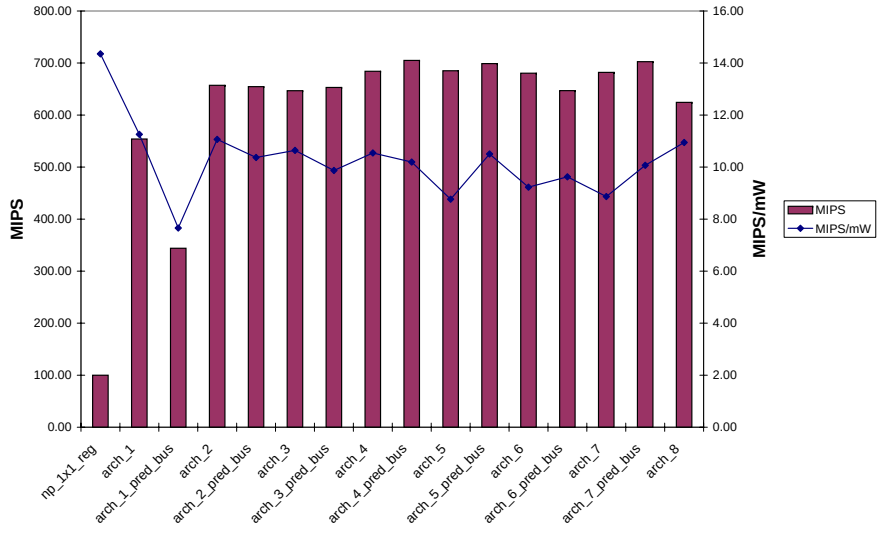


Figure 5.14: FFT Performance of Second Architectural Exploration @ 100MHz

area as well by 14.4%. The energy consumption of the architectures are almost the same, which is caused by the increased number of cycles with arch_8. Power consumption, however, is reduced by 22% proving the effectiveness in power and performance for this architecture. It is merged with the optimizations as explained in this chapter into the final architecture. The results of that architecture will be provided in Chapter 6.

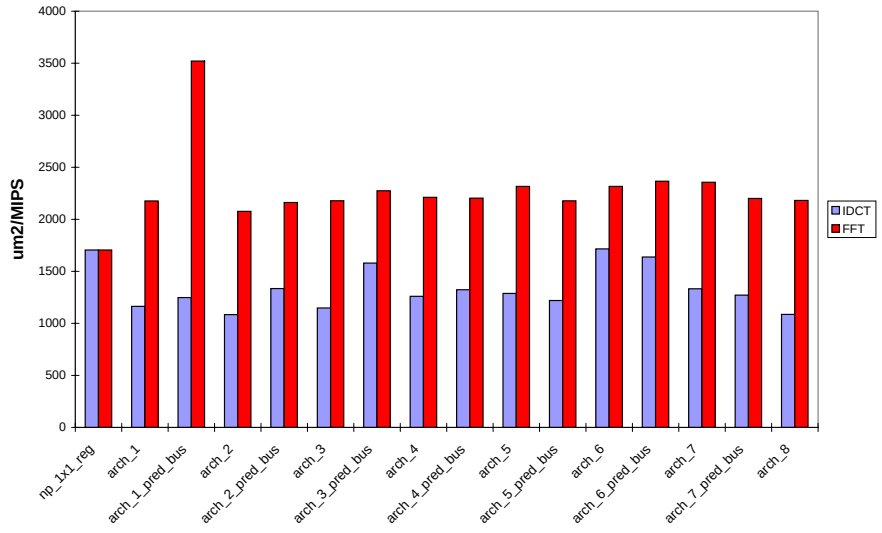


Figure 5.15: Area vs. Performance of Second Architectural Exploration @ 100MHz

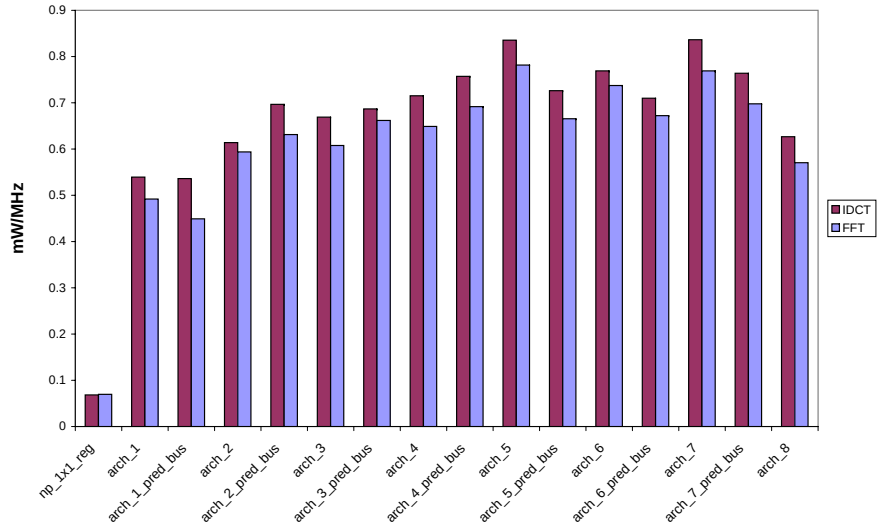


Figure 5.16: Power vs. Frequency of Second Architectural Exploration @ 100MHz

Register File Size Modifications To validate the proposed register file reductions by Kwok et al. [24] similar experiments are performed on the architecture selected in the previous paragraph. The depth of the local PRFs is equal to that of the data register files. Due to Esterel simulator restrictions the size of the global DRF is fixed to 64 registers. The empirical results we obtained after IDCT and FFT simulations are placed in Table 5.3.

Looking at the number of instructions, cycles and IPC for both IDCT and FFT in Table 5.3 the optimal number of register is 4 for a 4x4 architecture as marked in the red shaded column. The results in the first column (2 registers) show an increase of cycles

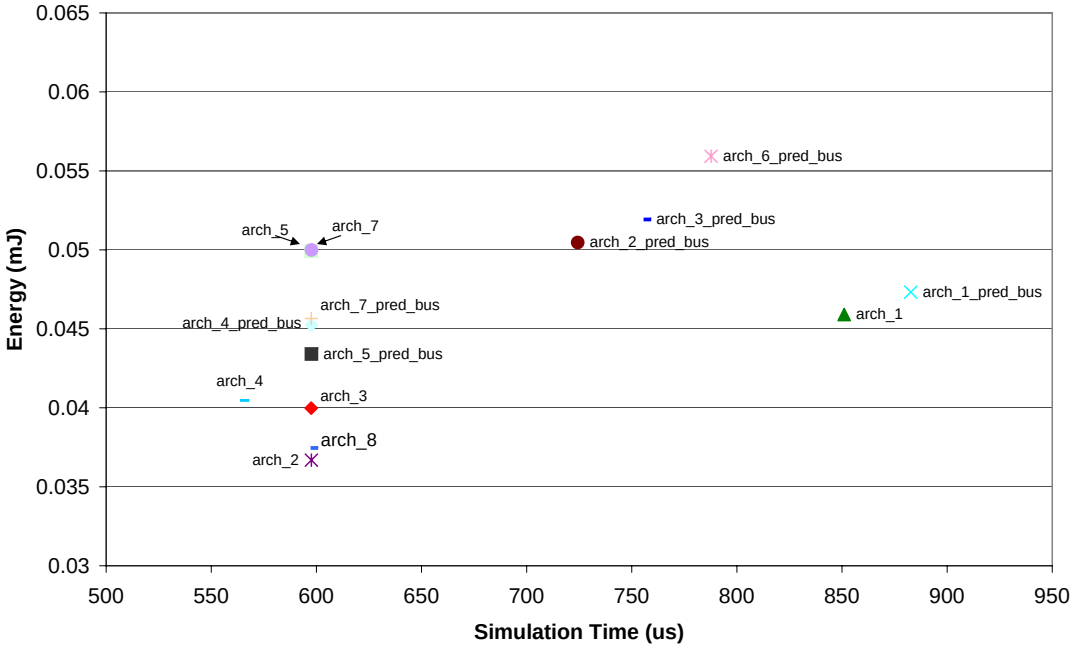


Figure 5.17: IDCT Energy-Delay of Second Architectural Exploration @ 100MHz

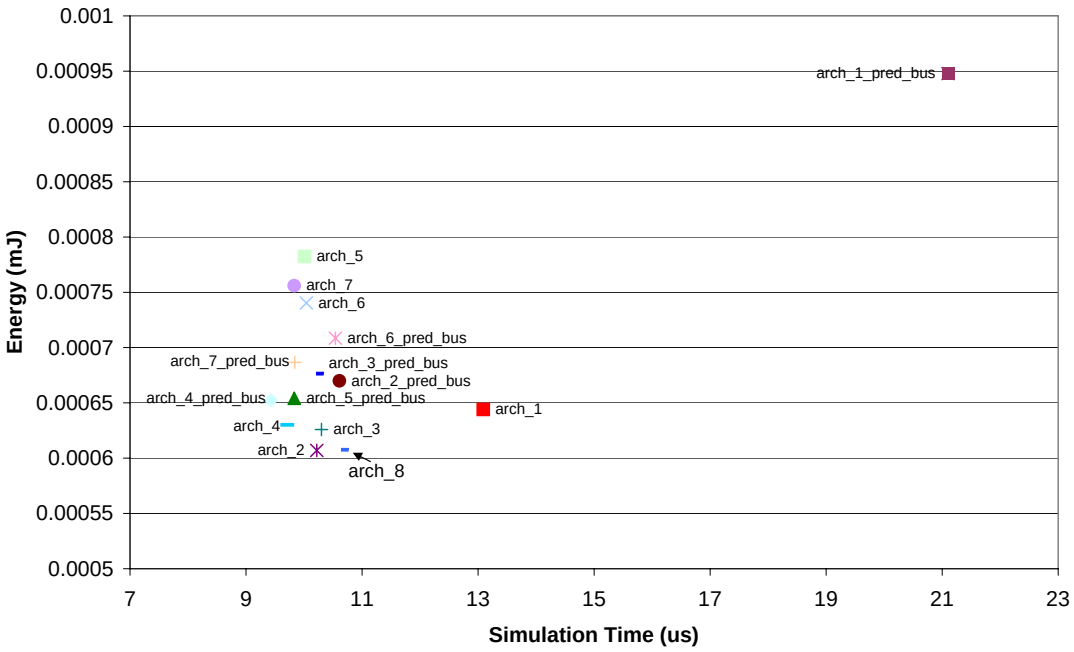


Figure 5.18: FFT Energy-Delay of Second Architectural Exploration @ 100MHz

and instructions and a decrease of the IPC. Increasing the amount of registers showed no improvements. Kwok et al. concluded that the local register files obtain optimal results with just 1 or 2 registers. Although these tests are not as elaborate as Kwok's method

Table 5.2: Differences between 4x4_reg_con_all and arch_8 for IDCT and FFT

Benchmark	MIPS/mW	mW/MHz	Power (mW)	Energy (uJ)	Area mm^2
IDCT					
4x4_reg_con_all	17.51	0.81	80.45	37.72	1.59
arch_8	20.00	0.63	62.68	37.46	1.36
Improvement	14.22%	22%	22%	0.6%	14.4%
FFT					
4x4_reg_con_all	9.40	0.72	73.28	0.62	
arch_8	10.95	0.57	57.05	0.61	
Improvement	16.5%	20.8%	22.1%	1.6%	

Table 5.3: Reducing Register File Size arch_8

Application	Local Register File Size 2	4	8	16
IDCT				
Instructions	974632	923940	923980	923960
Cycles	62924	59755	59762	59757
IPC	9.69	10.21	10.21	10.21
FFT				
Instructions	11364	10532	11040	11060
Cycles	1087	1035	1063	1065
IPC	2.48	2.73	2.57	2.58

it proves that the scheduler has improved over time. However, the number of registers in the register files is surprisingly low. For an 8x8 architecture it is suggested only to utilize 1 register, however, the same amount of 4 is utilized assuming this provides better results. The registers that are not utilized will not switch due to the clock gating feature.

5.3.3 Memory Segmentation

The configuration memories consume about 35 - 40% of the total power. This is quite significant, however, with a simple adjustment power can be reduced according to Gadelrab et al. [7]. The idea is to segment the single port configuration memories rowwise in equal sizes as depicted in Figure 5.19. By disabling all the sections that are not active only the section being accessed would consume power. This method has advantages for ADRES due to the principles of modulo scheduling applied where a loop iterates through a range of words (II) in the memories for the duration of the number of loop iterations. If the II of a loop is low enough only one segment would be active while the others are inactive.

The outputs of the sections have to be connected to the data output bus by a multiplexor as depicted in Figure 5.19. The size of the multiplexor depends on the number of memory sections connected to it. When the amount of memory sections double, so does the size of the multiplexor. Power reduction is therefore dependent on the combined power of the memory sections, multiplexors and additional logic until a point is reached

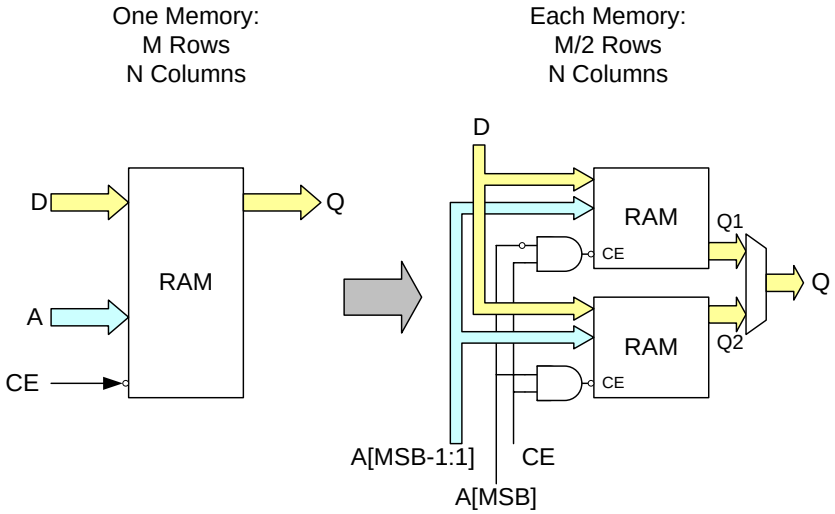


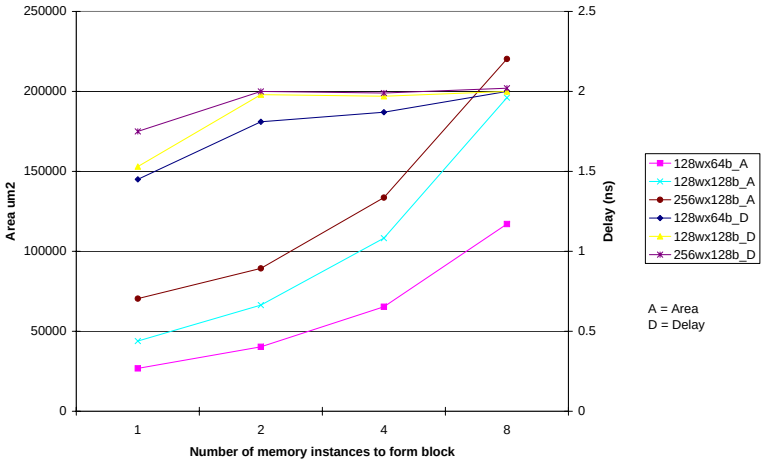
Figure 5.19: Memory Segmentation to save Power

where no more reduction is possible due to the additional circuitry.

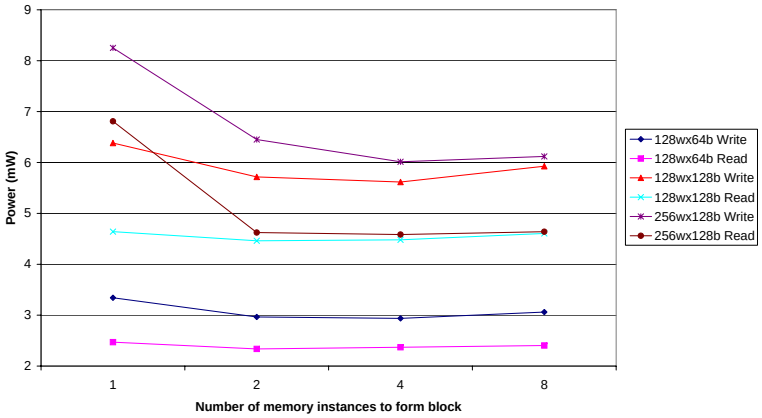
A total of three configurations 8, 16 and 32kB memory sizes are created, which are 128wx64b, 128wx128b and 256wx128b, respectively. Each memory is segmented in 2, 4 and 8 sections in which a binary file is loaded. This binary file is used to load the configuration memories for the MPEG2 application and has 53 lines. Since the configuration memories are written once and read multiple times during CGA execution, the simulation is constructed in the same way. Though the configurations are capable of frequencies around 500MHz, the utilized simulation frequency is 250MHz to avoid potential timing violations. The results of the gate-level simulations at this frequency are depicted in Figure 5.20. Note that a write operation always requires more power compared to a read operation, since the written data is always forwarded (write-through) to the output port in the next clock cycle consuming power as well.

As expected, area and delay increase as noted in Figure 5.20 when splitting up the memories due to the additional logic, multiplexors and column/row decoders. For area this can be as high as 200% and 15% for delay when segmenting the memory in 8 parts. Splitting up the memories in 2 and 4 sections themselves always reduce power for both read and write operations as depicted in Figure 5.20(b). With small memories these power savings are canceled due to the power consumption of the additional logic as depicted in Figure 5.20(c). Since the selected memories in Section 6.1 are already power optimized segmentation of the small memory sizes do not give any improvements unlike with bigger memories as the *256wx128b*. With 4 segments read and write power reduction is 28% and 23%, respectively, with an area increase of 89%. This area increase is significant and an intermediate solution would be more proper. Partitioning the 256wx128b into 2 segments gives a power reduction of 28% for a read and 18% for a write operation at the cost of 27% increase in area.

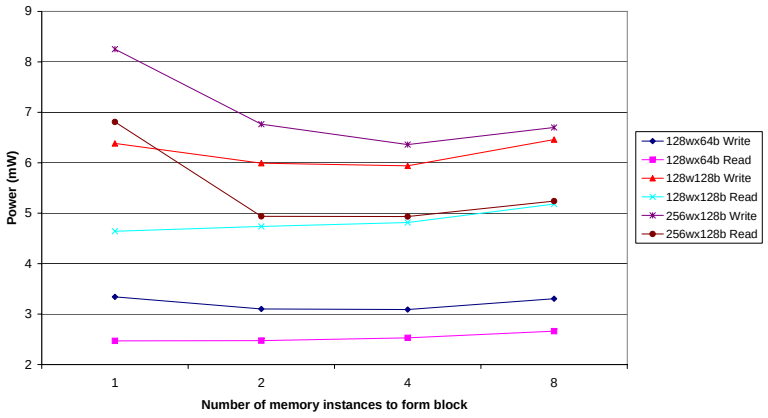
The configuration memories have 128 words for a 4x4 ADRES instance, which is sufficient for MPEG2 simulations. Memory segmentation is therefore not required in this



(a) Area and Delay



(b) Pure Memory Power



(c) Total Power

Figure 5.20: Result of Memory Segmentations with MPEG2 configuration file at 250 MHz

instance. The successor standard of MPEG2 for verification of ADRESv1 is MPEG4 AVC H.264 requiring at least 256 words in total, which would make the optimizations valuable. The same idea can also be applied to the instruction cache and data memory interface each having at least 512 words. Looking at Figure 5.20(a) the maximum frequency when segmenting the memory in 2 part is around 500MHz. This is well above the maximum frequency (312MHz) of the final architecture of ADRESv1 as will be noted in Section 6.5.

5.4 Summary

Power consists of dynamic and static components. The dynamic power component is important when ADRES is operational most of the time. The static power component becomes important if the chip is idle most of the time. In this thesis only the dynamic power component of ADRES is considered.

Dynamic optimizations are applied after the design phase, which are operand isolation and clock gating reducing switching activity. *Operand isolation* focusses on the data path of a functional unit and resulted in power reductions of 30 - 50%. *Clock gating* focusses on the clock signal and the registers connected to it. The load on the clock net is reduced and unnecessary switching of the registers is avoided. Power reductions of 20 - 25% were obtained.

Static optimizations are applied during the design phase, which are pipelining, architectural modifications and memory segmentation of the configuration memories. *Pipelining* improves throughput of a design while maintaining the energy level compared to non-pipelining. The *architectural modifications* changed the distribution of the local register files by sharing them among four diagonally neighboring FUs. Power reductions of 22%, performance increase of 14 - 16% and area reduction of 14.4% are obtained compared to the base architecture. *Memory Segmentation* splits up the configuration memories in a rowwise fashion. Advantages are only noticeable with 256wx128b or larger sizes into two sections resulting in power reductions of 18 - 28% for a write and read operation, respectively. Area of the memory is increased by 27%.

The improvements on the dynamic power component proved to have significant advantages. They are applied to the base ADRES instance with minimum penalty in performance. Pipelining and the architectural modifications will increase performance even further when applied in the final architecture.

6 Detailed Architecture Results

The dynamic and static optimization techniques described in Chapter 5 resulted in reasonable improvements in both power and performance. The dynamic optimizations e.g. operand isolation and clock gating were verified on two operational reference versions: the non-pipelined (ADRESv0) and pipelined (ADRESv1) versions. The static optimizations, architectural explorations and memory segmentation, were either evaluated individually or compared to the base architecture as selected in Section 2.5.

The architecture as selected in Section 5.3.2 is to be augmented with the optimization techniques providing the final power and performance results. The architectures during the coarse of the project are used as "milestones" to note the improvements obtained with the optimizations.

All the synthesis and power results are based on 90nm TSMC technology of which the appropriate library has to be selected based on power and performance. The two libraries available are from Artisan and Synopsys.

This chapter starts by selecting the appropriate 90nm TSMC library used for synthesis and power estimations. Next, the reference architectures and benchmark applications are briefly described. The results of clock gating and operand isolation implementation are depicted based on the reference architectures. The optimizations in Chapter 5 are merged together creating the final architecture for the ADRES core as noted in Section 6.5. In addition, based on data sheets of memories and the results of the final ADRES core power estimations are made for an ADRES processor consisting out of the ADRES core, data memory interface and the instruction cache. Finally, a comparison is made with a scalable VLIW.

6.1 Library Selection

Both the synthesis and simulation flows are dependent on 90nm TSMC libraries. There were two vendors available to select from: Artisan (CLN90G) [13] and Synopsys (tcbn90ghpnvt) [19]. Based on power and performance a selection between these has to be made.

As depicted in Figure 5.2(a) the configuration memories, register files and functional units consume most power. Therefore, these three components are used for library selection where timing also has to be considered.

6.1.1 Configuration Memories

The configuration memories can be implemented by Artisan CLN 90nm generated macro single-port register files, SRAMs and synthesized RFs of which the latter is synthesized from Artisan standard cells. The Synopsys libraries do not have generated macro register

file, but only standard cell libraries. In this section we only focus on Artisan libraries. The generated macro SRAMs have a minimum depth of 256 words, while only 128 are required for our benchmark applications. The SRAM application notes also revealed that the SRAMs require more power and area compared to macro RFs. Since we want to reduce power consumption as much as possible we don't have to consider the SRAMs.

The synthesized RFs are more flexible in bit width, but the macro RFs are optimized by the vendor for power and performance compared to what can be created by synthesizing the RFs. Area, timing and power charts are created based on a variety of bit widths as depicted in Figure 6.1 for only the generated and synthesized register files. The register files are generate/synthesized at a frequency of 500MHz. The power and performance figures in the Artisan data sheets are confidential, hence no absolute values can be printed in this thesis. Instead, the differences relative to the generated macro RFs are depicted in Figure 6.1.

The area and power charts show that the synthesized register files differ between 200 - 1100% compared to the macro RFs. Especially with large RF sizes the advantage of the macro RFs is noticable. Comparing the read and write operations in Figure 6.1(c) show that a write operation relative consumes more power than a read, however, this changes when the bit width of the register file increase above 32 bits. The synthesized register files showed better results with these sizes. This can also be noticed in the timing chart in Figure 6.1(b) where the synthesized RFs obtained better timing results with a bit width of 64 bits.

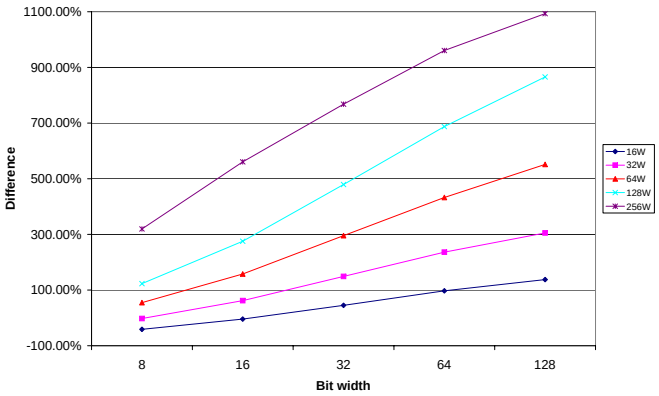
Except for timing the macro RFs are superior in both area and power. However, the largest generated RF (256wx128b) is able to operate at 500MHz. With this high frequency timing is not a problem in an ADRES instance. Based on power and area the Artisan macro RFs are the most optimal selection to be utilized for the configuration memories.

6.1.2 Register Files

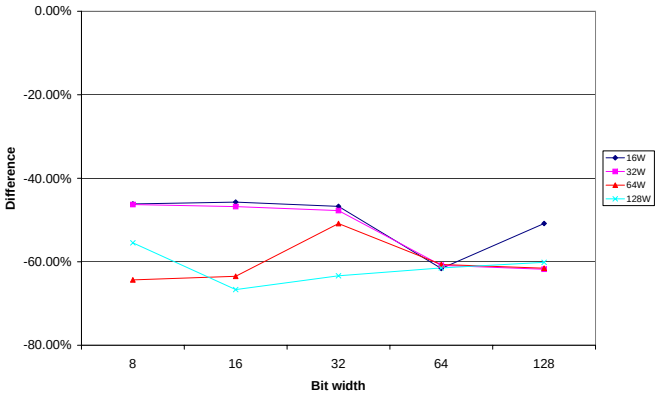
The register files consume about 28 - 30% of total power according to Figure 5.2(a). These have multiple ports ranging from 3 to 12 ports with a ratio of 2:1 for read and write ports. The multi port register files are synthesized with standard cells. The empirical experiments are based on the same methodology as utilized by Raghavan et al. [35]. The registers are all synthesized for the same clock frequency of 100MHz, which was the target frequency for the non-pipelined version of ADRES at that moment. Frequencies of 400 - 500MHz, however, are also possible for register files with 12 ports and 64 register words.

The predicate and data register files can be divided in VLIW and CGA sections and synthesized with or without clock gating. Since clock gating is to be utilized anyway the selection for the register files are based on the synthesis results with clock gating. Section 5.2.2 mentions this feature has no advantage when bus widths are 3 bits or smaller, hence the results of the PRFs are omitted.

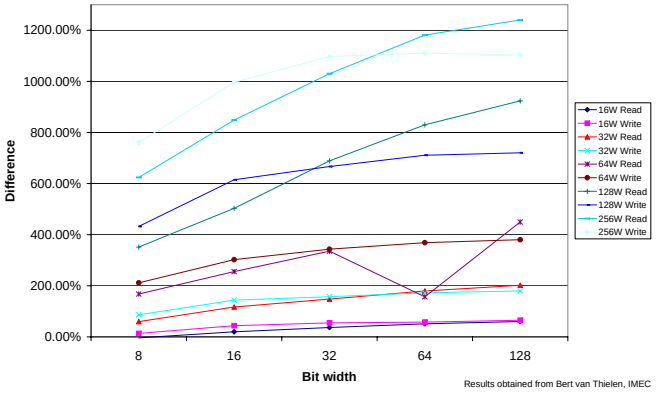
Two different gate-level simulations are performed on the DRFs of which testbench1 (TB1) accesses all ports with a write and read operation, while TB2 only accesses the first port of the register file. All the results are depicted in Appendix E due to vast



(a) Area



(b) Timing



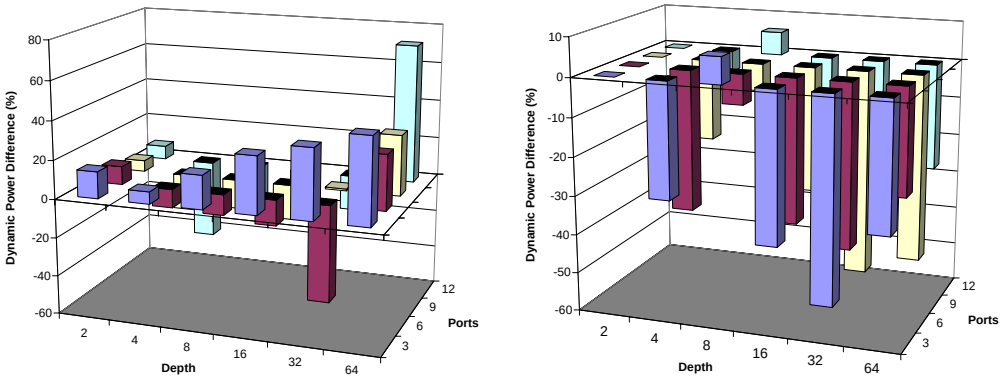
(c) Power

Figure 6.1: Register File Type selection for Configuration Memories

amount of data obtained, but some of those figures based on TB2 are depicted in this section. A positive value in the chart is in favor of Artisan, while a negative value is in favor for Synopsis libraries.

The write and read results for the CGA DRFs as depicted in Figure 6.2 show that

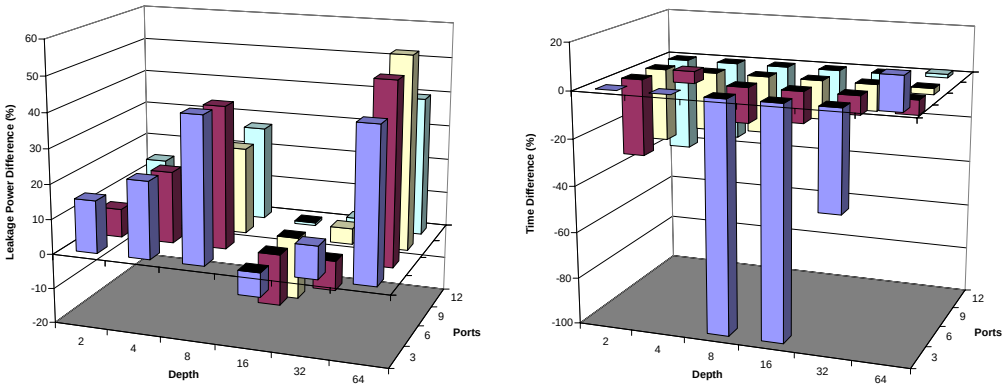
Artisan is only optimal for write operations. A write operation is also an asynchronous read as with the macro RFs in Section 6.1.1. That makes the write power value larger than a read focussing our attention at the write results. Although not shown in this section, the VLIW DRFs with the Synopsys libraries are most optimal for six or more ports, which is usually the case for an ADRES instance.



(a) Write (b) Read

Figure 6.2: CGA DRF differences between Synposis vs. Artisan for when accessing first port only

The leakage and timing results in Figures 6.3(a) and 6.3(b) show that Artisan has less leakage (10 - 30%) than Synopsys, but are slower as well (20 - 40%). Similar results can be found for the VLIW DRF, however, with 6 or more ports the leakage power is also in favor for Synopsys.



(a) Leakage (b) Time

Figure 6.3: CGA DRF differences between Synposis vs. Artisan

Based on the results the Synopsys libraries are preferred for both timing and leakage power, while Artisan libraries are preferred for dynamic power of write operations. The power values of the Synopsys libraries were not complete, which made the libraries questionable. Since I wanted to have reliable figures, I selected the Artisan libraries

for the architectural explorations with ADRESv0. The Synopsys libraries are selected for ADRESv1, since maximum performance was preferred by IMEC/DESICS for this instance.

The libraries are selected with clock gating already implemented. However, it is interesting to know what the maximum power reduction is compared to a design without clock gating. The maximum power reduction is obtained when accessing one port (TB2) only. A write operation consumes more power than a read operation as noted earlier and it is interesting to depict what the maximum power reduction could be (Figure 6.4). The read operation and leakage are added as well for comparison in Figures 6.5 and 6.6. The synthesis results are based on Synopsys standard cell libraries.

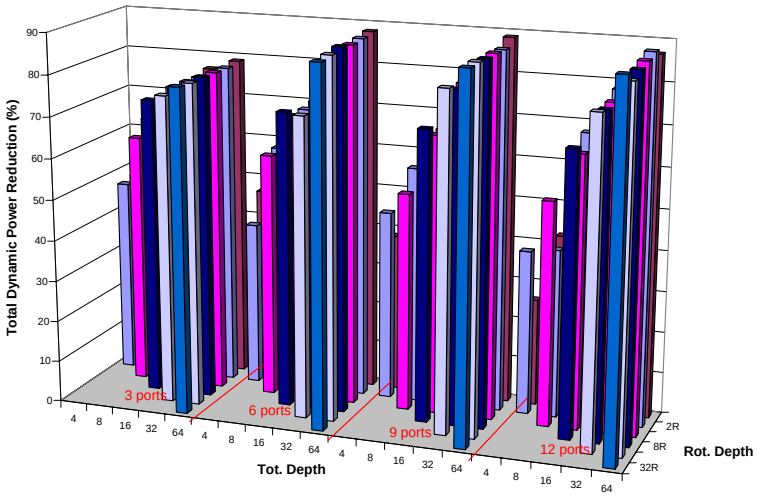


Figure 6.4: Write power reduction with clock gating for VLIW DRF

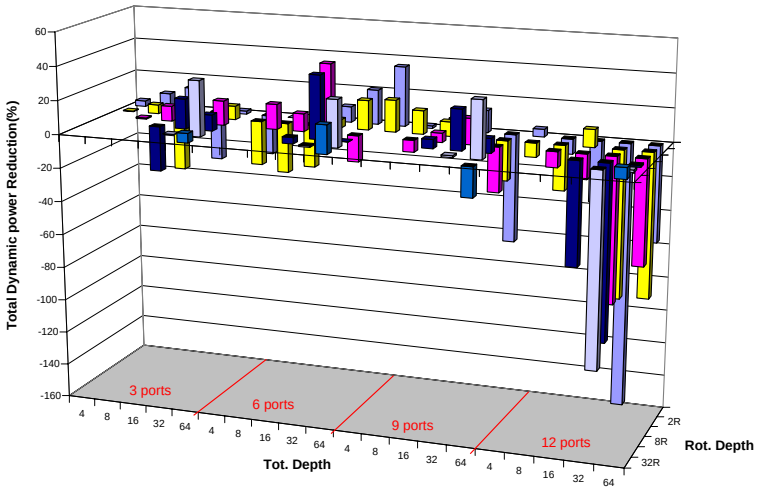


Figure 6.5: Read power reduction with clock gating for VLIW DRF

A write operation can be reduced in power as high as 80% when the register file

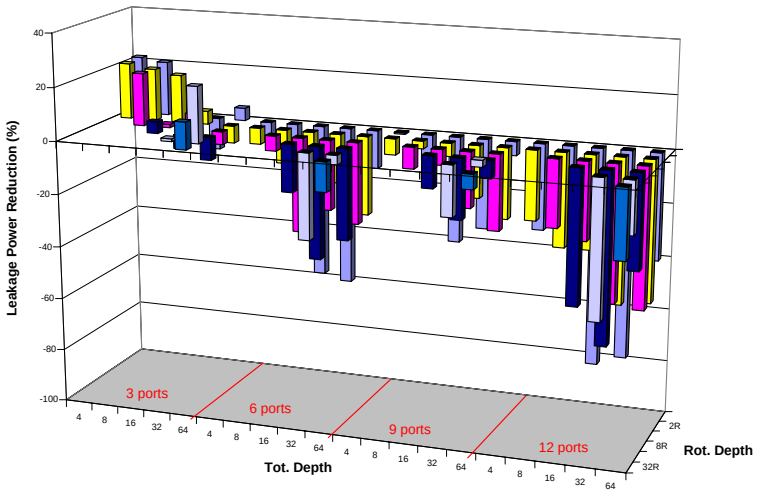


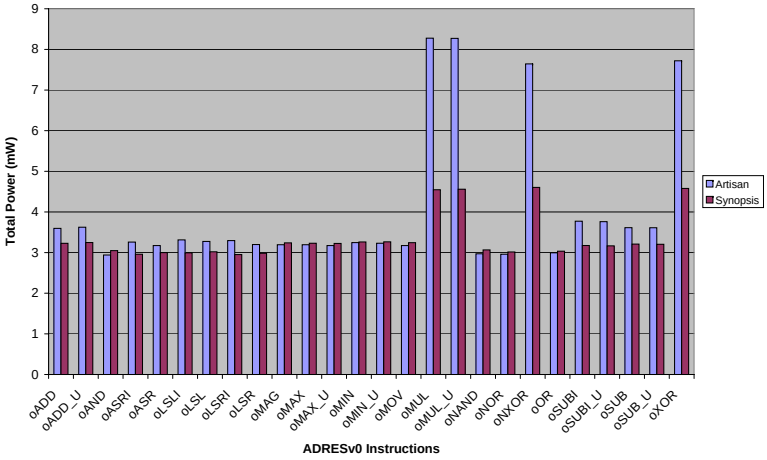
Figure 6.6: Leakage power reduction with clock gating for VLIW DRF

has between 32 and 64 words. For a read operation power reduction is negligible or even negative. Unlike a write operation, a read operation is not clocked, discarding the clock gating feature. Figure 6.6 depicts the leakage reductions, which is negative due to the added control logic. This was not expected, since area is reduced by removing the multiplexors as noted in Section 5.2.2. The control logic became larger than the removed multiplexors. Nevertheless, clock gating is proved to be a good dynamic optimization technique.

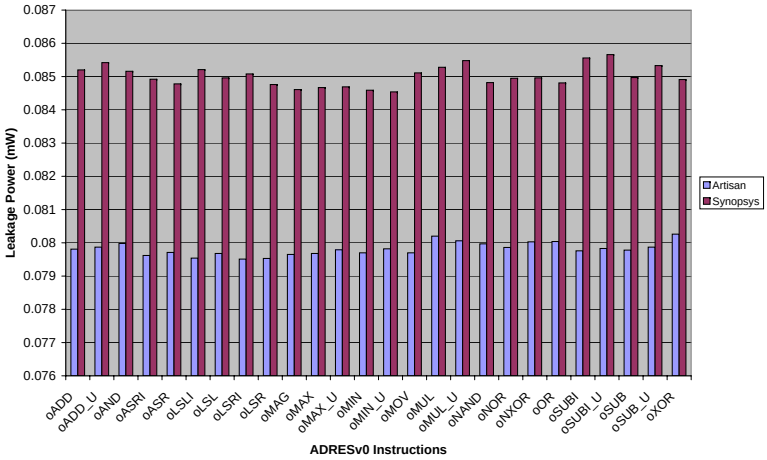
6.1.3 Functional Units

Without any optimizations the functional units consume about 22 - 27% of the total power as depicted in Figure 5.2(a). When the register files are clock gated reducing their power consumption, this percentage will only increase. To counter act this effect operand isolation as described in Section 5.2.1 is implemented in the pipelined functional unit of ADRESv1. The selection of the library utilized for the functional units is based on empirical results of the non-pipelined version at a frequency of 250MHz. The optimized, pipelined version was not available at the moment of selection, but became available later in the project. The pipelined FU is synthesized at 250MHz and compared with the non-pipelined FU.

The results of the non-pipelined FU's instructions for total and leakage power are depicted in Figures 6.7(a) and 6.7(b), respectively. Both libraries have similar values for total power except for the multiplication and (N)XOR operations, which are almost double compared to Synopsys. On the other hand, Artisan has less leakage compared to Synsopsys. Considering Figure 6.8 depicts the glitch power of the ALU in the CGA FU. The glitch values of Synopsys are completely zero, which is not expected. The reason could be the same as with the register files: incomplete Synopsys libraries. Therefore, the Artisan libraries are selected for reliability reasons, but the Synsopsys libraries are selected for maximum performance in ADRESv1.



(a) Total Power



(b) Leakage Power

Figure 6.7: Power Results of Instructions for non-pipelined Functional Unit in CGA Section

Operand isolation is only implemented in ADRESv1, since this feature reduces power significantly in a multi-cycle design as with pipelining. The effect is less with ADRESv0 as noted in Section 5.2.1, but also it was more convenient to implement in the optimized ALU. Figures 6.9(a) and 6.9(b) show the total and leakage power consumptions of the functional unit with and without operand isolation.

Additionally, notice the large power reduction between Figures 6.7(a) and 6.9(a), which can be as high as 60%. Operand isolation reduces power consumption of individual instructions and leakage power. As noted in Section 5.1 true leakage occurs when gate voltages are just below threshold values. Since OR gate implementation is utilized the gate voltages are always above the threshold values reducing leakage by 3% as depicted in Figure 6.9(b). Total power is slightly reduced for the individual instructions, but significant advantages require multiple cycles. Empirical tests with sequential execution

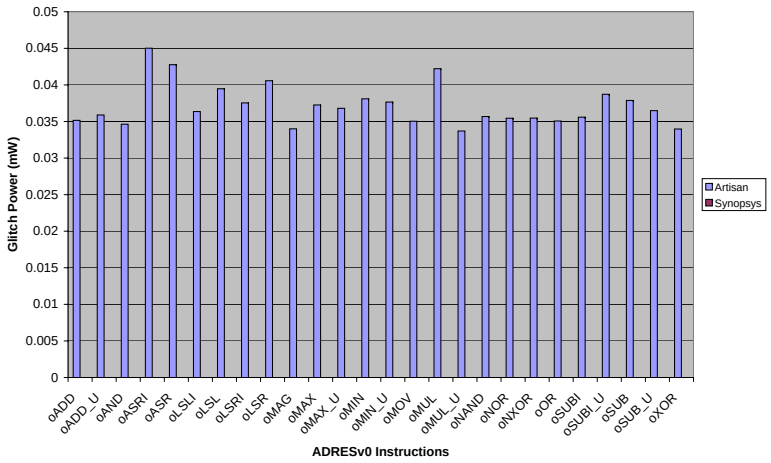


Figure 6.8: Glitch Power of a non-pipelined ALU in the CGA FU

of several CGA instructions showed a decrease of power in the FUs of about 30% with just an area increase of 6% as noted in Table 6.1, which is quite promising.

Table 6.1: CGA FU Area Results between with and without Operand Isolation with Synopsys Libraries

	Without OI	With OI
FU_CGA	1296.89	1224.22
ALU	35985.55	38537.60
Miscellaneous	8713.45	8959.86
Total	45995.89	48721.68

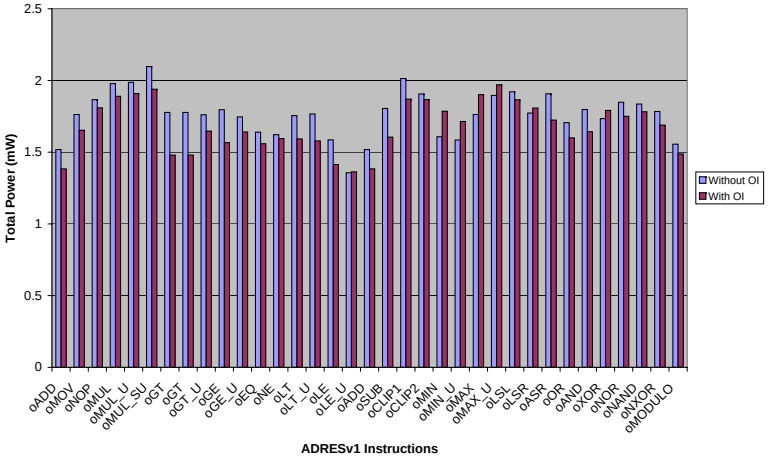
Synthesis results showed a timing penalty of 60 - 70 pico seconds in the FU due to the additional logic for operand isolation. The overall power reduction with operand isolation will be noted in Section 6.4.

6.2 Verification Environment

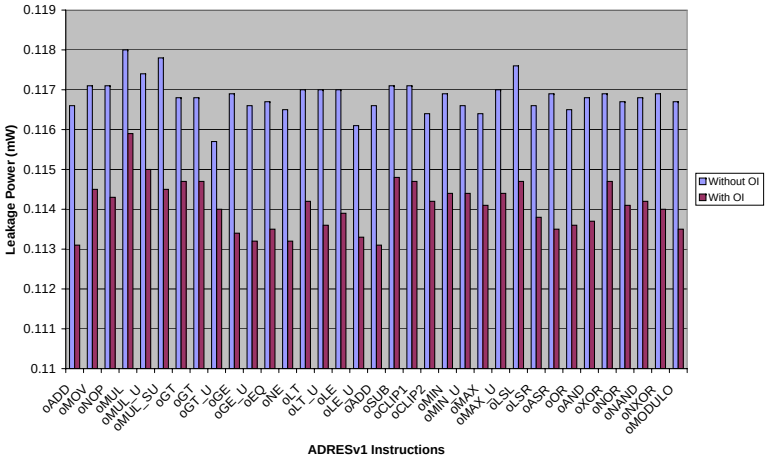
The optimization techniques explained in Chapter 5 had to be validated on functional operating architectures. The two architecture reference versions of ADRES are described that were utilized for verification of the optimization techniques. Additionally, the benchmark applications are described briefly as well.

6.2.1 Reference Architectures

ADRESv0 had an array size of 3x4 and was the first demo architecture for a proof-of-concept as depicted in Figure 6.2.1. This single cycle architecture consisted out of 3 rows and 4 columns of which FU_2 was a load unit and FU_3 a store unit, while FU_0 controlled the array with the CGA command and was able to perform branches. The



(a) Total Power



(b) Leakage Power

Figure 6.9: Power Results of Instructions for pipelined CGA FU

performance at 100MHz was not significantly high due to the lack LD/ST units, but sufficed to execute applications like MPEG2 and IDCT.

The next version ADRESv1 was the first pipelined version with multiple LD/ST units in an array of 4x4 elements. The architecture is similar to the one in Figure 2.2(a), however, the architecture was constructed differently for debugging reasons. The VLIW and CGA views where now completely separated from each other, however, connections still existed between the VLIW and CGA views. Figure 6.2.1 depicts how this version was constructed, basically. It has 3 VLIW FUs in the VLIW section of which two have LD/ST capabilities. VLIW FU_0 is capable of control instructions and branches. In the array itself there are four LD/ST units and connected to the DMEM. This increased bandwidth significantly compared to ADRESv0 and was tested with the MPEG2 benchmar. The array does not contain predicate registers, but predicate busses instead. Additionally, all FUs have connections to the VLIW DRF through registers. This array is utilized for

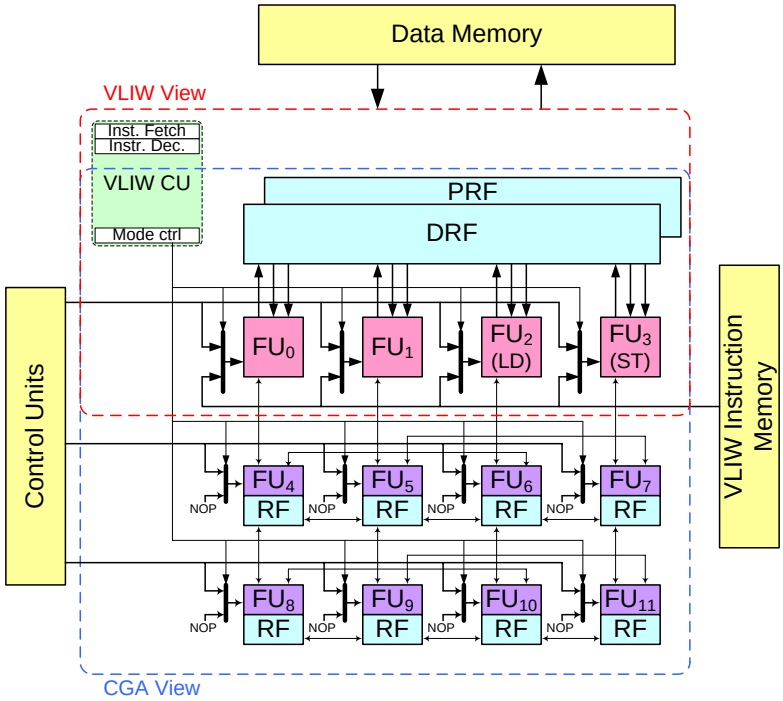


Figure 6.10: ADRESv0 version for proof-of-concept

validation of clock gating in Section 6.3.

During the coarse of the project the ADRESv1 was modified by registering the outputs of the local DRFs. The ALU in the FUs of this instance was also improved by rewriting the entire VHDL code. The performance was not improved by registering the DRFs output, but was compensated by the optimization techniques and the new ALU. This resulted in a synthesized architecture with a maximum frequency of 370MHz. Power estimation, however, were determined at 500MHz. This was originally the target frequency and requested by IMEC DESICS department. This 'pipelined CGA' architecture is utilized for validation of operand isolation.

6.2.2 Benchmark Applications

The applications used for verification are Fast Fourier Transformation [34], Inverse Discrete Cosine Transformation [5] and MPEG2 [21]. Detailed explanation of the code is beyond the scope of this thesis, but the algorithm documentation provide extensive information.

The FFT application is based on radix-4 operating on 1024 samples. IDCT operates on 396 macro blocks with a size of 8x8, which is utilized in the MPEG2 algorithm. The MPEG2 algorithm itself starts by decoding an I-frame followed by either a B or P-frame with a resolution of 176x144 pixels. The I-frame has more IDCT operations and less motion compensation, while the second frame has less IDCT and more motion compensation [21]. In the simulations only the first two frames are decoded to obtain

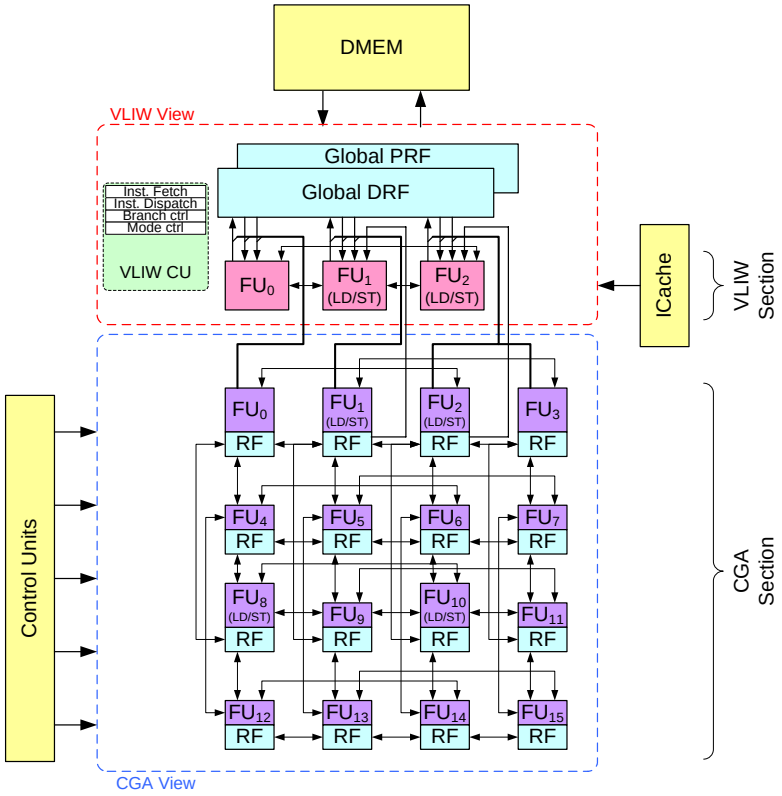


Figure 6.11: ADRESv1 version

switching activity, however, more frames should be decoded to get more reliable power figures. To validate the expected outcome of the power results we compared two frames with each other in terms of power. The power figures of the two frames are from the ADRESv1 version.

Table 6.2: MPEG2 Frame Power Results

	Cycles	Power (mW)
Frame 1	2659169	76.04
Frame 2	3624528	78.38

The results have an offset of 3% showing that the power consumption is more or less reliable for MPEG2. Decoding the I-frame take about 2.6 million cycles compared to the 1 million cycles for the second frame. Although the power consumptions are similar, energy consumption for the first frame is twice as high as that of the second frame. For even more accurate power figures, more frames should be simulated.

6.3 Clock Gating Results

Clock gating is implemented in ADRESv0 and v1 during synthesis by the Synopsys tools as depicted in the synthesis flow of Figure 4.2. The IDCT application is simulated on both architectures, while MPEG2 is only simulated on ADRESv1. Tables 6.3 and 6.4 depict the results for ADRESv0 and ADRESv1, respectively. The FFT results are not provided, since the benchmark did not work properly on the reference architectures. Reliable results are required for validation of clock gating and are therefore omitted. The clock gating results are compared to the not-optimized versions. Note that the amount of IDCT data with ADRESv1 simulation is larger than ADRESv0 enlarging energy consumptions, relatively.

Table 6.3 show a total power reduction of about 6% for ADRESv0, while leakage power has increased by 4% due to the additional logic. With the same MIPS and frequency values the MIPS/mW and nW/MHz improved as well as with 6%.

Table 6.3: Clock Gating improvements for ADRESv0

ADRESv0	Total		Leakage		MIPS	MIPS/mW	mW/MHz
	Power (mW)	Energy (uJ)	Power (mW)	Energy (uJ)			
IDCT							
No opt.	59.16	2.80	2.29	0.108	643	10.87	0.59
Clock Gating	55.66	2.63	2.38	0.112	643	11.55	0.55
Improvement	5.9%	5.9%	-4.0%	-4.0%	0%	5.9%	5.9%

The results of clock gating with ADRESv1 showed a significant reduction in power and energy of 14 - 21% for IDCT and MPEG2, respectively. The leakage power, showed an increase of 13.3%, which is due to the additional logic as with ADRESv0. Additionally, MIPS values remained the same and resulted in higher MIPS/mW and mW/MHz values. Comparing the MIPS values of ADRESv0 and ADRESv1 (although not quite fair as noted in Section 5.3.1) shows an increase of about 750%, which is significant. This is not only due to pipelining, but also because of more resources in the ADRESv1 reference architecture.

For correct arrival of the clock signal in a chip of the pipelined design, ADRESv1 has a clock tree implemented. Power consumption of the clock tree is about 6 - 15% of total power for IDCT and MPEG2, respectively. Table 6.4 shows that clock gating reduces power of the clock tree by 10%, which is due to the reduction of the load on the clock tree.

Comparing the Tables 6.3 and 6.3 show that clock gating has large advantages with a multi-cycle than with a single-cycle architecture. In addition, the timing reports did not show any significant timing penalties in the clock gated design.

6.4 Operand Isolation Results

Operand isolation is automatically inserted in ADRESv0 by the synthesis tools, while this is done manually in ADRESv1 as noted in Section 5.2.1. During the coarse of the

Table 6.4: Clock Gating improvements for ADRESv1

ADRESv1	Total		Leakage		MIPS	MIPS/mW	mW/MHz
	Power (mW)	Energy (uJ)	Power (mW)	Energy (uJ)			
IDCT							
No opt.	373	45.18	3.171	0.38	5458.71	14.63	0.75
Clock Gating	319.9	38.75	3.59	0.43	5458.71	17.06	0.64
Improvement	14.24%	14.24%	-13.21%	-13.21%	0%	16.6%	14.24%
MPEG2							
No opt.	166.1	1082.72	3.173	20.48	1066.27	6.42	0.33
Clock Gating	131.1	854.59	3.596	23.44	1066.27	8.13	0.26
Improvement	21.07%	21.07%	-13.33%	-13.33%	0%	26.6%	21.07%
Clock Tree							
No opt.	21.39		0.015				
Clock Gating	19.21		0.045				
Improvement	10.19%		-205.46%				

thesis clock gating was first implemented and then operand isolation together with clock gating. Therefore, the clock gated architectures in the previous section are utilized as reference.

Tables 6.5 and 6.6 note the results of operand isolation combined with clock gating for ADRESv0 and ADRESv1, respectively, compared to the architectures with only clock gating. For completeness, we also compare with the original architecture in the tables. All architecture are simulated at 500MHz for power estimations. Again, the IDCT application is simulated on both architectures, while MPEG2 is only simulated on ADRESv1. FFT did not work properly on the reference architectures.

Considering Table 6.5, operand isolation only has a positive effect when compared to the original architecture. When comparing with clock gating alone it does not have any positive effect for a non-pipelined architecture with automatic insertion of operand isolation. When comparing the figures with the original version in Table 6.3 there is an improvement of 1% only. The figures in Table 6.5 prove that automatic insertion of operand isolation should be avoided.

As noted in Section 6.2.1 the architecture of ADRESv1 was modified as well as its ALU. This enhanced version is utilized for implementation of operand isolation. The results noted in Table 6.6 show power improvements of 30 - 40% when compared to clock gating and even 40 - 52.8% when compared to the original architecture. Due to the additional registers in the array the performance (MIPS) reduced by 19 - 26%. The operand isolation and clock gating features reduced power by 30 - 40%. Combining the performance and power consumption results actually increased MIPS/mW by 4 - 35% when compared to the architecture with only clock gating. The modified architecture also has a positive effect on the mW/MHz by 29 - 51% reduction and the clock tree power reducing by 25 - 33%.

As expected, the leakage power is increased due to additional logic. Compared to the original architecture leakage has increased by 13.45%, however, clock gating and operand

Table 6.5: Operand Isolation + Clock Gating Improvements for ADRESv0

ADRESv0	Total		Leakage		MIPS	MIPS/mW	mW/MHz
	Power (mW)	Energy (uJ)	Power (mW)	Energy (uJ)			
IDCT							
No opt.	59.16	2.80	2.29	0.108	643	10.87	0.59
Clk Gating + OI	58.45	2.76	2.45	0.115	643	11	0.58
Improvement	1.2%	1.2%	-7.0%	-6.5%	0%	1.2%	1.2%
IDCT							
Clock Gating	55.66	2.63	2.38	0.112	643	11.55	0.55
Clk Gating + OI	58.45	2.76	2.45	0.115	643	11	0.58
Improvement	-5%	-5%	-2.9%	-2.9%	0%	-5%	-5%

isolation target reduction of switching power and not leakage power.

Table 6.6: Operand Isolation + Clock Gating Improvements for ADRESv1

ADRESv1	Total		Leakage		MIPS	MIPS/mW	mW/MHz
	Power (mW)	Energy (uJ)	Power (mW)	Energy (uJ)			
IDCT							
No opt.	373	45.18	3.171	0.38	5458.71	14.63	0.75
Clk Gating + OI	226.2	35.86	3.591	0.57	4009.99	17.72	0.45
Improvement	39.36%	20.6%	-13.25%	-50.00%	-26.54%	21.12%	40%
IDCT							
Clock Gating	319.9	38.75	3.59	0.43	5458.71	17.06	0.64
Clk Gating + OI	226.2	35.86	3.591	0.57	4009.99	17.72	0.45
Improvement	29.29%	7.45%	-0.03%	-30.92%	-26.54%	3.87%	29.7%
MPEG2							
No opt.	166.1	1082.72	3.173	20.48	1066.27	6.42	0.33
Clk Gating + OI	78.38	568.18	3.6	26.10	860.51	10.98	0.16
Improvement	52.81%	47.52%	-13.46%	-27.44%	-19.30%	71.03%	51.52%
MPEG2							
Clock Gating	131.10	854.59	3.596	23.44	1066.27	8.13	0.26
Clk Gating + OI	78.38	568.18	3.600	26.10	860.51	10.98	0.16
Improvement	40.21%	38.06%	-0.11%	-11.35%	-19.30%	35.06%	38.46%
Clock Tree							
No opt.	21.39		0.015				
Clk Gating + OI	14.34		0.01023				
Improvement	32.96%		-31.80%				
Clock Tree							
Clock Gating	19.21		0.045				
Clk Gating + OI	14.34		0.01023				
Improvement	25.35%		77.31%				

The combination of clock gating and operand isolation proved to enhance power results by 39 - 53% for a pipelined design compared to the original architecture of ADRESv1. As noted earlier power estimations are performed at 500MHz. However, the actual frequency obtained after synthesis decreased from 370MHz (2.70nsec) to 340MHz (2,94nsec) due to the added blocking logic in the data path.

6.5 Milestone Architectures

The ADRES instances selected during the architectural explorations and modifications are considered as milestones of which the results are depicted in this section. The first architecture is the base architecture *4x4_reg_con_all* selected in Section 2.5 with an array size of 4x4. Next, is *4x4_reg_con_shared_morhposys_64G_16L* in Section 5.3.2 with the shared register files among the functional units (*4x4_reg_con_shared*), the Morphosys option (*morphosys*) as depicted in Figure 2.14 and with 64 registers for the global DRF (*64G*) and 16 for the local DRFs (*16L*). In the same section the number of registers of the local RFs are reduced from 16 to 4 of which the results are noted by *4x4_reg_con_shared_morhposys_64G_4L*. These non-pipelined architectures were compliant with DRESC2.0 of which switching activity for power calculations is obtained by the Esterel methodology as explained in Section 4.2.

The dynamic and static optimizations described in Chapter 5 are implemented in the lastly named architecture, compliant with DRESC2.4 and simulated with the ModelSim v6.0a simulator. For this final step three array dimensions are created: 2x2, 4x4 and 8x8. The architectures in the text are called *2x2_final*, *4x4_final*, *8x8_final* for ease of discussion and are all pipelined. The maximum clock frequencies for these architectures are 322, 312 and 294MHz, respectively. A total overview of the frequencies is noted in Table 6.7.

Table 6.7: Milestones Frequencies

Architecture	Frequency (MHz)
4x4_reg_con_all	100
4x4_reg_con_shared_morhposys_64G_16L	100
4x4_reg_con_shared_morhposys_64G_4L	100
4x4_reg_con_shared_morhposys_64G_4L_final	312
2x2_reg_con_shared_morhposys_64G_4L_final	322
8x8_reg_con_shared_morhposys_64G_4L_final	294

Memory segmentation did not provide improvements for memory sizes less than 256 words as already noted in Section 5.3.3. The 2x2 array, however, has less resources for scheduling increasing the initiation interval (II) to 223 configuration lines for MPEG2 justifying segmentation.

6.5.1 Results

The area results of the architectures are depicted in Figure 6.5.1. Starting from 4x4_reg_con_all the areas of the local DRFS are reduced and are hardly noticeable in

the 4x4_reg_con_shared_morhposys_64G_4L architecture. The areas of the FUs in the final architectures are larger than those of the non-pipelined versions. This was expected, since the improved FU in Table 6.1 showed an increase of area due to pipelining and operand isolation. The 8x8 architecture requires a large amount of area for the configuration memories and CGA FUs.

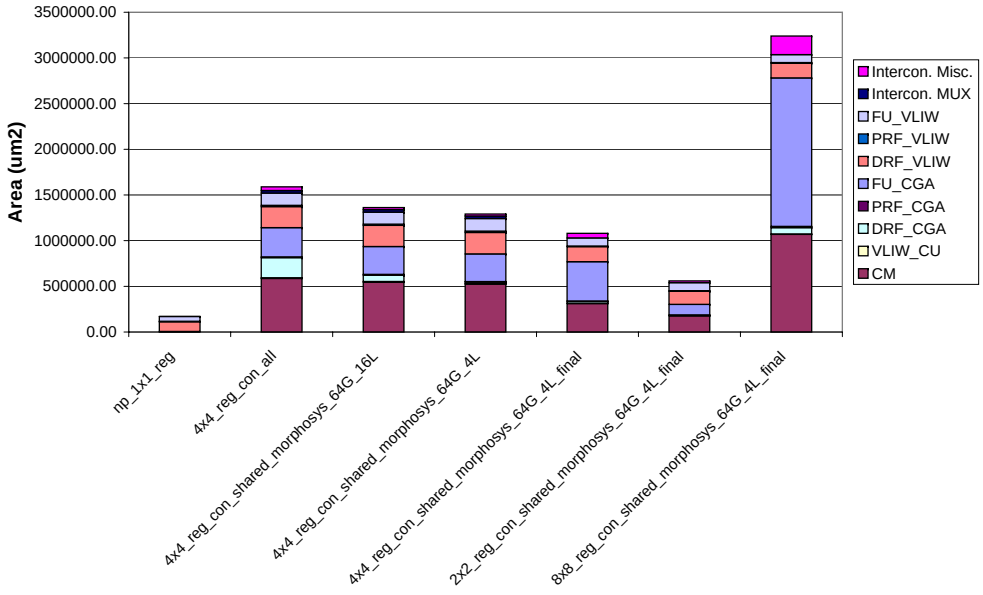


Figure 6.12: Milestones Area

Comparing the area results with leakage power as depicted in Figure 6.13 we see there is consistency between the two charts, except for 4x4_reg_con_shared_morhposys_64G_4L. Normally, leakage power is equivalent to area as Figures 2.15 and 2.16 depict. This is caused by bad results of the synthesis tool. The pipelined designs show an increase of leakage power in the FUs due to operand isolation logic compared to the non-pipelined designs. The np_1x1_reg requires less area and less leakage, however, 2x2_final is also a good choice.

In this section the IDCT results are utilized for discussion, while the results of the FFT and MPEG2 benchmarks can be found in Appendix G.

The power results in Figure 6.14 show a gradual decrease in power for the first three 4x4 architectures after which power increases due to the higher clock frequencies. Power consumed by the local DRFs is reduced due to the small register file sizes. The Global DRFs is not reduced in size, however, their power is reduced by the clock gating feature.

The power results of the configuration memories in architectures 4x4_reg_con_shared_morhposys_64G_4L and 4x4_final appear to have the same values despite the increase of frequency. The difference between the two architectures is caused by the different ADRES versions. With ADRESv0 the configuration memories of 4x4_reg_con_shared_morhposys_64G_4L are in total 1144 bits wide, while those of the 4x4_final (ADRESv1) are 729 bits wide, while the top-level architecture files are basically the same. In addition to this, there are more CMs in the first architecture

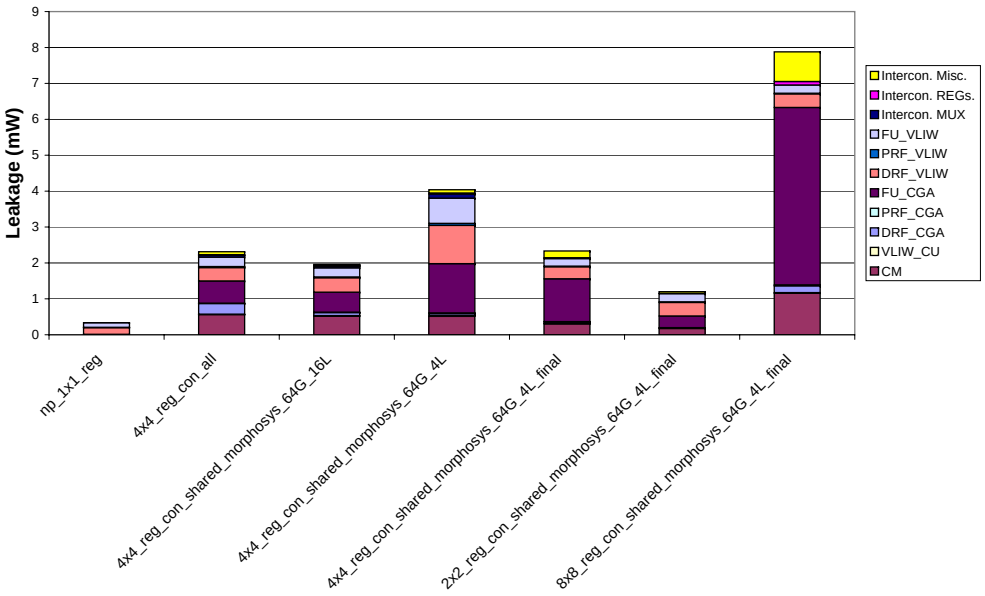


Figure 6.13: Milestone IDCT Leakage

increasing area and power for decoding logic. Figure 6.5.1 depicts the decrease of area for the CMs between the two architectures. Due to the smaller sizes the CMs consume less power, however, by increasing the frequency from 100MHz to 312MHz, as with the 4x4_final architecture, the power values are increased to a value similar to that of the CMs of 4x4_reg_con_shared_morhposys_64G_4L.

Interesting to note is the power consumption in the CGA FUs, which is higher than expected. For example, the FUs in 4x4_reg_con_shared_morhposys_64G_4L consumes about 11.7mW. When multiplying this with 312/100 = 3.12 the expected power consumption is 37.44mW. However, this is 41.8mW in the final 4x4 architecture. Sections 6.3 and 6.4 proved that clock gating and operand isolation reduce power, however, the problem is caused by the benchmark simulations themselves. After the applications went into CGA mode, they did not return to VLIW mode producing more switching activity than should, since in VLIW mode the CGA is idle. Although output data was corrupted, the activity results were utilized to calculate power in a worst case like scenario of operation.

The energy results in Figure 6.15 decrease with every milestone except for the 8x8_final architecture. This is due to the high power consumption and relatively long simulation times. The 4x4_final architecture is almost the same to the 2x2_final architecture. The 4x4_final architecture consumes more power than the 2x2_final, but requires less cycles for IDCT. The 2x2_final architecture has less energy consumption, however, the 4x4_final architecture is preferred, since it has higher throughput with the same amount of energy.

The performance results in Figure 6.16 show that the highest MIPS with a relatively high MIPS/mW is accounted for the 4x4_final architecture. Although, the 2x2_final has low MIPS results the MIPS/mW is similar to the 4x4_final architecture. The low MIPS

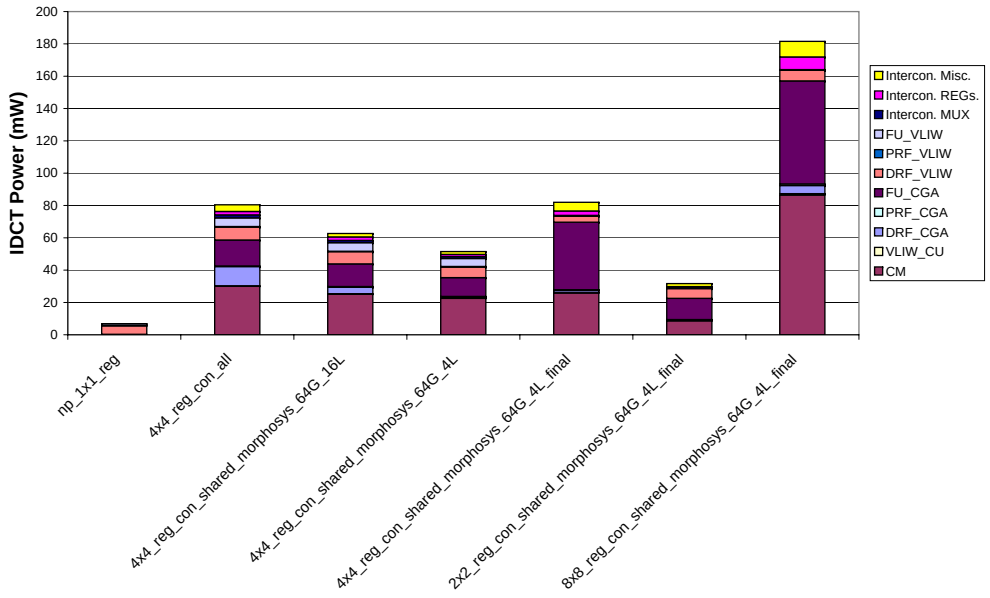


Figure 6.14: Milestone IDCT Power

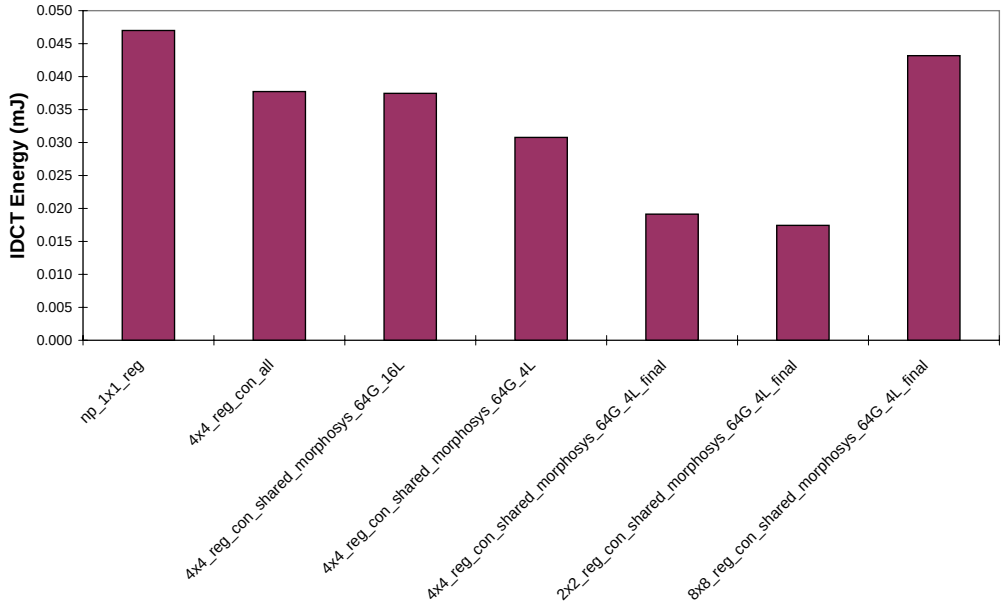


Figure 6.15: Milestone IDCT Energy

of the 2x2_final is due to the lack of routing resources increasing II and number of cycles to execute an iteration in the loop. The MIPS results influence the um^2/MIPS where the 4x4 and 2x2 final architectures are as well most optimal as depicted in Figure 6.17.

Since the 2x2_final architecture has the lowest power consumption with a high frequency at 322MHz, the mW/MHz is also low as depicted in Figure 6.18. This effect

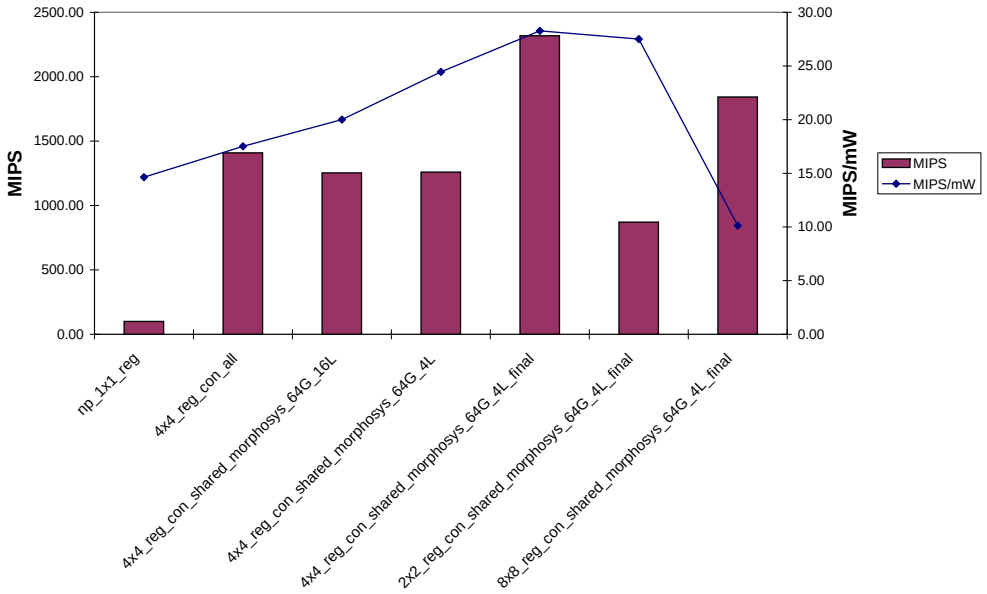


Figure 6.16: Milestone IDCT Performance

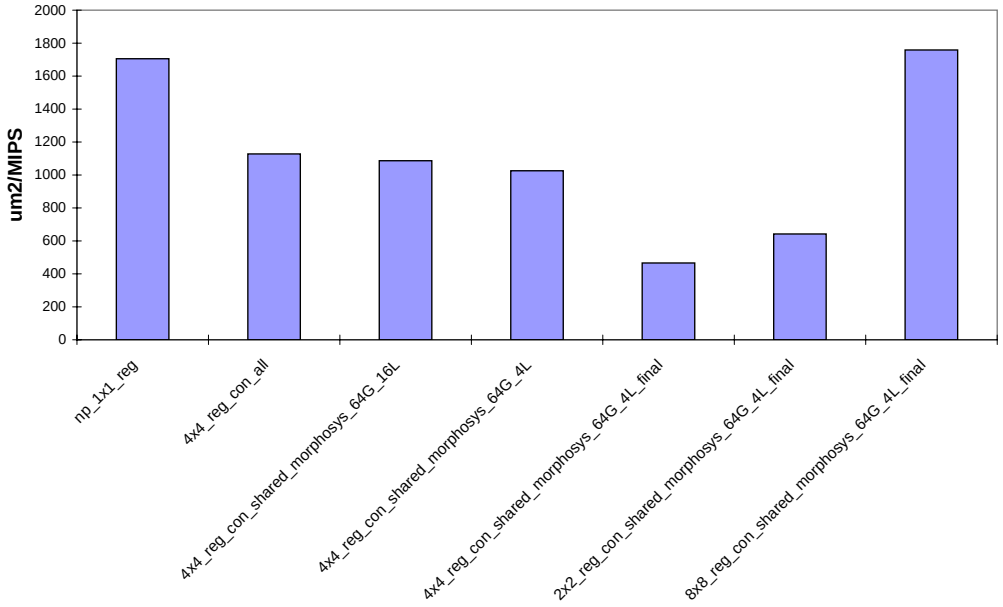


Figure 6.17: Milestone IDCT Area vs. Performance

applies to every benchmark and is even as low as that of the single functional unit np_1x1_reg.

Combining the energy charts and simulated time (Total time for simulating benchmark) for the applications creates the energy-delay pareto points of the architectures as depicted in Figure 6.19. The np_1x1_reg architecture is omitted, since the delay was too

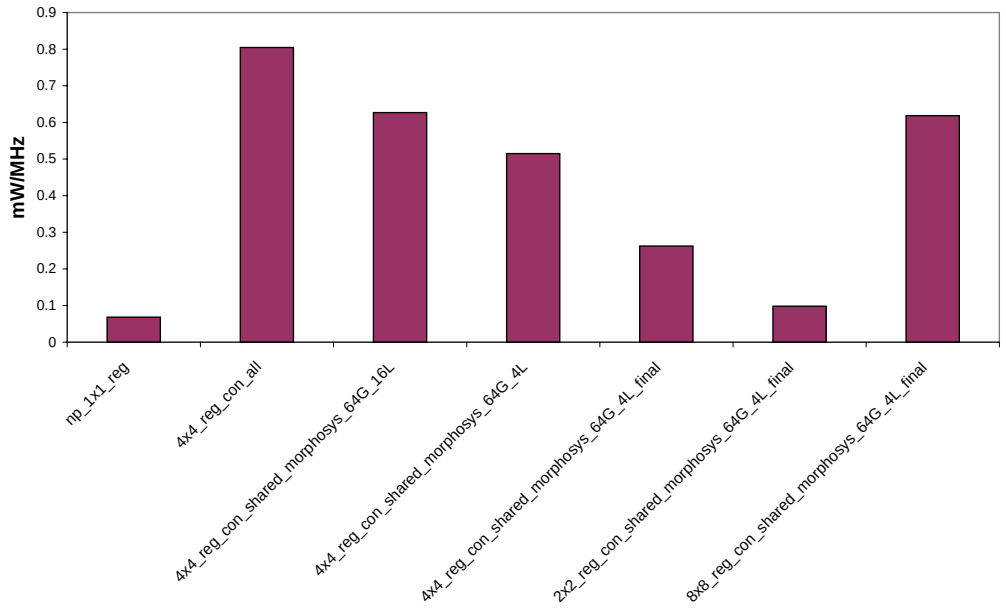
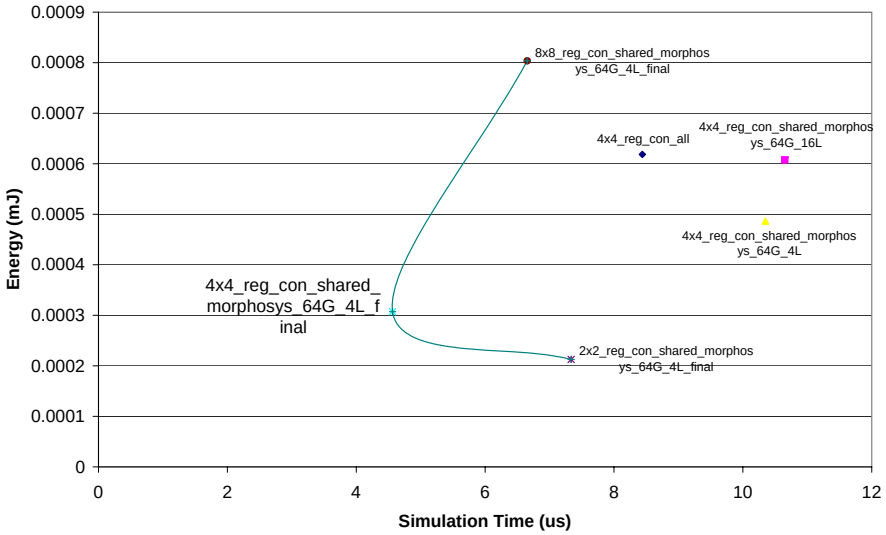


Figure 6.18: Milestone IDCT Power vs. Frequency

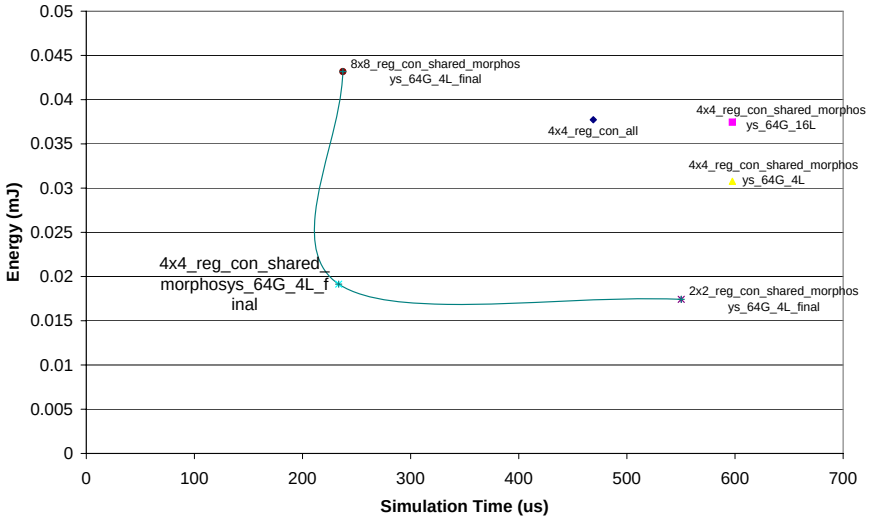
large to display it properly on the charts. The energy-delay charts show the scalability of ADRESv1 for the 2x2, 4x4 and 8x8 architecture and their impact on energy and delay. The 8x8 architecture requires most energy consumption as was expected, however, in the MPEG2 simulation the simulation time was even longer than that of the 2x2 simulation. It appeared that the scheduler failed to map a loop of IDCT onto the array, which reduced performance considerably. The same thing happened with FFT where a loop could not be mapped properly resulting in longer simulation time. Therefore, it is important to map an application as much as possible on the array.

Combining the results and especially looking at the pareto points the *4x4_reg_con_shared_morphosys_64G_4L_final* architecture is selected as most optimal and final architecture based on power, performance and energy-delay. Table 6.8 depicts the absolute figures of the base architecture *4x4_reg_con_all* with the proposed architecture.

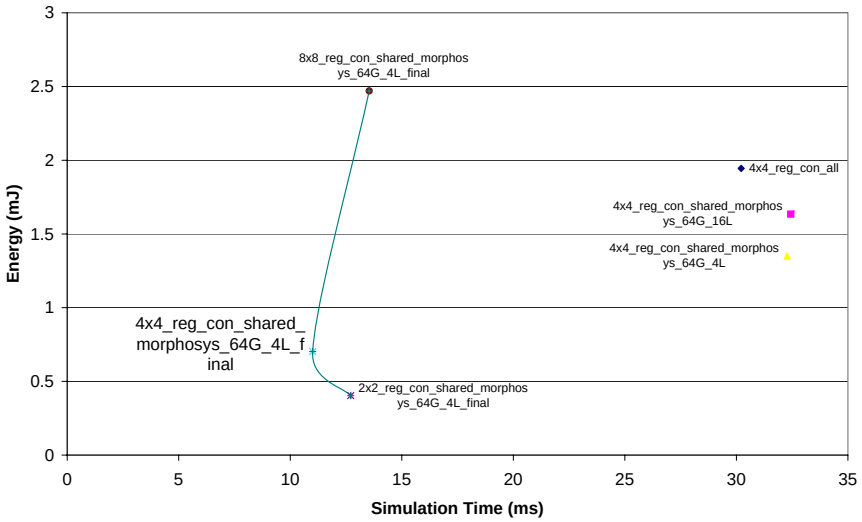
The results in Table 6.8 show a moderate improvement in power, but also a increase in power with IDCT. However, total energy results are more important as noted in Section 2.5.2 ranging from 49 - 64%. The same applies to leakage where no improvements are for leakage power, but energy is improved significantly due to high increase of performance results. The area was improved from 1.59mm^2 (544k gates) to 1.08mm^2 (370k gates), which is equal to 32%. In overall, the base *4x4_reg_con_all* is improved significantly in energy, performance and area compared to the proposed architecture *4x4_reg_con_shared_morphosys_64G_4L_final*.



(a) Milestones FFT



(b) Milestones IDCT



(c) Milestones MPEG2

Figure 6.19: Milestones Energy-Delay

Table 6.8: Comparing base with final architecture

ADRESv1	Total		Leakage		MIPS	MIPS/mW	mW/MHz	Freq.
	Power (mW)	Energy (uJ)	Power (mW)	Energy (uJ)				(MHz)
FFT								
Base	73.28	0.619	2.307	1.947E-02	759	10.35	0.7328	100
Final	67.29	0.307	2.337	1.066E-02	1190	17.68	0.2153	312
Improvement	8.17%	50.4%	-1.3%	45.25%	56.78%	70.82%	70.62%	212%
IDCT								
Base	80.45	37.72	2.312	1.084	1409	17.51	0.8045	100
Final	81.99	19.14	2.333	0.545	2318	28.27	0.2624	312
Improvement	-1.91%	49.25%	-0.91%	49.72%	64.51%	61.45%	67.38%	212%
MPEG2								
Base	64.35	1944	2.297	69.41	338	5.25	0.6435	100
Final	63.87	702.7	2.341	25.76	674	10.56	0.2043	312
Improvement	0.75%	63.85%	-1.92%	62.89%	99.4%	101.14%	68.25%	212%

6.6 Total ADRES Processor Power

The power measurements of ADRES are based on the core architecture only, but for a total overview the entire processor should be looked at. Therefore, application notes of Artisan SRAMs are utilized to predict the power consumption of the memories in the ADRES processor with a 4x4 core.

The four memories connected to the data memory interface are 32bx2048w SRAMs with a typical consumption of 0.03mW/MHz. The instruction cache has 4 banks of 32bx512w SRAMs with 0.022mW/MHz making it 0.088mW/MHz for all. There are 3 VLIW FUs in any ADRESv1 instance meaning only 96 bits are required of each bank. With a maximum of two banks active the power consumption becomes: $2 \times 3/4 \times$ 0.022mW/MHz = 0.033mW/MHz. The TAG memories are 17bx512w SRAMs with 2×0.014mW/MHz = 0.028mW/MHz, since both are always active. Combining the TAG and data memories results in 0.061mW/MHz for the ICache. According to Artisan data sheets the power consumptions of the memories are negligible when they are not used.

Table 6.9 depicts the theoretical total powers for the verification programs where α_{dmem} is the percentage of LD/ST instructions of total amount of cycles. The ICache is only accessed in VLIW mode of which the average activity is noted by α_{icache}. Figure 6.20 depicts the distribution of all the components in the processor for the IDCT benchmark.

Table 6.9: ADRES processor *4x4_reg_con_shared_morphosys_64G_4L* at 312MHz

App.	P_{core} (mW)	α_{dmem}	P_{dmem} (mW)	α_{icache}	P_{icache} (mW)	P_{tot} (mW)	E_{tot} (mJ)	mW/MHz Total	MIPS/mW Total
FFT	67.29	0.61	16.85	0.520	9.980	94.12	$0.429 \cdot 10^{-3}$	0.30	12.64
IDCT	81.99	1.00	27.46	0.043	0.936	110.39	25.77	0.35	21.00
MPEG2	63.87	0.60	16.54	0.750	14.35	94.76	1042.53	0.30	7.11

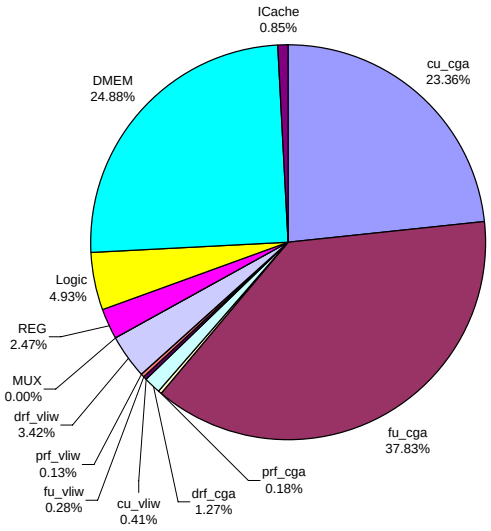


Figure 6.20: Overview of ADRES Processor Power Consumption with IDCT

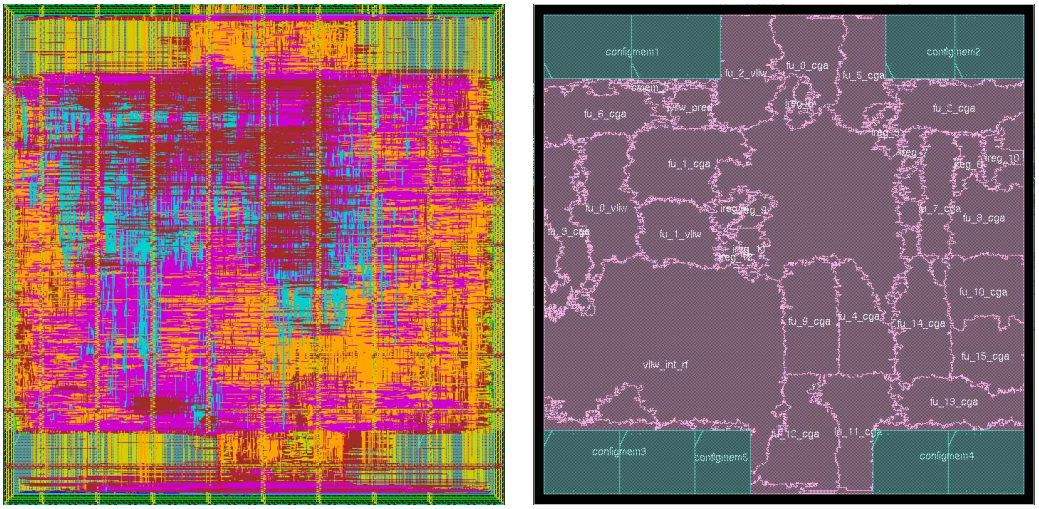
The chart in Figure 6.20 shows that the CGA FUs, data memory and configuration memories (cu_cga) consume most of the total power. The FUs are already optimized with operand isolation and segmenting the configuration memories does not have any positive effect with this configuration. The Data Memory Interface is the only component that could be optimized by memory segmentation, which could be done in future developments. The MUX value in the chart is set to 0.00%. ADRES utilizes a lot of multiplexors, however, higher frequencies can be obtained during synthesis when the utilization of these is disabled. The Synopsys Design and Physical Compiler optimizes the architecture better without multiplexors and uses standard logic components instead.

Although these calculation are theoretical, compared to the performance of the Trimedia processor TM3270 with 0.649mW/MHz [43] the improvement is 46 - 53%, which is quite considerable indicating the power efficiency of ADRES. A complete gate level simulation with DMEM, ICache and ADRES core should be performed to validate these assumptions.

The final 4x4_reg_con_shared_morphosys_64G_4L architecture is synthesized with Synsopsys Design Compiler and place&routed by Encounter [15]. The verilog netlist and SPEF file (capacitance and resistor values) are utilized for the power calculations as with the 2x2 and 8x8 architectures. The architecture after place&route is depicted in Figure 6.21(a) of which the *amoeba* view is depicted in Figure 6.21(b). The configuration memories are placed on the side of the area, since this is better for timing. The rest of the area is filled with the FUs, DRFs and PRFs.

6.7 Comparison with LISATeK VLIW Architecture

Schuster et al. [39] created a scalable sub-word-parallelism-enabled VLIW architecture targeting 100Mbps Software Defined Radios [9] (SDR) created with LISATEK [16]. A 64 points FFT radix-4 kernel is utilized for verification. The number of FUs varied from



(a) Physical View (b) Amoeba View

Figure 6.21: 4x4_reg_con_shared_morphosys_64G_4L ADRES Core Layout Views

2, 4, 6 to 8 to obtain a energy-delay chart similar to Figure 6.19(a). The architecture with 4 FUs, intrinsics, pipelining and modulo scheduling has its energy and performance figures depicted in Table 6.10 based on the Synopsys TSMC library *tcbn90ghpnvt*.

Table 6.10: FFT Figures of VLIW instance from [39] and final 4x4 ADRES instance

	4FUs VLIW	4x4 ADRES
Energy	167.82nJ	307.10nJ
Simulation Time	1.64usec	4.6usec
mW/MHz	0.28	0.22
Frequency	360MHz	312MHz
FFT Points	64	1024

The 4x4 ADRES architecture consumes more energy, but it also operates on more FFT points. The number of FFT points is equal to 4^V where V is the number of steps required to process the values. Therefore, the 64 FFT points require 3 steps, while 1024 points require 5 steps. Modifying the results of the 1024 points to 64 points gives the following outcome: 184.26nJ and 2.76usec. The VLIW has better results compared to ADRES, however, the VLIW does not have configuration memories as ADRES does. The FFT benchmark also had a low IPC of 5.58 on the ADRES architecture, which is relatively low compared to the entire array of 16 FUs.

6.8 Summary

Based on the configuration memories, functional units and data register files the appropriate TSMC library is selected. The Artisan generated macro RFs are utilized for the configuration memories. Artisan TSMC standard cell libraries are used during the

architectural explorations based on ADRESv0 due to reliability issues with power, while Synopsys TSMC standard cell libraries are used for ADRESv1 for maximum performance.

The dynamic optimizations in this thesis are verified using two reference versions of ADRES: v0 and v1 with FFT, IDCT and MPEG2 simulations. ADRESv0 was a non-pipelined 3x4 architecture with a single load and store unit, while ADRESv1 was a pipelined 4x4 architecture with multiple LD/ST units, hence better performance.

The clock gating and operand isolation optimizations were applied to the architectures and verified at RT level with ModelSimv6.0a. Clock gating was applied with the synthesis tools in both versions. With ADRESv0 power was reduced by 6% compared to 14 - 21% for ADRESv1 with clock gating. Operand isolation was automatically applied in ADRESv0 and manually in ADRESv1. Combining automatic implementation of operand isolation with clock gating in ADRESv0 increased power by 5% compared to the architecture with only clock gating. Therefore, automatic implementation of operand isolation should not be utilized. Manual implementation in the pipelined design ADRESv1 reduced power by 30 - 40%.

The optimizations are merged in the architecture with shared register files as selected in Section 5.3.2. The energy-delay chart depicted that the pipelined 4x4 architecture (*4x4_reg_con_shared_morphosys_64G_4L*) is most optimal for the targeted benchmarks. The architecture consumes 64 - 82mW at 312MHz with an operating voltage of 1V (Synopsys TSMC library *tcbn90ghpnvt*). The area is 1.08mm^2 for the core which is equivalent to 370k gates. Compared to the base architecture, power is improved by 8%, energy by 49 - 65%, MIPS/mW by 61 - 70% at a frequency of 312MHz.

7 Conclusions and Future Work

7.1 Conclusions

The purpose of this thesis was to evaluate the performance and power consumptions of a basis ADRES architecture using a set of benchmark applications and optimize the architecture balancing power and performance. A synthesis, simulation and power calculation flow has been created to obtain power and performance results of a variety of benchmarks: FFT, IDCT and MPEG2. For the synthesis flow we utilized Synopsys synthesis tools and 90nm TMSC libraries. A simulation flow is composed verifying the benchmarks and obtained switching activity of the nets at RT level. The regular methodology of capturing switching activity is with a RT/gate level VHDL simulator (ModelSim v6.0a). In addition to this a cycle true instruction set simulator Esterel was used to capture switching activity of the nets of the synthesis invariant components as well. The composed power flow annotated the switching activities of the nets onto the synthesized gate level design and calculated power with detailed information of the components' consumptions. The much faster Esterel simulation resulted in power estimations of the ADRES instances within 11% accuracy compared to RT level simulation with ModelSim.

A number of optimization techniques were applied and evaluated: operand isolation, clock gating, pipelining, architectural modifications and memory segmentation. *Operand isolation* reduced dynamic power by 30 - 40% with an absolute timing penalty of approximately 0.24nsec (8.8% of a 2.7nsec clock period) for a pipelined reference architecture. *Clock gating* reduced power by 14 - 21% without any noticeable timing penalty. *Pipelining* increased performance and power, but energy consumption was reduced. This feature could not be compared to a non-pipelined design, since the ALUs and FUs were completely redesigned. The *architecture modification* profited of sharing the local register files among four functional units. Compared to the base architecture by using this technique power consumption was reduced by 36% and energy by 18 - 21%. *Memory segmentation* only has a positive effect on power when the memories are larger than 128bx256w.

An architecture was proposed that incorporated shared registers among the FUs, clock gating, operand isolation and pipelining. Due to the higher clock frequency power increased, but energy was decreased. The architecture reduced energy consumption by 49.5 - 64% compared to the base architecture at a frequency of 312MHz.

7.2 Future Work

The power optimizations in this thesis are only some of the many available. A few were mentioned in the thesis and are noted below together with other possibilities.

- Prohibit read and write operations to a register file when not required. This reduces power significantly without any significant impact on timing.
- Utilize transparent pipelining to avoid unnecessary switching of the pipeline stages. With this technique the pipeline stages are transparent for data when in absence of hazards. For this the clock gating principle is utilized with an expected power reduction of 20 - 60%.
- Segmenting the instruction cache and data memories will result in significant power reductions, since these are accessed quite often. However, this will impact the maximum frequency of the ADRES processor possibly. The impact on performance should be looked at more closely when investigating this option.
- Using multiple libraries with different Vt (Multi-Vt) during synthesis is expected to have an impact on both power and performance. The critical data paths are synthesized with low Vt libraries increasing speed, while those off the non-critical path are synthesized with high Vt libraries lowering power. This could produce new critical paths, but Multi-Vt libraries are inevitable to meet performance and power goals, while minimizing leakage power.
- Apply power gating to reduce leakage power. Leakage, however, is only 3 - 4% of total power. Total power will not be decreased significantly with this technique

The proposed ADRES architectures had some problems with the benchmarks during VHDL RTL simulations. This problem should be resolved or power analysis would become impossible to justify properly. In addition, to validate the RT level simulations a working gate level simulation should be performed.

Bibliography

[1] N. Banerjee, A. Raychowdhury, S. Bhunia, H. Mahmoodi, and K. Roy, *Novel low-overhead operand isolation techniques for low-power datapath synthesis*, IEEE (2005), 206–211, Proceedings. 2005 International Conference on Computer Design, 2005.

[2] Grard Berry, *The esterel v5 language primer*, Centre de Mathmatiques Appliques, 2004 Route des Lucioles, v5_91 ed., June 2000.

[3] Ramon Canalt, Antonio Gonz5lezt, and James E. Smith, *Very low power pipelines using significance compression*, (2000), 181–190, International Symposium on Microarchitecture, 2000. MICRO-33. Proceedings. 33rd Annual IEEE/ACM.

[4] Benjamin Chen and Ivailo Nedelchev, *Power compiler: A gate-level power optimization and synthesis system*, IEEE (1997), 74–79, Proceedings., 1997, IEEE International Conference on Computer Design: VLSI in Computers and Processors, 1997. ICCD '97.

[5] Wen Hsiung Chen, C. Harrison Smith, and S. C. Fralick, *A fast computational algorithm for the discrete cosine transform*, (1977), 1004–1009, IEEE Transaction on Communications.

[6] Subash Chandar G, Mahesh Mehendale, and R. Govindarajan, *Area and power reduction of embedded dsp systems using instruction compression and re-configurable encoding*, (2001), 631–634, International Conference on Computer Aided Design, 2001. ICCAD 2001. IEEE/ACM.

[7] Serag GadelRab, David Bond, and David Reynolds, *Fight the power: Power reduction ideas for asic designers and tool providers*, SNUG (2005), Best of SNUG San Jose 2005 at SNUG Boston 2005.

[8] Patrick P. Gelsinger, *Microprocessors for the new millennium: Challenges, opportunities and new frontiers*, IEEE Int. Solid-State Circuits Conf. (ISSCC) Dig. Tech. Papers (Februari 2001), 22–25.

[9] John Glossner, Daniel Iancu, Jin Lu, Erdem Hokenek, and Mayan Moudgill, *A software-defined communications baseband design*, (Januari 2003), 120–128, IEEE Communications Magazine.

[10] John L. Hennessy and David A. Patterson, *Computer architecture: A quantitative approach*, third ed., ISBN: 1-55860-724-2.

[11] Willm Hinrichs, *Verlustleistungreduktion beim schaltungsentwurf auf register-transfer-ebene in architekturen der digitalen signalverarbeitung*, Ph.D. thesis, Universitt Hannover, 2004, ISBN: 3-18-337409-9.

[12] http://www-sop.inria.fr/esterel org/.

[13] http://www.arm.com/products/physicalip/productsservices.html.

[14] http://www.arm.com/products/solutions/AMBAHomePage.html.

[15] http://www.cadence.com/.

[16] http://www.coware.com/.

[17] The IMPACT Group. http://www.crhc.uiuc.edu/Impact/.

[18] http://www.mentor.com/.

[19] http://www.synopsys.com/.

[20] http://www.tsmc.com/.

[21] ISO/IEC, *Iso/iec 13818-2 itu-t h.262, mpeg2 manual*, ISO/IEC.

[22] Hans M. Jacobson, *Improved clock-gating through transparent pipelining*, ACM (2004), 26–31, Proceedings of the 2004 International Symposium on Low Power Electronics and Design, 2004. ISLPED '04.

[23] Ismail Kadayif and Mahmut T. Kandemir, *Instruction compression and encoding for low-power systems*, (2002), 301–305, 15th Annual IEEE International ASIC/SOC Conference, 2002.

[24] Zion Kwok and Steven J. E. Wilton, *Register file architecture optimization in a coarse-grained reconfigurable architecture*, (April 2005), 35–44, Proceedings of the 13th Annual IEEE Symposium on Field-Programmable Custom Computing Machines (FCCM'05) - Volume 00.

[25] Hyungwoo Lee, Hakgun Shin, and Juho Kim, *Glitch elimination by gate freezing, gate sizing and buffer insertion for low power optimization circuit*, IEEE (2004), 2126–2131, The 30th Annual Conference of the IEEE Industrial Electronics Society.

[26] Renu Mehra and Jan Rabaey, *Behavioral level power estimation and exploration*, Proc. First International Workshop on Low Power Design, University of California at Berkeley, April 1994.

[27] Bingfeng Mei, *A coarse-grained reconfigurable architecture template and its compilation techniques*, Ph.D. thesis, Katholieke Universiteit Leuven, Januari 2005, ISBN: 90-5682-578-X.

[28] Bingfeng Mei, Serge Vernalde, Diederik Verkest, Hugo De Man, and Rudy Lauwereins, *Adres: An architecture with tightly coupled vliw processor and coarse-grained reconfigurable matrix*, (2003), DATE 2004.

[29] ———, *Dresc: A retargetable compiler for coarse-grained reconfigurable architectures*, (December 2002), 166–173, Proceedings. 2002 IEEE International Conference on Field-Programmable Technology, 2002. (FPT).

[30] ———, *Exploiting loop-level parallelism on coarse-grained reconfigurable architectures using modulo scheduling*, **150** (September 2003), 575–581, Proc. Design, Automation and Test in Europe (DATE).

[31] Bingfeng Mei, Steven J.E. Wilton, and Noha Kafafi, *Interconnect architectures for modulo-scheduled coarse-grained reconfigurable arrays*, (2004), 33–40, Proceedings. 2004 IEEE International Conference on Field-Programmable Technology, 2004.

[32] Giovanni De Micheli, *Synthesis and optimization of digital circuits*, ISBN: 0-07-016333-2.

[33] M. Münch, B. Wurth, R. Mehra, J. Sproch, and N. Wehn, *Automating rt-level operand isolation to minimize power consumption in datapaths*, (2000), 624–631, Proceedings Design, Automation and Test in Europe Conference and Exhibition 2000.

[34] William H. Press, Brian P. Flannery, Saul A. Teukolsky, and William T. Vetterling, *Numerical recipes in c*, 1992, ISBN: 0-521-43108-5.

[35] Praveen Raghavan, Andy Lambrechts, and Murali Jayapala, *Empirical power/area/-timing models for register files*, (To appear in 2006).

[36] B. Ramakrishna Rau, *Iterative modulo scheduling: An algorithm for software pipelining loops*, (1994), 63–74, Proceedings of the 27th Annual International Symposium on Microarchitecture, 1994. MICRO-27.

[37] Mariagiovanna Sami, Donatella Sciuto, Cristina Silavano, Vittorio Zaccaria, and Roberto Zafalon, *Exploiting data forwarding to reduce the power budget of vliw embedded processors*, (2001), 252–257, IEEE transactions on Very Large Scale Integration (VLSI) Systems.

[38] ———, *Low-power data forwarding for vliw embedded architectures*, (October 2002), 614–622, IEEE transactions on Very Large Scale Integration (VLSI) Systems.

[39] T. Schuster, D. Novo Bruna, B. Bougard, V. Derudder, A. Hoffman, and L. Van der Perre, *Subword-parallel vliw architecture exploration for multimode software defined radio*, To appear in 2006.

[40] Hartej Singh, Ming-Hau Lee, Guangming Lu, Fadi J. Kurdahi, and Nader Bagherzadeh, *Morphosys: an integrated reconfigurable system for data-parallel and computation-intensive applications*, (May 2000), 465 – 481, IEEE Transactions on Computers.

[41] Synopsys, *Design compiler user guide*, Januari 2005, Release W-2004.12.

[42] ———, *Power compiler user guide*, Januari 2005, Release W-2004.12.

[43] Jan willem van de Waerdt, Stamatis Vassiliadis, Sanjeev Das, Sebastian Mirolo, Chris Yen, Bill Zhong, Carlos Basto, Jean-Paul van Itegem, Dinesh Amirtharaj,

Kulbhushan Kalra, Pedro Rodriquez, and Hans van Antwerpen, *The tm3270 media-processor*, (2005), Proceedings of the 38th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO'05).

[44] Javier Zalamea, Josep Llosa, Eduard Ayguad, and Mateo Valero, *Register constrained modulo scheduling*, (May 2004), 417–430, IEEE Transactions on Parallel and Distributed Systems.

XML Architectural File

A

Listing A.1: XML Architectural File

```
<!-- This part defines available resources -->
<resource>
    <!-- Functional Units -->
    <FU name = "fu_0">
      <in name = "src1" width = "32" /> <!-- Inputs -->
      <in name = "src2" width = "32" />
      <in name = "src3" width = "32" />
      <out name = "dst1" width = "32" /> <!-- Outputs -->
      <op>
        <opgroup name = "ldmem"/>
        <opgroup name = "stmem"/>
        <opgroup name = "arith"/>
        ...
        <!-- Operation Groups -->
      </op>
    </FU>
    <!-- Describe all FUs ... -->
    <FU name = "fu_15">
      <in name = "pred" width = "1" />
      <in name = "src1" width = "32" />
      <in name = "src2" width = "32" />
      <out name = "pred_dst1" width = "1" />
      <out name = "pred_dst2" width = "1" />
      <out name = "dst1" width = "32" />
      <op>
        <opgroup name = "arith"/>
        <opgroup name = "logic"/>
        <opgroup name = "shift"/>
        <opgroup name = "comp"/>
        <opgroup name = "pred"/>
        <opgroup name = "phi"/>
      </op>
      <param area = "0.01" conf_bits = "9" />
    </FU>
```

```
    <!-- Global Registers -->
    <RF name = "vliw_int_rf" width = "32" size = "64"
nonrotsize = "32">
       <in name = "in1"  we = "we1" />
        ...
       <in name = "in4"  we = "we4" />
       <out name = "out1" />
        ...
       <out name = "out8" />
    </RF>
    <RF name = "vliw_pred" width = "1" size = "64" nonrotsize = "32">
      <in name = "in1"  we = "we1" />
        ...
      <in name = "in4"  we = "we4" />
      <out name = "out1" />
        ...
      <out name = "out4" />
    </RF>
    <!-- Local Registers -->
    <RF name = "ireg_0" width = "32" size = "4">
      <in name = "in1"  we = "we1" />
      <out name = "out1" />
      <out name = "out2" />
    </RF>
    <RF name = "pred_4" width = "1" size = "4">
      <in name = "in1"  we = "we1" />
      <out name = "out1" />
    </RF>
    <!-- Constant values -->
    <CONST name = "const_0" width = "15" delay = "0" />
    ...
    <CONST name = "const_15" width = "15" delay = "0" />
    <!-- Transition Nodes -->
    <TRN type = "mux" name = "outireg1_0" width = "32" delay = "1" />
    <TRN type = "mux" name = "outpred1_0" width = "1" delay = "1" />
    <TRN type = "mux" name = "outpred2_0" width = "1" delay = "1" />
    ...
    <TRN type = "mux" name = "outireg1_15" width = "32" delay = "1" />
    <TRN type = "mux" name = "outpred1_15" width = "1" delay = "1" />
    <TRN type = "mux" name = "outpred2_15" width = "1" delay = "1" />
    ...
    <TRN type = "mux" name = "loop_start" width = "1" delay = "0" />
    <TRN type = "mux" name = "loop_stop" width = "1" delay = "0" />
</resource>
<!-- This part defines their connections -->
```

```
<connection>
    <!-- Connection 1 -->
    <connect>
        <src entity = "const_0" />
        <dst entity = "fu_0" port = "src1"/>
    </connect>
    <!-- Connection 2 -->
    <connect>
        <src entity = "fu_0" port = "dst1" />
        <dst entity = "ireg_0" port = "in1" />
    </connect>
    <!-- Connection 3 -->
    <connect>
        <src entity = "ireg_0"  port = "out1" />
        <dst entity = "fu_0" port = "src1" />
    </connect>
    <!-- Connection 4 -->
    <connect>
        <src entity = "ireg_0"  port = "out2" />
        <dst entity = "fu_0" port = "src2" />
    </connect>
    <!-- Connection 5 -->
    <connect>
        <src entity = "fu_0" port = "dst1" />
        <dst entity = "outireg1_0" />
    </connect>
    <!-- Connection 6 -->
    <connect>
        <src entity = "outireg1_0" />
        <dst entity = "fu_1" port = "src2" />
    </connect>
</connection>
<!-- This part defines the instruction set and behaviours -->
<behaviour>
    <!-- section to describe VLIW view of the architecture -->
    <vliw_section>
        <vliw_int_rfs>
            <vliw_int_rf name = "vliw_int_rf"/>
        </vliw_int_rfs>
        <vliw_pred_rfs>
            <vliw_pred_rf name = "vliw_pred" />
        </vliw_pred_rfs>
        <vliw_fus>
           <vliw_fu name = "fu_0"/>
           <vliw_fu name = "fu_1"/>
```

```
            <vliw_fu name = "fu_2"/>
        </vliw_fus>
        <!-- where to detect the stop signal of the loop -->
        <start_sig name = "loop_start" />
        <stop_sig name = "loop_stop" />
    </vliw_section>
    <!-- Intrinsic functions -->
    <intr_op>
        <op name = "intr00_gp__min" commutative = "true"/>
        <op name = "intr01_gp__min_u" commutative = "true"/>
        ...
        <op name = "intr18_gp__modulo" commutative = "false"/>
    </intr_op>
    <!-- Instruction Group -->
    <op_section>
        <opgroup name = "logic" delay = "1">
            <op name = "or" />
            <op name = "and" />
            <op name = "xor" />
            <op name = "nor" />
            <op name = "nand" />
            <op name = "nxor" />
        </opgroup>
        ...
        <opgroup name = "ldmem" delay = "4" queuedelay="6">
            <!-- load operations -->
            <op name = "ld_c" />
            <op name = "ld_c2" />
            <op name = "ld_i" />
            <op name = "ld_uc" />
            <op name = "ld_uc2" />
        </opgroup>
    </op_section>
</behaviour>
```

Instruction Set Architecture

B

Instruction Set

Instructions	ARITH_1 100	ARITH_2 101	ARITH_X1 110	ARITH_X2 111	LOGIC 001	PRED 011	LDST 010	SPECIAL (*) 000
0000	MOV	GT	SUBABS	CLIP1	LSL	PRED_GT	LD_UC	SETLO_0
0001	NOP	GT_U	INNERSUM	CLIP2		PRED_GT_U	LD_C	JMP_BR_0
0010	SHRMB	GE	AVGU4	MIN	LSR	PRED_GE	LD_UC2	SETHI_0
0011	RPHI (*)	GE_U	ADD2	MIN_U		PRED_GE_U	LD_C2H	MV2SR
0100	SHLMB	EQ	SUB2	MAX	ASR	PRED_EQ	LD_C2	SETLO_1
0101	PACK2		AVG_E	MAX_U			LD_I	JMPL_BRL_0
0110	PHI (*)	NE	SH_RND		OR	PRED_NE	LD_I2	SETHI_1
0111	SPACK2		ADD		AND			HALT
1000	MUL	LT	SUB		XOR	PRED_LT	ST_C	SETLO_2
1001	MUL_U	LT_U			NOR	PRED_LT_U	ST_C2	JMP_BR_1
1010	SPACKU4	LE			NAND	PRED_LE	ST_C2H	SETHI_2
1011	MUL_SU	LE_U			NXOR	PRED_LE_U	ST_C2R	MVFSR
1100						PRED_SET	ST_I	SETLO_3
1101						PRED_CLEAR	ST_I2	JMPL_BRL_1
1110								SETHI_3
1111					MODULO			CGA

(*) Not supported in CGA mode

Figure B.1: Operation Code Table

ENCO-DING	INS. GROUP	DETAILED ENCODING																															
		31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
LD_{RR}	LOAD	pred	pred	pred	pred	pred	0	1	0	0	opc	opc	opc	opc	dest	dest	dest	dest	dest	dest	src2	src2	src2	src2	src2	src2	src1	src1	src1	src1	src1	src1	stop
		p4	p3	p2	p1	p0					0	o2	o1	o0	d5	d4	d3	d2	d1	d0	s5	s4	s3	s2	s1	s0	s5	s4	s3	s2	s2	s0	0/1
LD_{RI}		pred	pred	pred	pred	pred	0	1	0	1	opc	opc	opc	opc	dest	dest	dest	dest	dest	dest	imm	imm	imm	imm	imm	imm	src1	src1	src1	src1	src1	src1	stop
		p4	p3	p2	p1	p0					0	o2	o1	o0	d5	d4	d3	d2	d1	d0	i5	i4	i3	i2	i1	i0	s5	s4	s3	s2	s2	s0	0/1
ST_{RI}	STORE	pred	pred	pred	pred	pred	0	1	0	1	opc	opc	opc	opc	imm	imm	imm	imm	imm	imm	src3	src3	src3	src3	src3	src3	src1	src1	src1	src1	src1	src1	stop
		p4	p3	p2	p1	p0					1	o2	o1	o0	i5	i4	i3	i2	i1	i0	s5	s4	s3	s2	s1	s0	s5	s4	s3	s2	s2	s0	0/1
A_{RR}	ARITHM.	pred	pred	pred	pred	pred	1	opc	opc	0	opc	opc	opc	opc	dest	dest	dest	dest	dest	dest	src2	src2	src2	src2	src2	src2	src1	src1	src1	src1	src1	src1	stop
		p4	p3	p2	p1	p0		o5	o4		o3	o2	o1	o0	d5	d4	d3	d2	d1	d0	s5	s4	s3	s2	s1	s0	s5	s4	s3	s2	s2	s0	0/1
A_{RI}	SIMD	pred	pred	pred	pred	pred	1	opc	opc	1	opc	opc	opc	opc	dest	dest	dest	dest	dest	dest	imm	imm	imm	imm	imm	imm	src1	src1	src1	src1	src1	src1	stop
		p4	p3	p2	p1	p0		o5	o4		o3	o2	o1	o0	d5	d4	d3	d2	d1	d0	i5	i4	i3	i2	i1	i0	s5	s4	s3	s2	s2	s0	0/1
A_I	...	pred	pred	pred	pred	pred	1	opc	opc	1	opc	opc	opc	opc	dest	dest	dest	dest	dest	dest	imm	imm	imm	imm	imm	imm	src1	src1	src1	src1	src1	src1	stop
		p4	p3	p2	p1	p0		o5	o4		o3	o2	o1	o0	d5	d4	d3	d2	d1	d0	i5	i4	i3	i2	i1	i0	s5	s4	s3	s2	s2	s0	0/1
P_{RR}	PREDICAT	pred	pred	pred	pred	pred	0	1	1	0	opc	opc	opc	opc		dest	dest	dest	dest	dest	src2	src2	src2	src2	src2	src2	src1	src1	src1	src1	src1	src1	stop
		p4	p3	p2	p1	p0					o3	o2	o1	o0		d4	d3	d2	d1	d0	s5	s4	s3	s2	s1	s0	s5	s4	s3	s2	s2	s0	0/1
P_{RI}		pred	pred	pred	pred	pred	0	1	1	1	opc	opc	opc	opc		dest	dest	dest	dest	dest	imm	imm	imm	imm	imm	imm	src1	src1	src1	src1	src1	src1	stop
		p4	p3	p2	p1	p0					o3	o2	o1	o0		d4	d3	d2	d1	d0	i5	i4	i3	i2	i1	i0	s5	s4	s3	s2	s2	s0	0/1
L_{RR}	LOGICAL	pred	pred	pred	pred	pred	0	0	1	0	opc	opc	opc	opc	dest	dest	dest	dest	dest	dest	src2	src2	src2	src2	src2	src2	src1	src1	src1	src1	src1	src1	stop
	SHIFT	p4	p3	p2	p1	p0					o3	o2	o1	o0	d5	d4	d3	d2	d1	d0	s5	s4	s3	s2	s1	s0	s5	s4	s3	s2	s2	s0	0/1
L_{RI}		pred	pred	pred	pred	pred	0	0	1	1	opc	opc	opc	opc	dest	dest	dest	dest	dest	dest	imm	imm	imm	imm	imm	imm	src1	src1	src1	src1	src1	src1	stop
		p4	p3	p2	p1	p0					o3	o2	o1	o0	d5	d4	d3	d2	d1	d0	i5	i4	i3	i2	i1	i0	s5	s4	s3	s2	s2	s0	0/1
J_R	JMP/JMPL	pred	pred	pred	pred	pred	0	0	0	0		opc	opc	opc							src2	src2	src2	src2	src2	src2							stop
		p4	p3	p2	p1	p0						o2	o1	1							s5	s4	s3	s2	s1	s0							0/1
J_I	BR/BRL	pred	pred	pred	pred	pred	0	0	0	1	imm	opc	opc	opc	imm	imm	imm	imm	imm	imm	imm	imm	imm	imm	imm	imm	imm	imm	imm	imm	imm	imm	stop
		p4	p3	p2	p1	p0					i18	o2	o1	1	i17	i16	i15	i14	i13	i12	i5	i4	i3	i2	i1	i0	i11	i10	i9	i8	i7	i6	0/1
MSP	MV2/FSR	pred	pred	pred	pred	pred	0	0	0	0	opc	opc	opc	opc	dest	dest	dest	dest	dest	dest	src2	src2	src2	src2	src2	src2							stop
		p4	p3	p2	p1	p0					o3	0	1	1	d5	d4	d3	d2	d1	d0	s5	s4	s3	s2	s1	s0							0/1
SET	SETLO/HI	imm	pred	pred	pred	pred	0	0	0	imm	imm	imm	opc	opc	imm	imm	imm	imm	imm	imm	imm	imm	imm	imm	imm	imm	dest	dest	dest	dest	dest	dest	stop
		i15	p3	p2	p1	p0				i14	i13	i12	o1	0	i11	i10	i9	i8	i7	i6	i5	i4	i3	i2	i1	i0	d5	d4	d3	d2	d1	d0	0/1
HALT	HALT	pred	pred	pred	pred	pred	0	0	0		opc	opc	opc	opc																			stop
		p4	p3	p2	p1	p0					0	0	1	1																			0/1
CGA_R	CGA	pred	pred	pred	pred	pred	0	0	0	0	opc	opc	opc	opc							src2	src2	src2	src2	src2	src2							stop
		p4	p3	p2	p1	p0					1	1	1	1							s5	s4	s3	s2	s1	s0							0/1
CGA_I		pred	pred	pred	pred	pred	0	0	0	1	opc	opc	opc	opc	imm	imm	imm	imm	imm	imm	imm	imm	imm	imm	imm	imm	imm	imm	imm	imm	imm	imm	stop
		p4	p3	p2	p1	p0					1	1	1	1	i17	i16	i15	i14	i13	i12	i5	i4	i3	i2	i1	i0	i11	i10	i9	i8	i7	i6	0/1

Author(s): Bjorn De Sutter , Mladen Berekovic, Andreas Kanstein

Figure B.2: Instruction Set Architecture ADRESv1

C

Interconnection Options

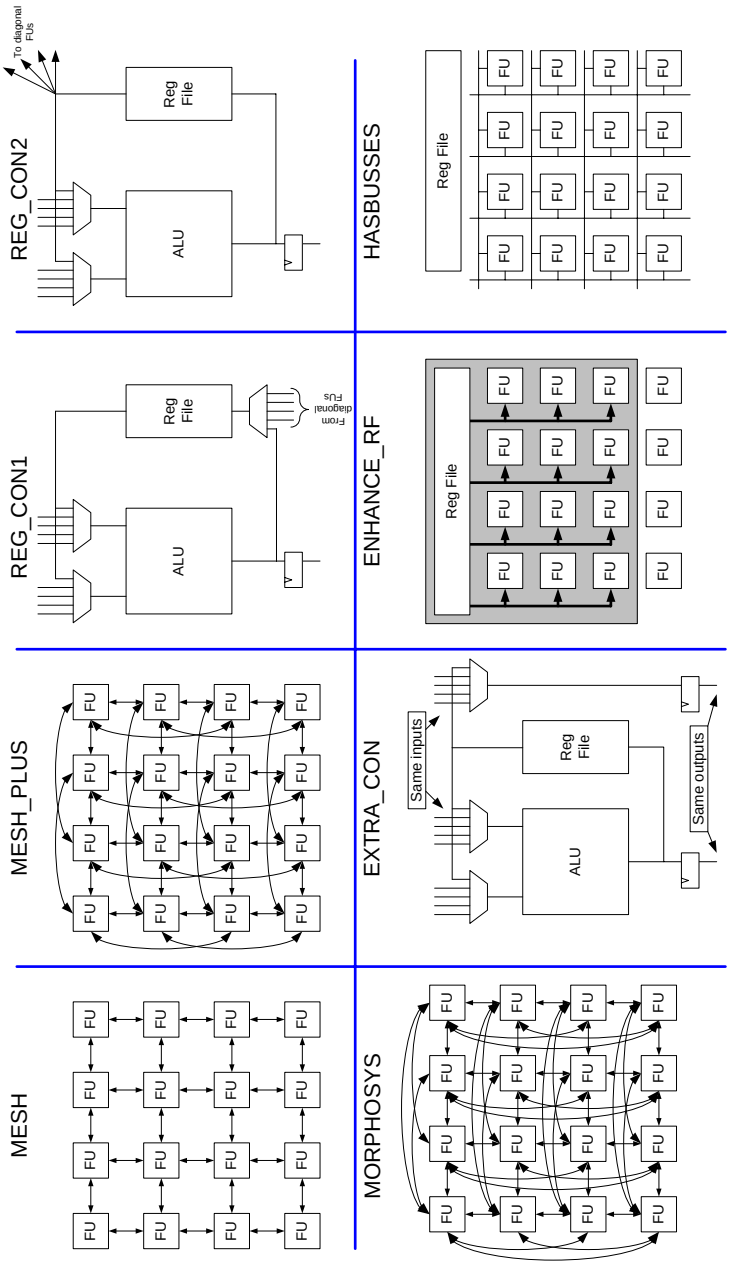


Figure C.1: Interconnection Options for Architectural Experiments

Architectural Explorations

D

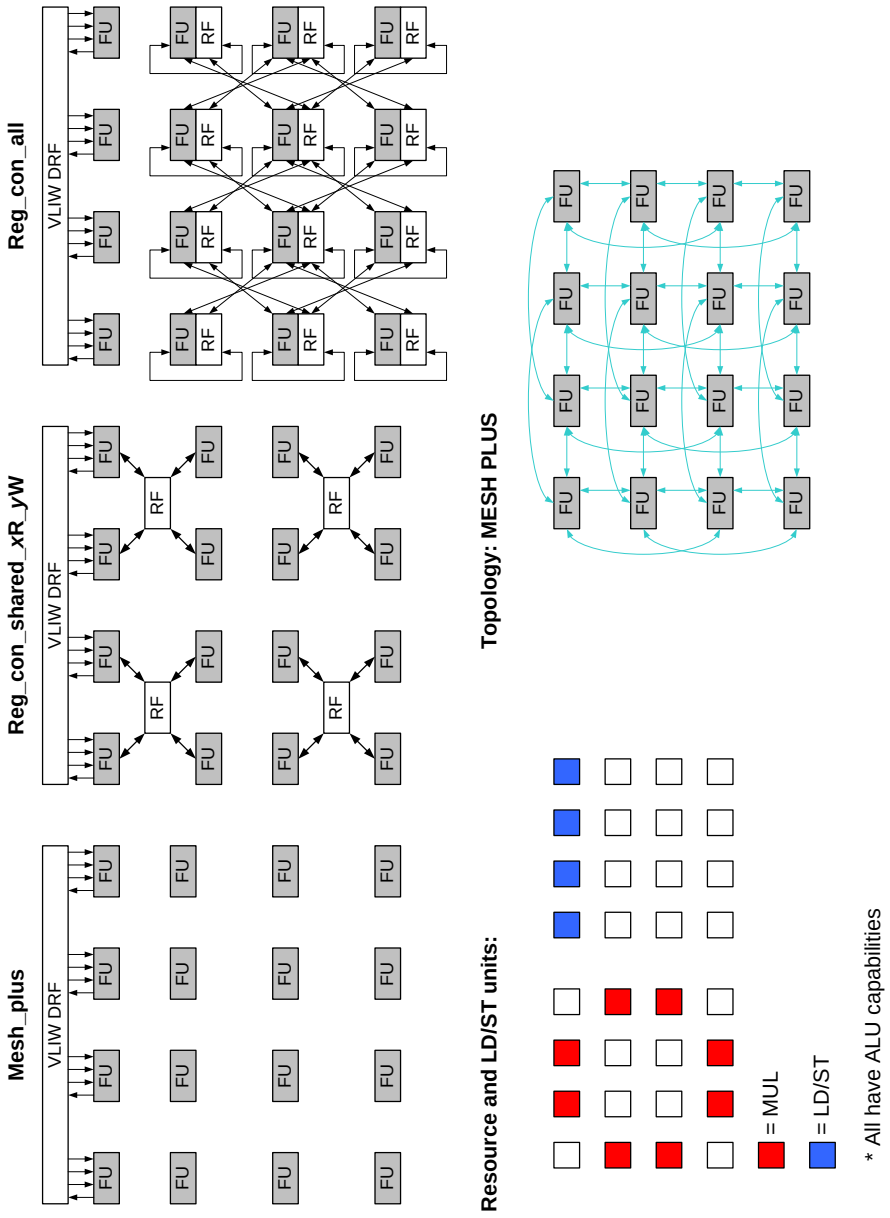


Figure D.1: Distributing the Local Data Register Files

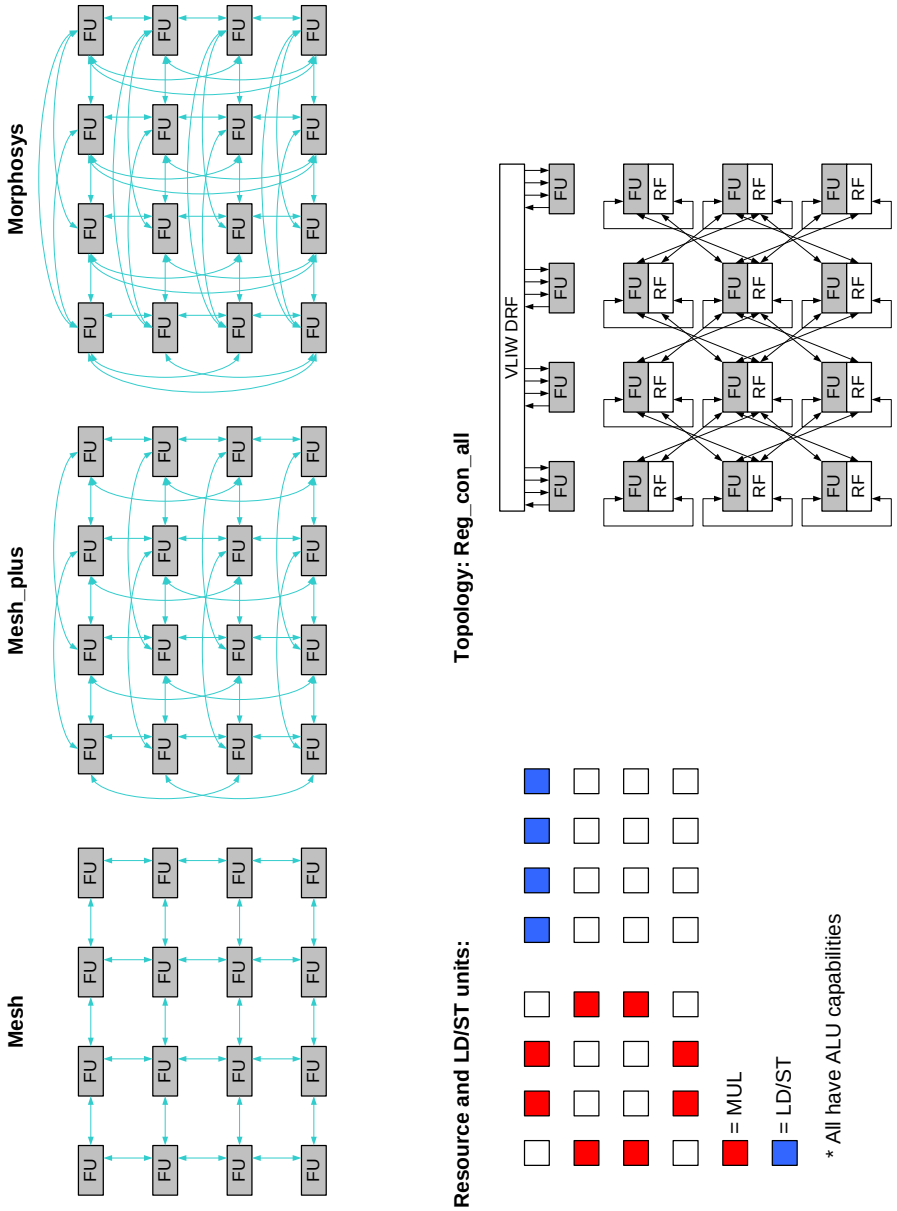
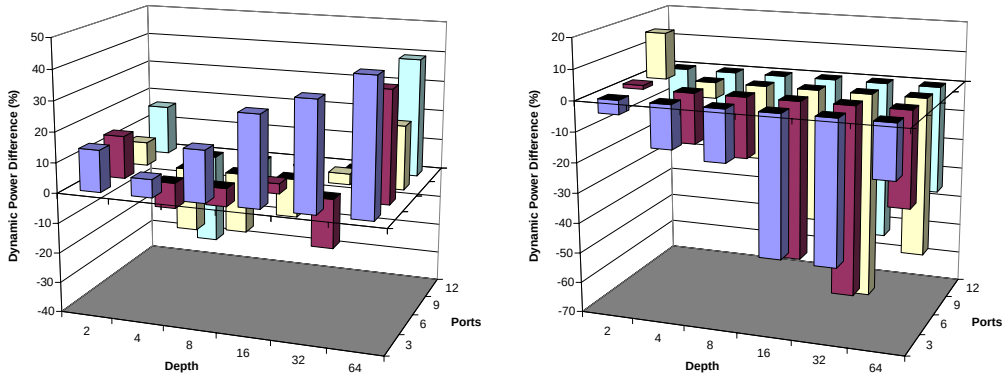


Figure D.2: Interconnection Topologies

DRF Library Selection

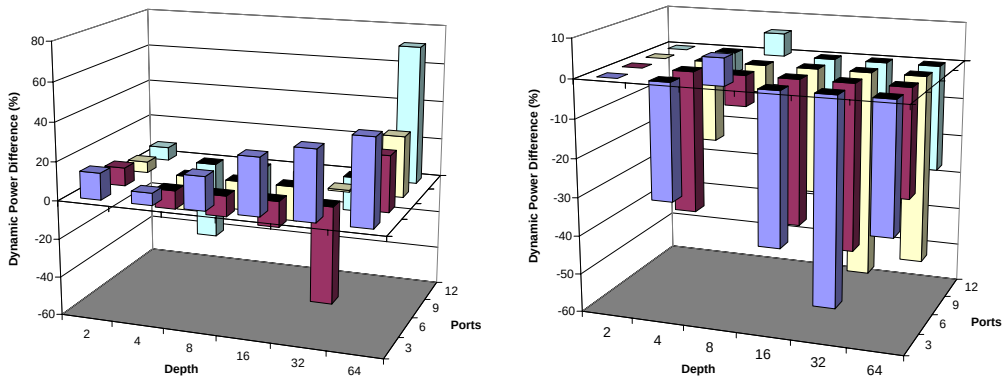
E

This appendix depicts the differences between Artisan and Synopsys libraries for data register files. The difference is referenced from the Artisan libraries and calculated as: $\frac{Synopsys\ values - Artisan\ values}{Artisan\ values}$ So, a negative percentage value is in favor of Synopsys libraries and consequently a positive value is in favor of the Artisan libraries.



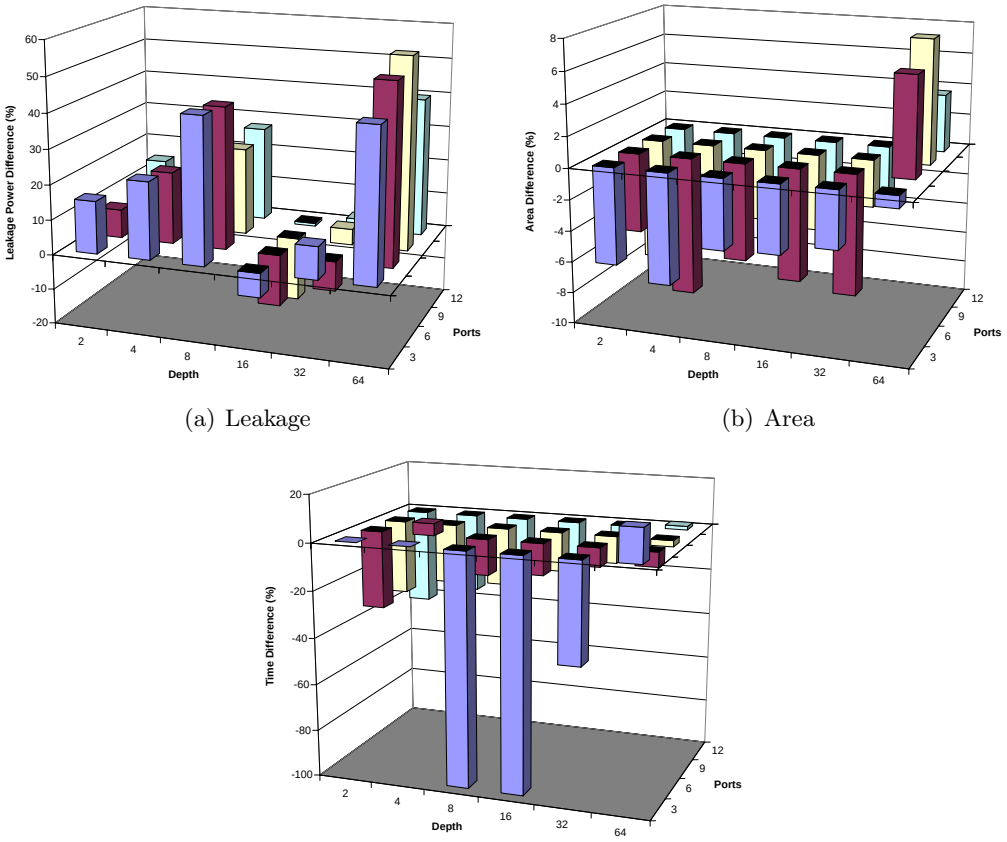
(a) Write (b) Read

Figure E.1: CGA DRF differences between Synposis vs. Artisan for when accessing all ports



(a) Write (b) Read

Figure E.2: CGA DRF differences between Synposis vs. Artisan for when accessing first port only

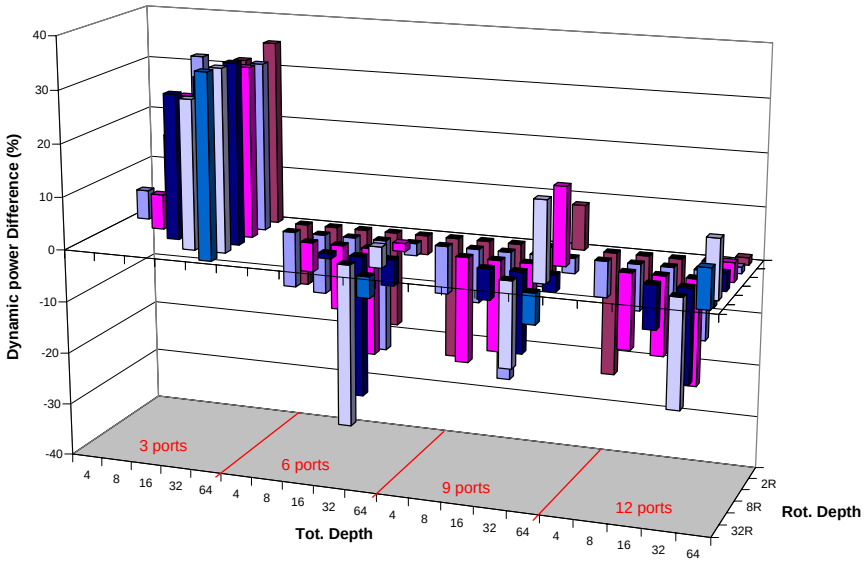


(a) Leakage

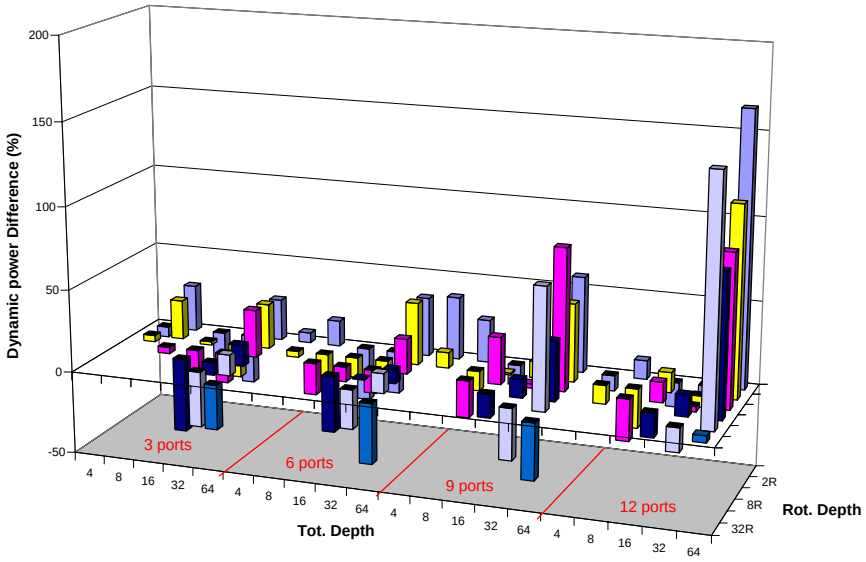
(b) Area

(c) Time

Figure E.3: CGA DRF differences between Synposis vs. Artisan

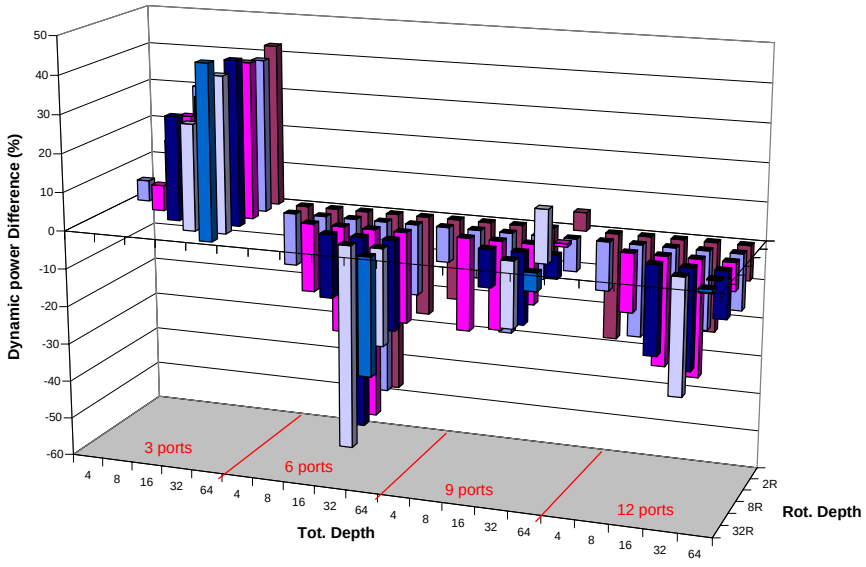


(a) Write

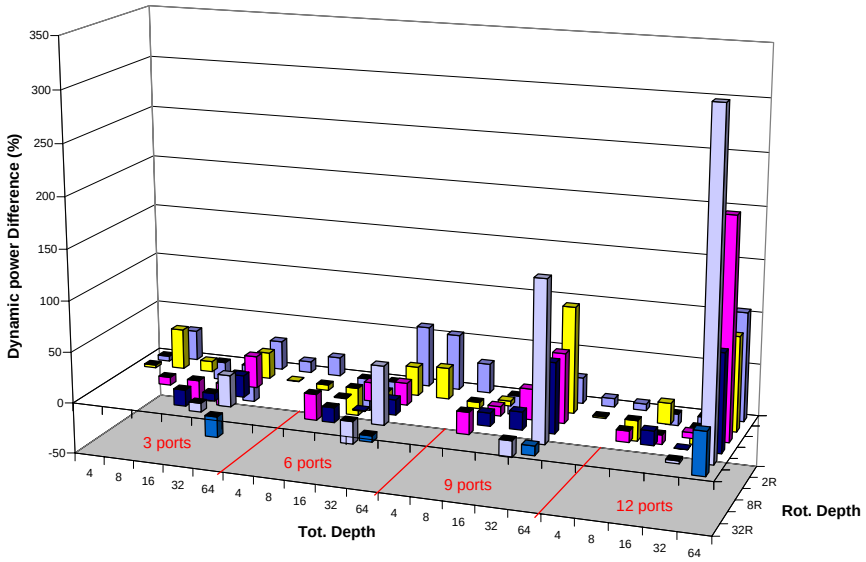


(b) Read

Figure E.4: VLIW DRF differences between Synposis vs. Artisan for when accessing all ports

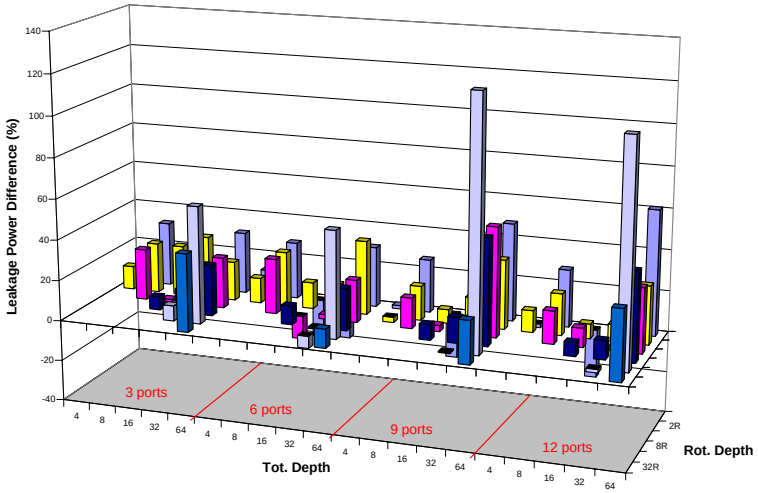


(a) Write

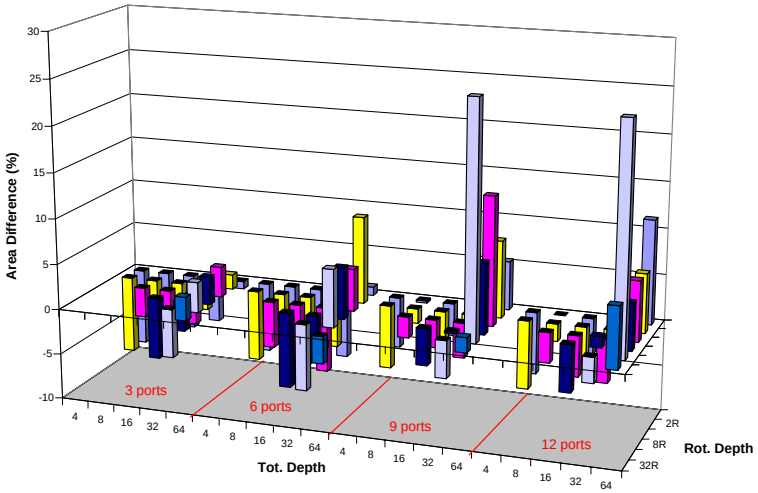


(b) Read

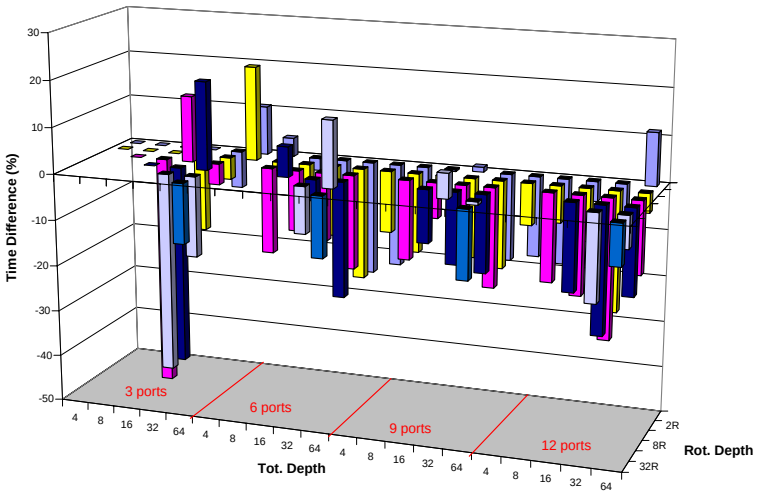
Figure E.5: VLIW DRF differences between Synposis vs. Artisan for when accessing first port only



(a) Leakage

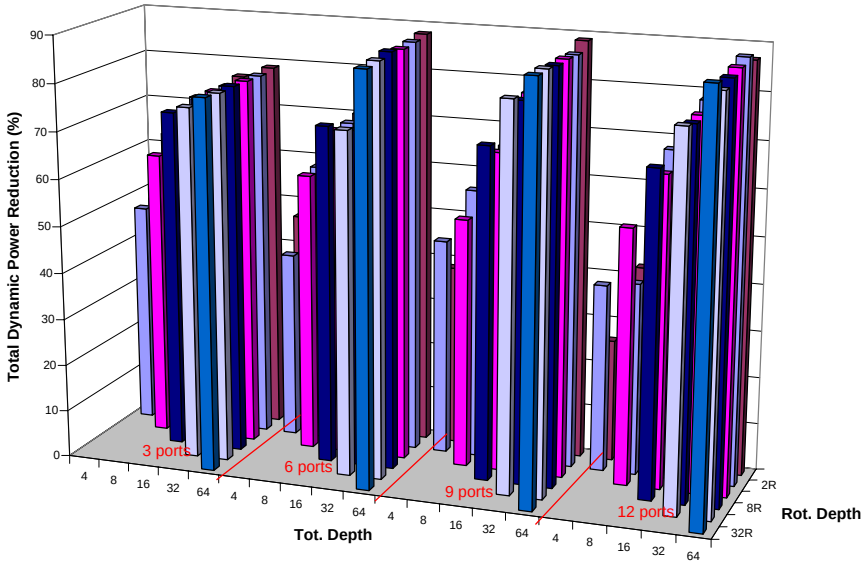


(b) Area

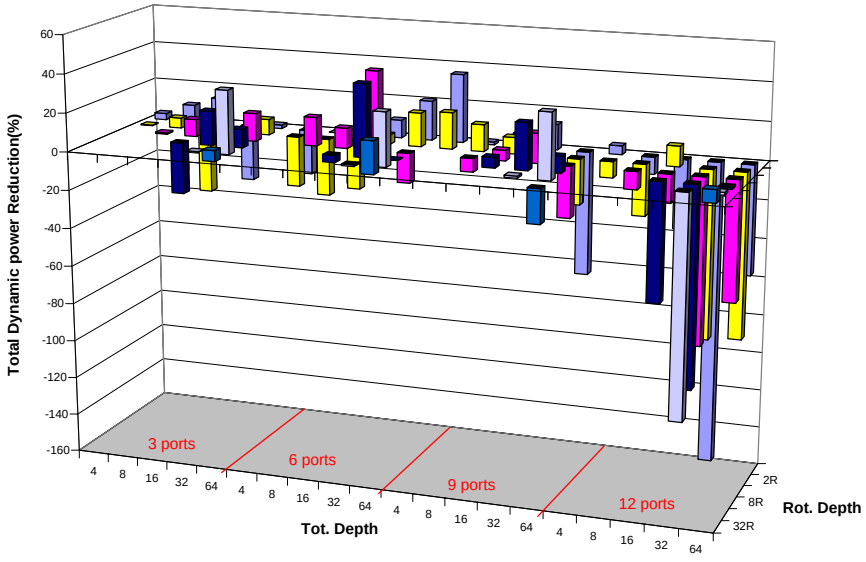


(c) Time

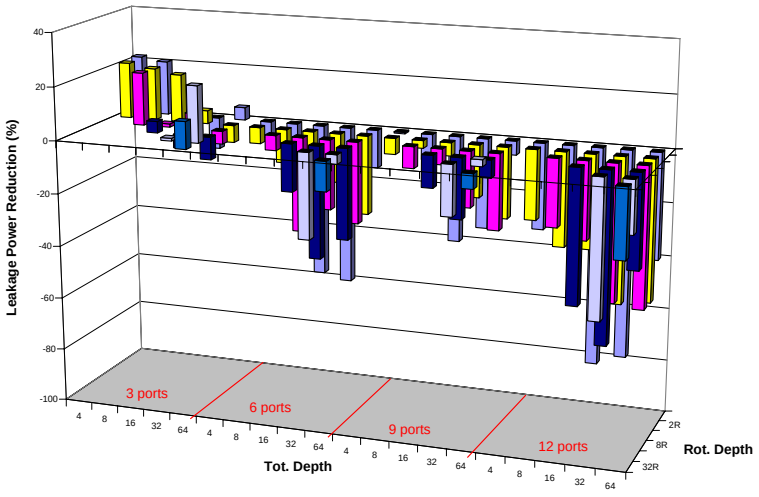
Figure E.6: VLIW DRF differences between Synposis vs. Artisan



(a) Write



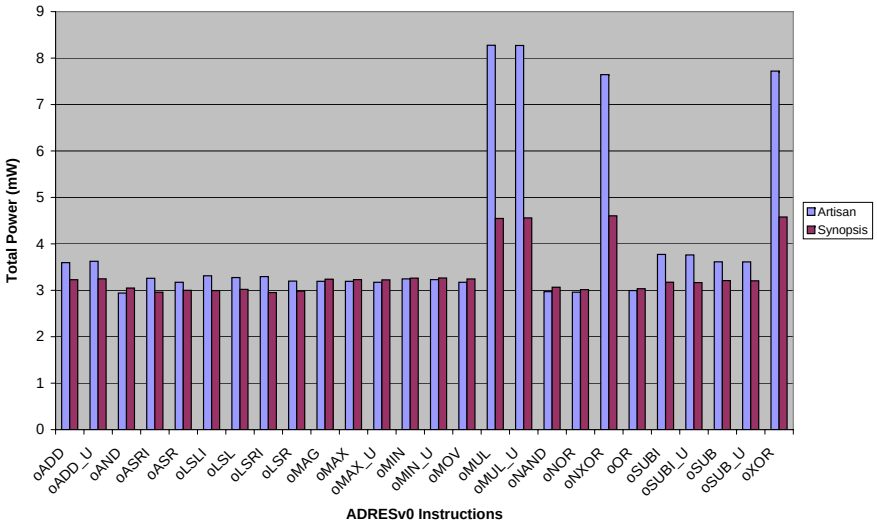
(b) Read



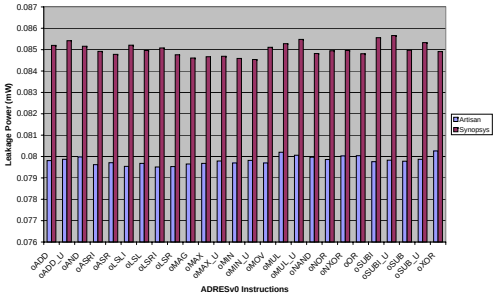
(c) Leakage

Figure E.7: VLIW DRF differences between no optimizations and clock gating with Synposis libraries when accessing first port only

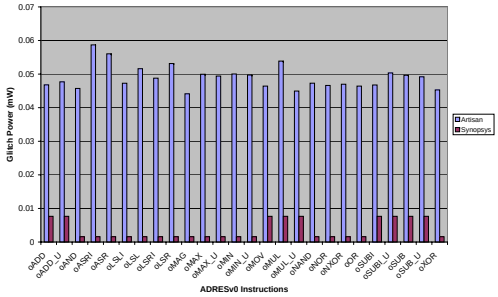
FU Library Selection



(a) Total Power



(b) Leakage Power

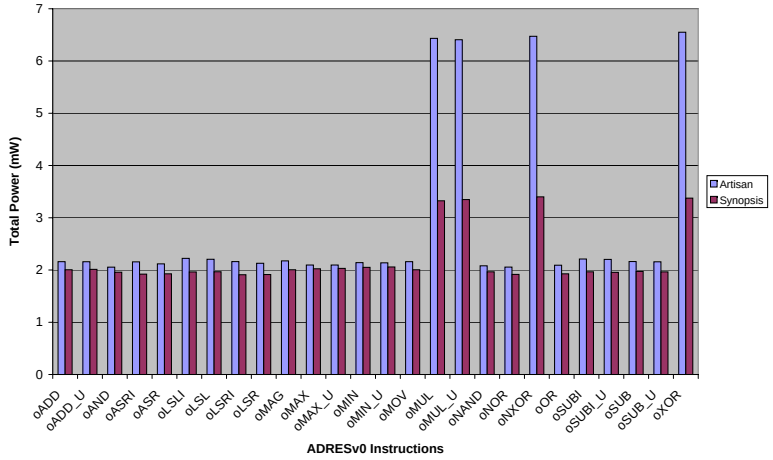


(c) Glitch Power

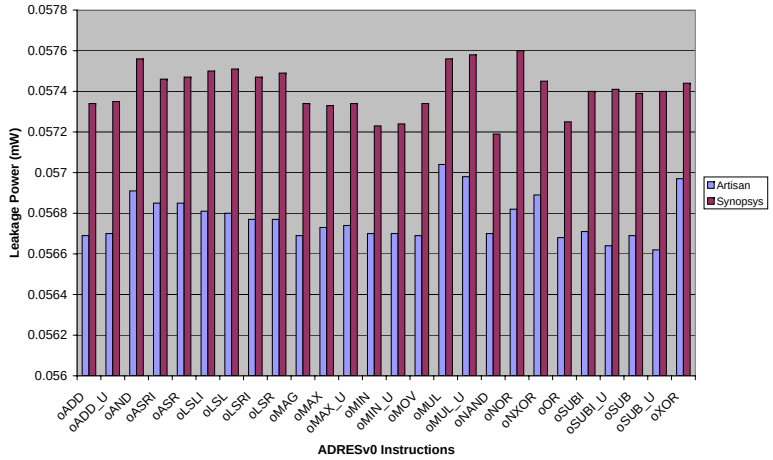
Figure F.1: Power Results of Instructions for non-pipelined Functional Unit in CGA Section

Table F.1: CGA FU Area Results between Artisan and Synopsys Libraries

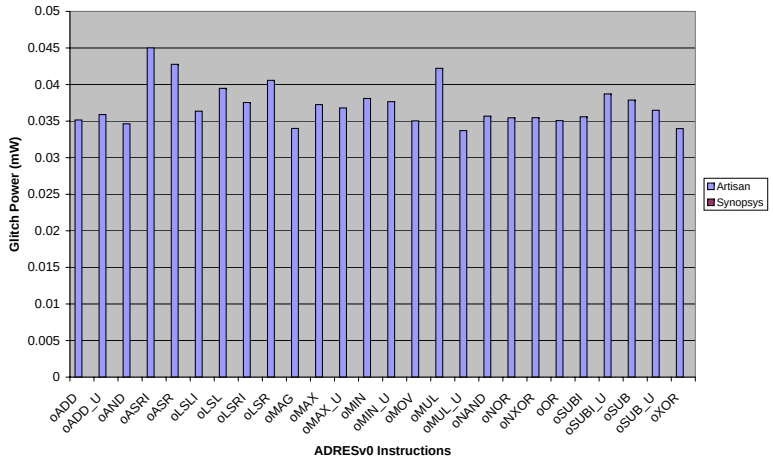
	Artisan	Synopsys
FU_CGA	457.23	630.81
ALU	22951.75	20523.08
Miscellaneous	8860.22	8834.82
Total	32269.20	29988.71



(a) Total Power

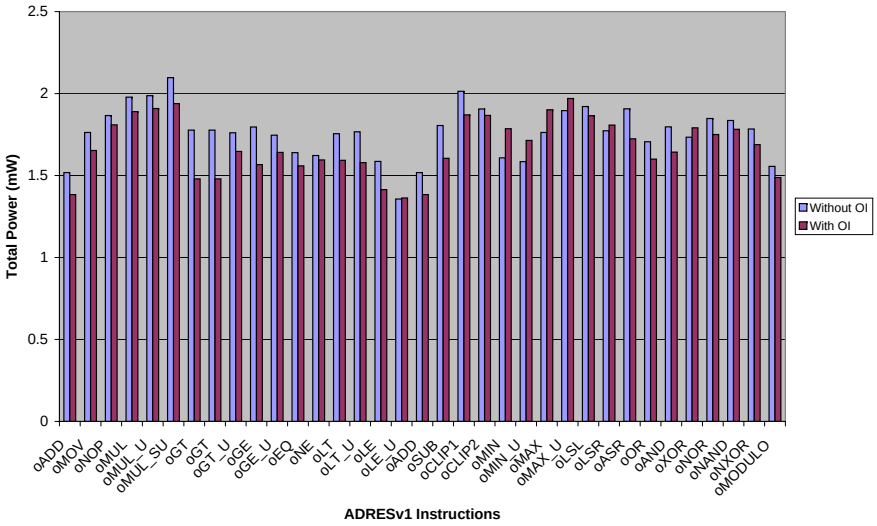


(b) Leakage Power

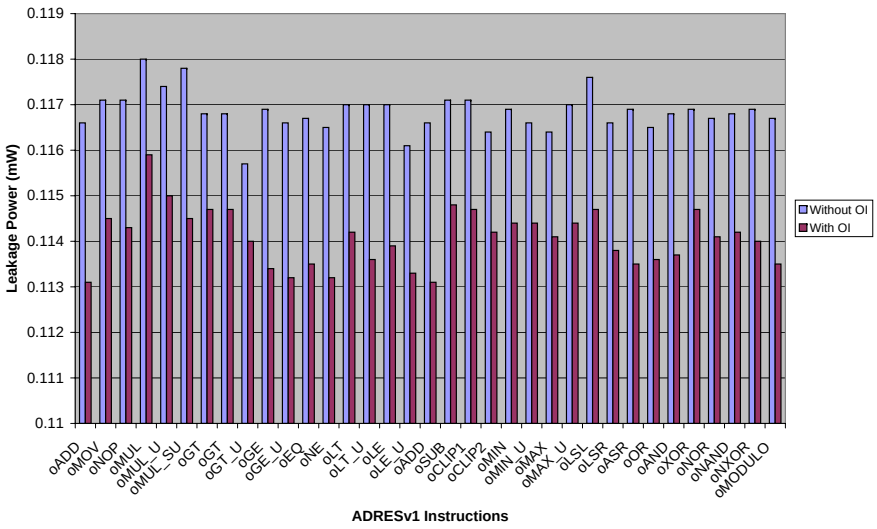


(c) Glitch Power

Figure F.2: Power Results of Instructions for non-pipelined ALU in FU CGA Section



(a) Total Power

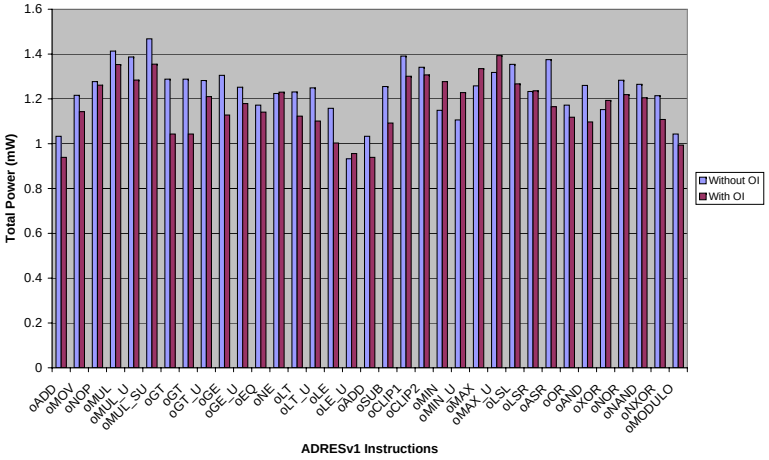


(b) Leakage Power

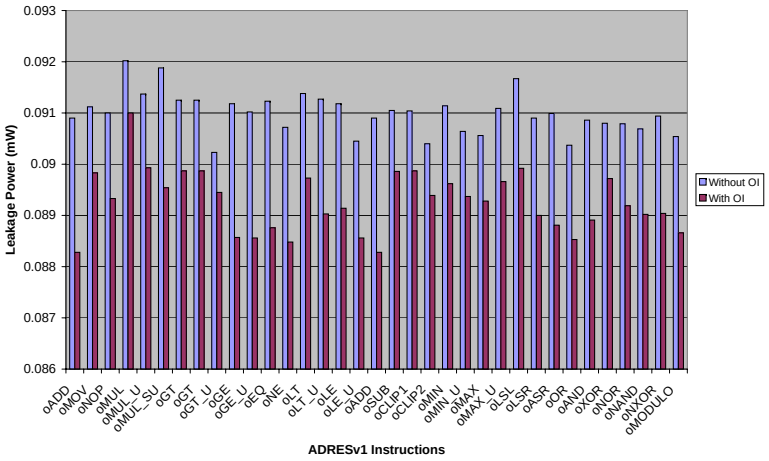
Figure F.3: Power Results of Instructions for pipelined Functional Unit in CGA Section

Table F.2: CGA FU Area Results between without and with Operand Isolation with Synopsys Libraries

	Without OI	With OI
FU_CGA	1296.89	1224.22
ALU	35985.55	38537.60
Miscellaneous	8713.45	8959.86
Total	45995.89	48721.68



(a) Total Power



(b) Leakage Power

Figure F.4: Power Results of Instructions for pipelined ALU in FU CGA Section

Milestone Results

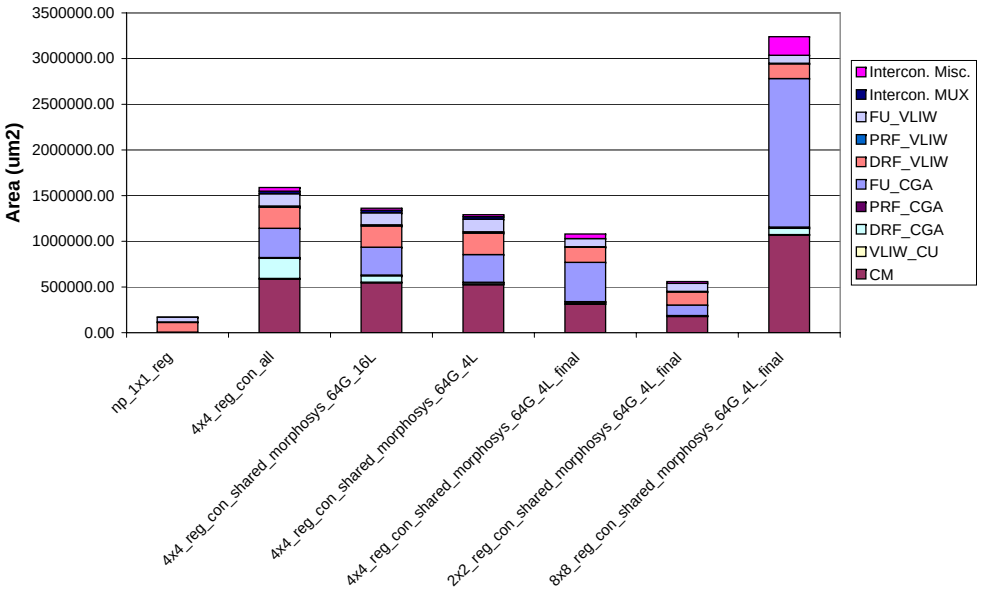
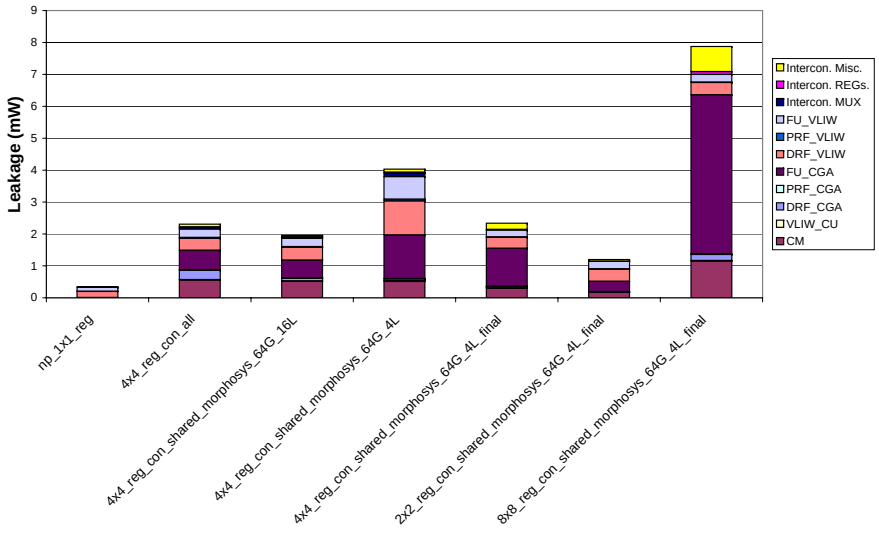
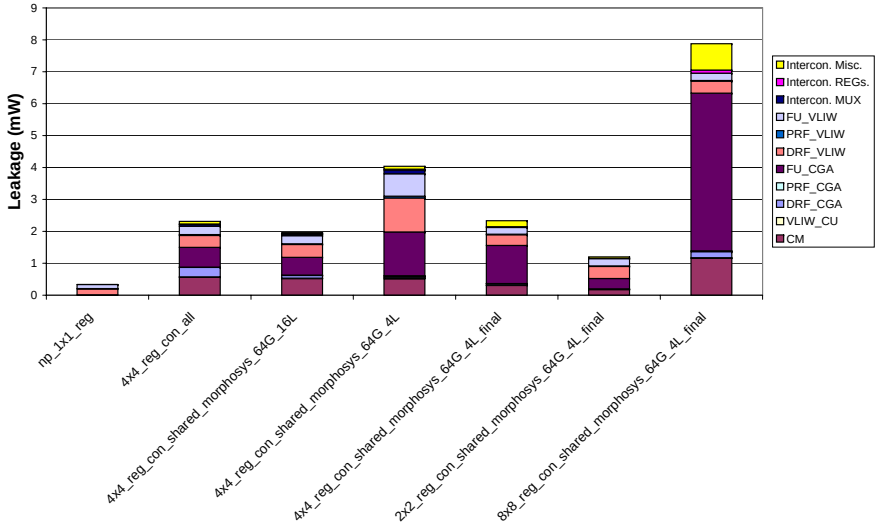


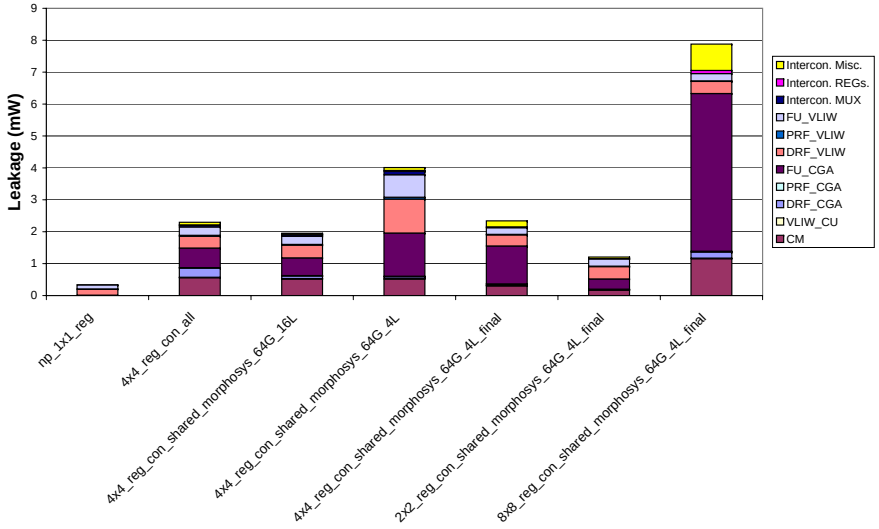
Figure G.1: Milestones Area



(a) Milestones FFT

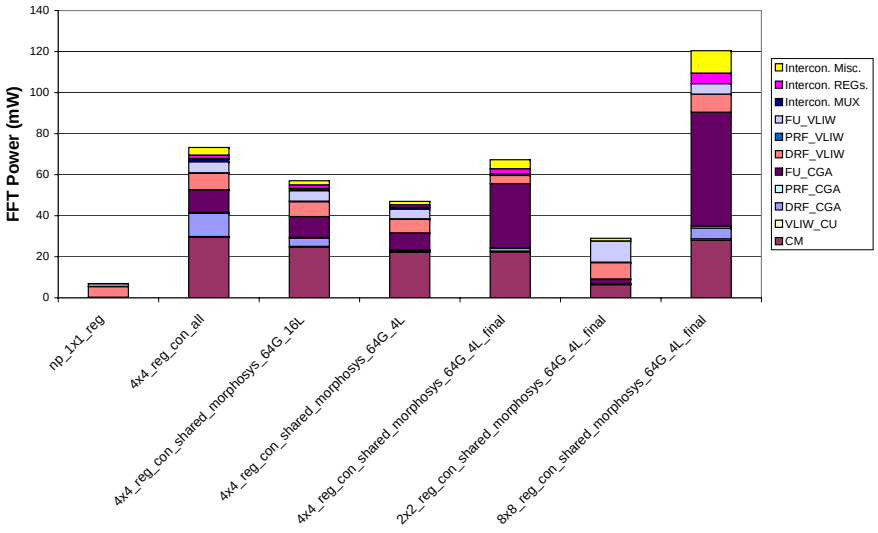


(b) Milestones IDCT

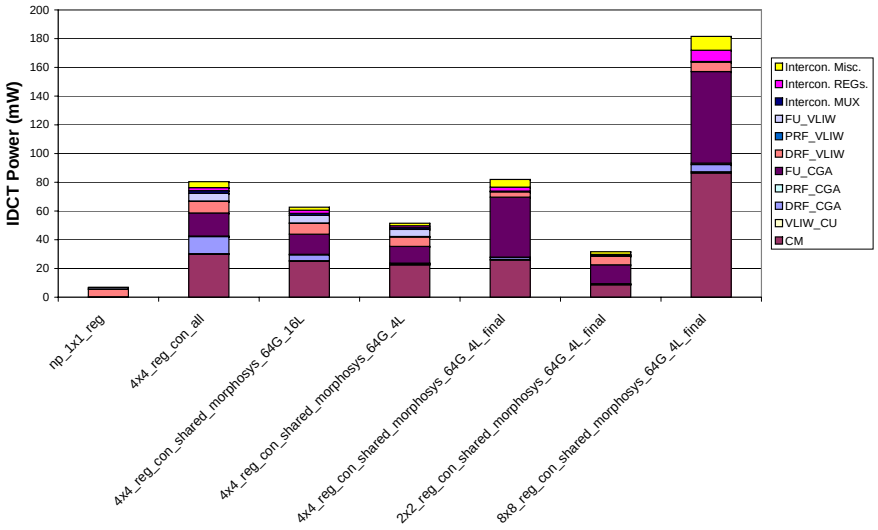


(c) Milestones MPEG2

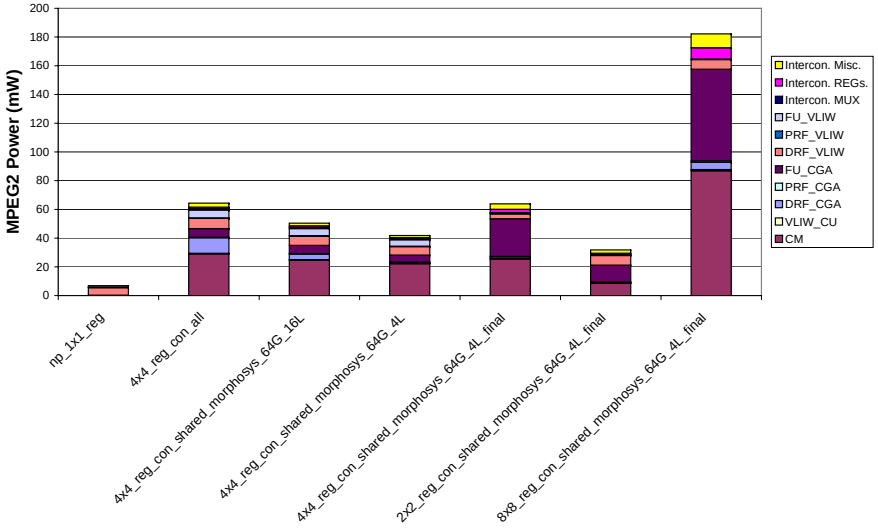
Figure G.2: Milestones Leakage



(a) Milestones FFT

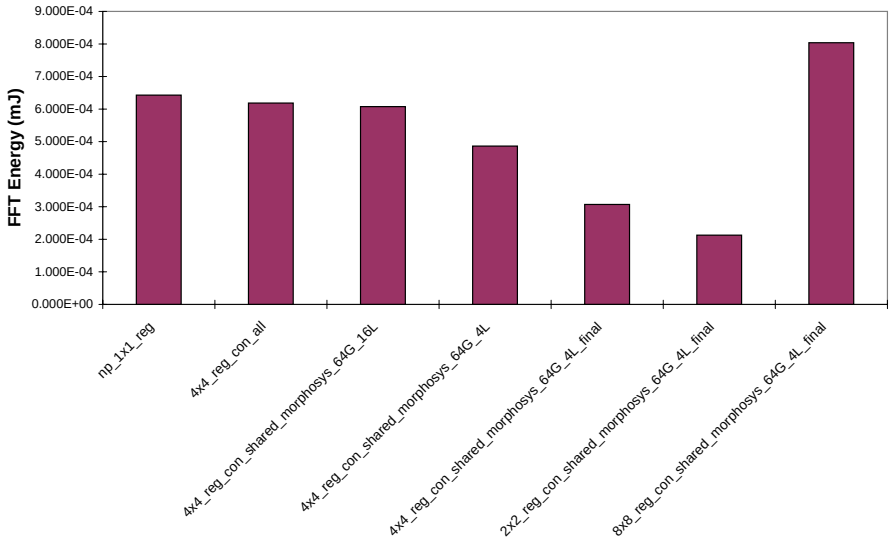


(b) Milestones IDCT

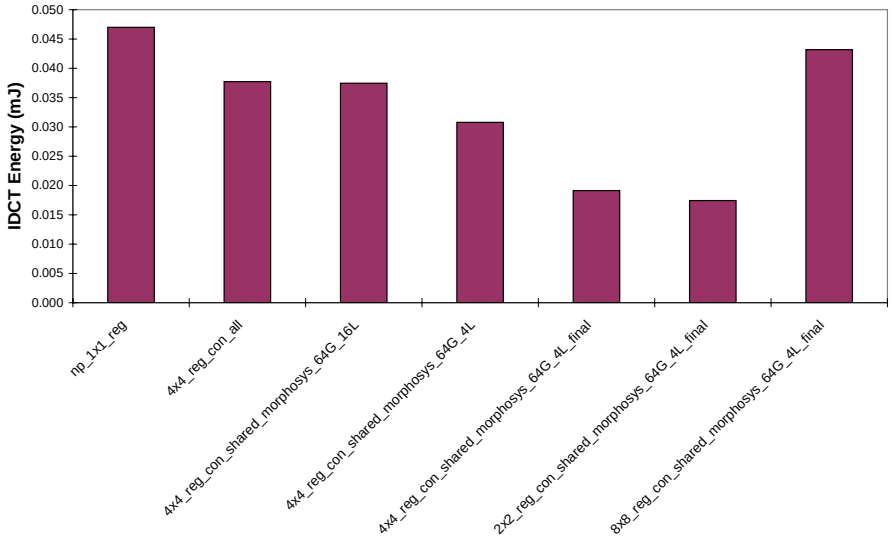


(c) Milestones MPEG2

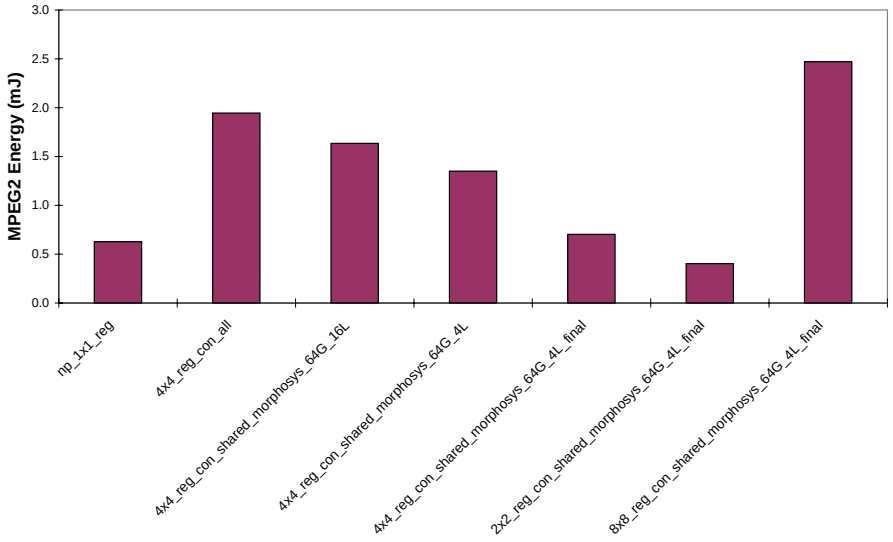
Figure G.3: Milestones Power



(a) Milestones FFT

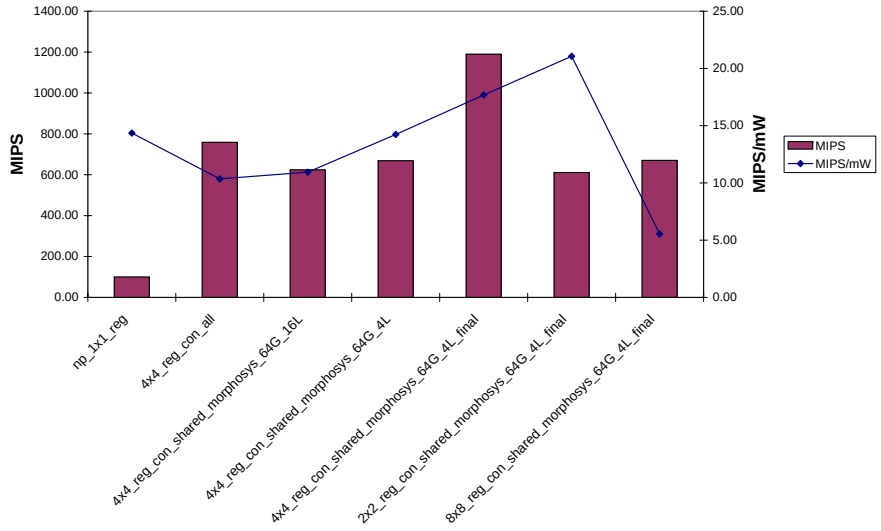


(b) Milestones IDCT

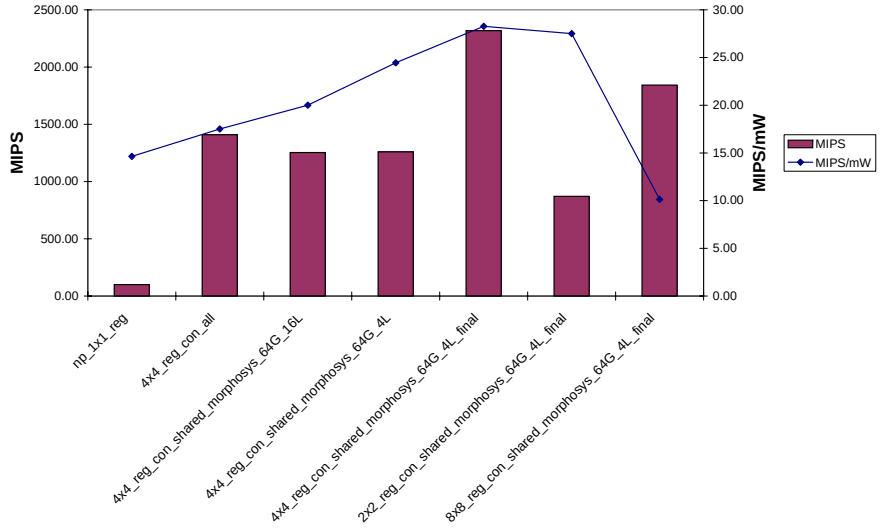


(c) Milestones MPEG2

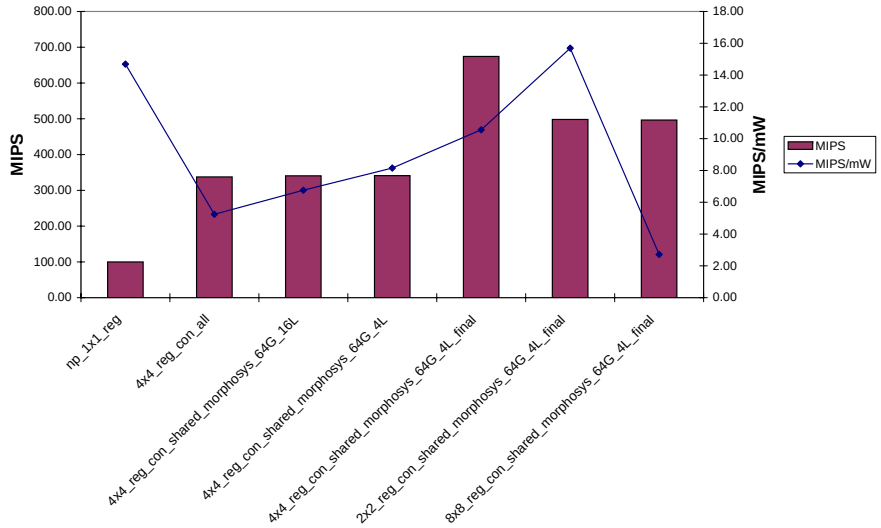
Figure G.4: Milestones Energy



(a) Milestones FFT

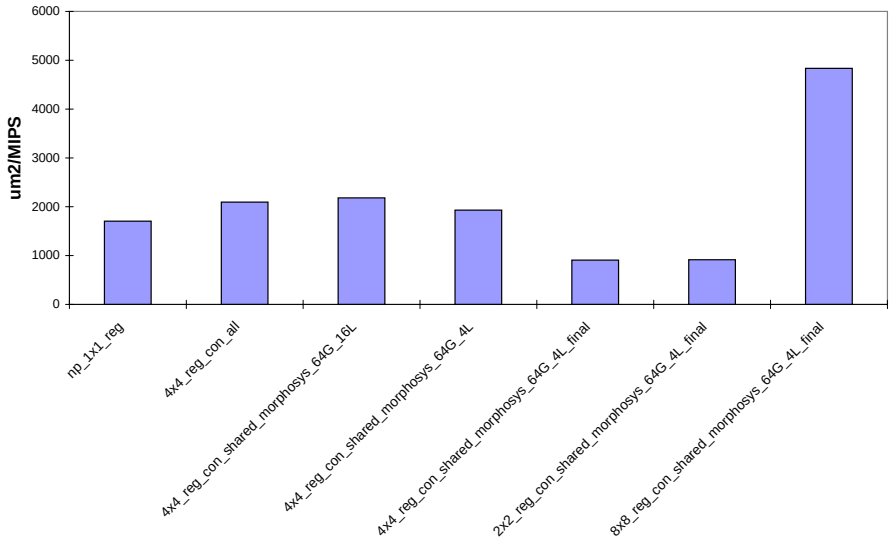


(b) Milestones IDCT

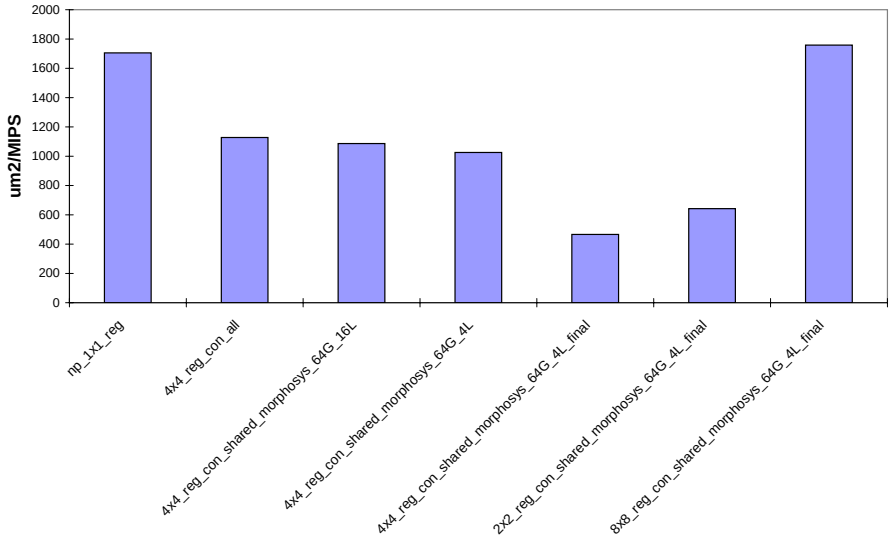


(c) Milestones MPEG2

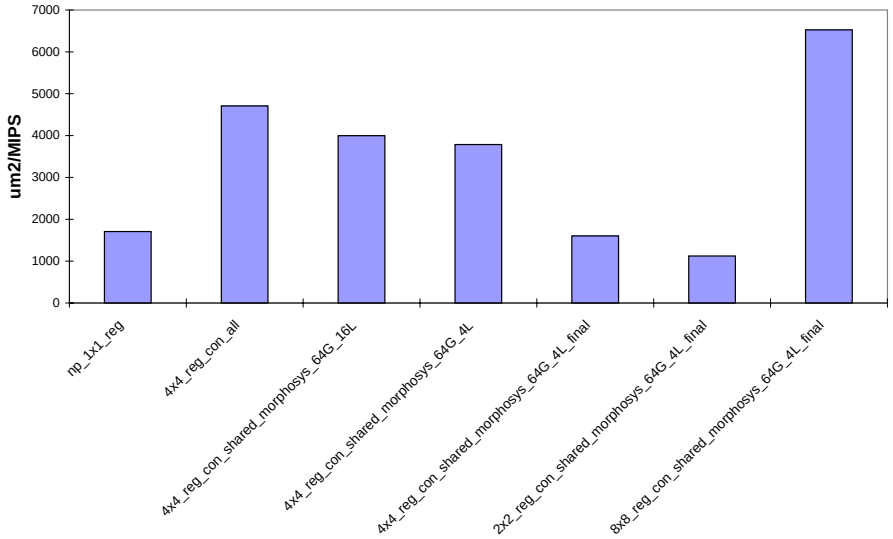
Figure G.5: Milestones Performance



(a) Milestones FFT

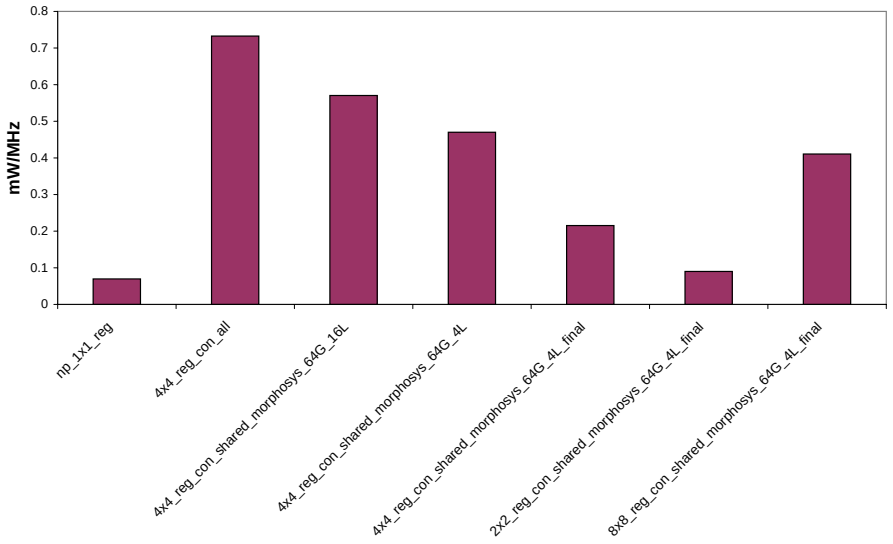


(b) Milestones IDCT

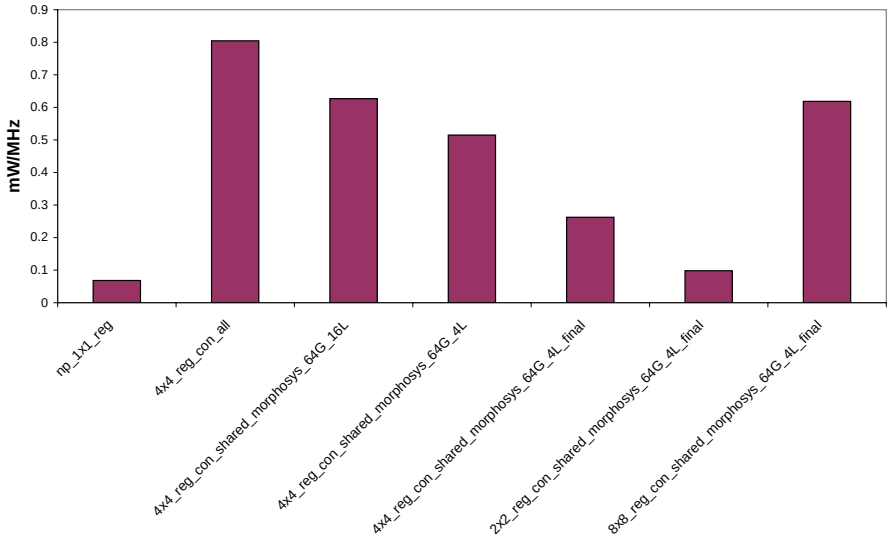


(c) Milestones MPEG2

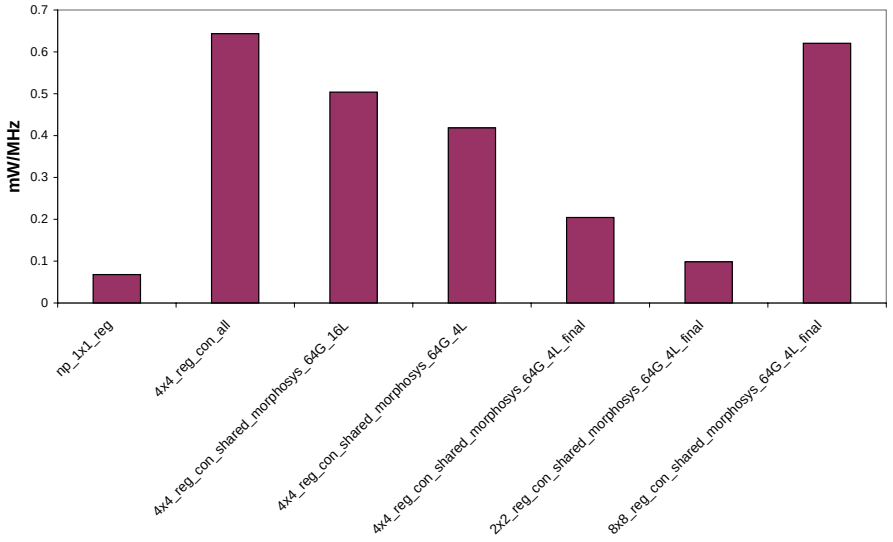
Figure G.6: Milestones Area vs. Performance



(a) Milestones FFT

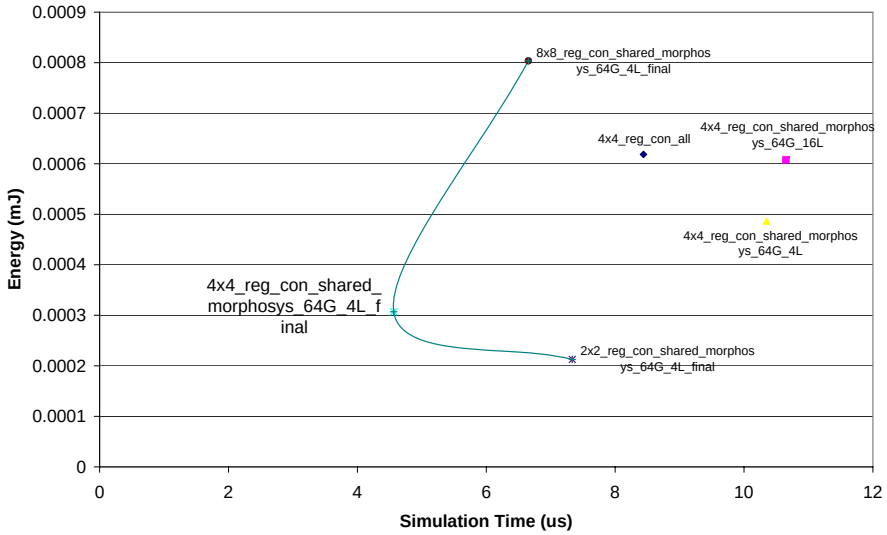


(b) Milestones IDCT

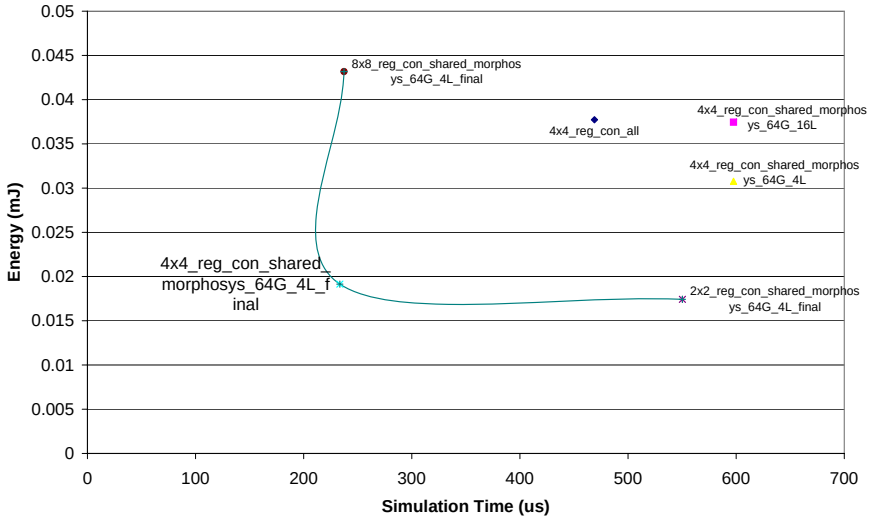


(c) Milestones MPEG2

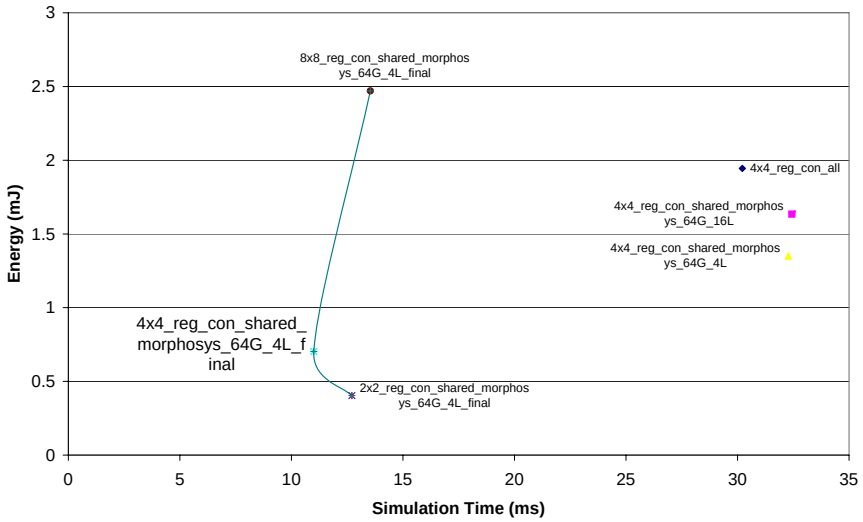
Figure G.7: Milestones Power vs. Frequency



(a) Milestones FFT



(b) Milestones IDCT



(c) Milestones MPEG2

Figure G.8: Milestones Energy-Delay

Curriculum Vitae

Frank Bouwens was born in Oostburg, the Netherlands on June 27, 1980. After receiving his intermediate vocational education diploma in Computer Interface Technology at the department of Electronics in July 2000, he joined the Electronic System Engineering department at the Zeeland University of Professional Educational. He gained experience in embedded and automotive systems at Philips Semiconductors in Nijmegen, the Netherlands before receiving his Bachelor of Science in Electronic System Engineering in July 2003. Eager for knowledge and his passion for embedded systems he started his Master of Science study in Computer Engineering of the Electrical Engineering department at the Delft University of Technology in September 2003. His research interests are advanced low power, high speed computer architectures for embedded multi-media systems and hardware/software co-design.