Fakhar Anjam

Run-time Adaptable VLIW Processors

Resources, Performance, Power Consumption, and Reliability Trade-offs

Run-time Adaptable VLIW Processors

Resources, Performance, Power Consumption, and Reliability Trade-offs

---

PROEFSCHRIFT

ter verkrijging van de graad van doctor
aan de Technische Universiteit Delft,
op gezag van de Rector Magnificus prof. ir. K.C.A.M. Luyben,
voorzitter van het College voor Promoties,
in het openbaar te verdedigen

op dinsdag 27 augustus 2013 om 15:00 uur

door

Fakhar ANJAM

Master of Science in Information Technology
Pakistan Institute of Engineering and Applied Sciences (PIEAS), Islamabad

geboren te Karak, Pakistan

Dit proefschrift is goedgekeurd door de promotor:
Prof. dr. K.L.M. Bertels

Copromotor:
Dr. ir. J.S.S.M. Wong

Samenstelling promotiecommissie:

Rector Magnificus	voorzitter
Prof. dr. K.L.M. Bertels	Technische Universiteit Delft, promotor
Dr. ir. J.S.S.M. Wong	Technische Universiteit Delft, copromotor
Prof. dr. E. Charbon	Technische Universiteit Delft
Prof. dr. L. Carro	Universidade Federal do Rio Grande do Sul, Brazilië
Prof. Dr.-Ing. H. Blume	Leibniz Universität Hannover, Duitsland
Prof. Dr.-Ing. M. Hübner	Ruhr-Universität Bochum, Duitsland
Prof. dr. ir. G.N. Gaydadjiev	Chalmers University of Technology, Zweden
Prof. dr. G.J.T. Leus	Technische Universiteit Delft, reservelid

This thesis has been completed in partial fulfillment of the requirements of the Delft University of Technology (Delft, The Netherlands) for the award of PhD degree. The research described in this thesis was supported in parts by: (1) CE Lab. Delft University of Technology, (2) HEC Pakistan.

Published and distributed by: Fakhar Anjam Email: imfakhar@gmail.com

ISBN: 978-94-6186-191-7

Keywords: Computer Architecture, Parallel Execution, Softcore Processors, VLIW Processors, Run-time Reconfiguration, Fault Tolerance, Customization, Parametrization, FPGAs, Trade-offs

Cover page designed by Hanike (www.hanike.nl).

Printed in The Netherlands

To my father and all other members of my family

Summary

In this dissertation, we propose to combine programmability with reconfigurability by implementing an adaptable programmable VLIW processor in a reconfigurable hardware. The approach allows applications to be developed at high-level (C language level), while at the same time, the processor organization can be adapted to the specific requirements (both static and dynamic) of different applications.

Our proposed customizable VLIW processor called ρ-VEX can be adapted at design-time as well as at run-time. Its instruction set architecture (ISA) is based on the VEX ISA and a toolchain (parametrized C compiler and simulator) is publicly available from Hewlett Packard (HP) for architectural exploration and code generation. The design-time parameters include the processor's issue-width, the type of different functional units (FUs) and their latencies, the type and size of multiported register files, degree of pipelining, size of instruction and data memories, type of interrupt and exception systems, selection of default custom operations, datapath sharing. If the behavior of applications is not known at design-time or an application has different phases with distinct requirements, a fixed processor may not perform efficiently for all the applications/phases. To this end, we propose a run-time reconfigurable processor that can adapt its organization dynamically during execution. The run-time parameters include the processor's issue-width, the type and number of different FUs, and the register file size. Additionally, we propose configurable fault tolerance techniques for the ρ-VEX processor. The designer can choose to include or exclude the fault tolerance in the processor at design-time. When the fault tolerance is included, it can be made permanently enabled or enabled/disabled at run-time. All these options enable users to trade-off between hardware area/resources, performance, power/energy consumption, and reliability. The processor is available as open-source.

Samenvatting

In dit proefschrift stellen we voor om programmeerbaarheid te combineren met reconfigureerbaarheid door het implementeren van een aanpasbare programmeerbare VLIW processor in herconfigureerbare hardware. De aanpak staat het ontwikkelen van toepassingen op hoog niveau (C programmeer taalniveau) toe, terwijl op hetzelfde moment de processor organisatie kan worden aangepast aan de specifieke eisen (zowel statisch als dynamisch) van verschillende toepassingen.

Onze voorgestelde aanpasbare VLIW processor, genaamd ρ-VEX, kan tijdens design-time evenals tijdens run-time aangepast worden. De instructie set architectuur (ISA) is gebaseerd op de VEX ISA en een toolchain (geparametriseerde C compiler en simulator) is publiek beschikbaar gesteld door Hewlett Packard (HP) voor architectuur exploratie en code generatie. De design-time parameters omvatten de processor issue-breedte, de aard van verschillende functionele eenheden (FU's) en hun latencies, het type en grootte van multiported register files, de mate van pipelining, de grootte van instructie en data geheugens, het type interrupt en exceptie systemen, selectie van standaard aangepaste bewerkingen, het delen van het datapad. Indien het gedrag van applicaties niet bekend is tijdens design-time of wanneer een applicatie verschillende fases kent met verschillende eisen, kan het zijn dat een vaste processor niet efficiënt is in het uitvoeren van alle applicaties/fasen. Daartoe stellen we een run-time herconfigureerbare processor voor die zijn organisatie tijdens het berekenen dynamisch kan aanpassen. De run-time parameters omvatten de processor issue-breedte, het type en aantal verschillende FUs, en het register bestandsgrootte. Daarnaast stellen we voor de ρ-VEX processor herconfigureerbare fouttolerantie technieken voor. De ontwerper kan kiezen voor wel of geen fouttolerantie in de processor tijdens design-time. Wanneer fouttolerantie is inbegrepen, kan deze permanent ingeschakeld worden of ingeschakeld/uitgeschakeld tijdens run-time. Al deze opties geven de gebruikers de mogelijkheid om een afweging te maken tussen hardware area/resources, prestatie, stroom/energie verbruik en betrouwbaarheid. De processor is als open-source beschikbaar.

Prepositions

1. All hardware and software should be reconfigurable.
2. Hardwired multiported memories are a must for the efficient implementation of parallel hardware in FPGA.
3. Software comes from heaven when you have good hardware. (Ken Olsen)
4. The distinction between VLIW and superscalar processors is vanishing.
5. Normal life starts after the PhD study.
6. A good idea means nothing by itself; a good implementation is equally important.
7. Will is more important than competence to achieve something.
8. You are not doing research when you know what you are doing.
9. “Freedom of expression” should not be considered as unlimited.
10. Without improving the primary education system in Pakistan, spending billions in higher education is of little use.
11. Tolerance is the only thing the Pakistani nation needs nowadays.
12. A good way to learn new things is to be unlucky.

These propositions are regarded as opposable and defendable, and have been approved as such by the promotor Prof. dr. K.L.M. Bertels.

Stellingen

1. Alle hardware en software zou herconfigureerbaar moeten zijn.
2. Hardwired multiported geheugens zijn een vereiste voor het efficiënt implementeren van parallel hardware op FPGA.
3. Software komt van de hemel wanneer je goede hardware hebt. (Ken Olsen)
4. Het verschil tussen VLIW en superscalar processoren is aan het verdwijnen.
5. Het normale leven starts na de PhD studie.
6. Een goede idee betekent opzichzelfstaand niets, een goede implementatie is even belangrijk.
7. Wil hebben is belangrijker dan competentie om iets te bereiken.
8. Je bent geen onderzoek aan het doen als je weet wat je aan het doen bent.
9. "Vrijheid van meningsuiting" moet niet als onbeperkt worden beschouwd.
10. Zonder het verbeteren van het primair onderwijs in Pakistan is het uitgeven van miljarden in hoger onderwijs van weinig nut.
11. Vandaag de dag is tolerantie het enige dat de Pakistaanse natie nodig heeft.
12. Een goede manier om nieuwe dingen te leren is om een pechvogel te zijn.

Deze stellingen worden opponeerbaar en verdedigbaar geacht en zijn als zodanig goedgekeurd door de promotor Prof. dr. K.L.M. Bertels.

Acknowledgments

Here comes the end to my formal student life. That was a fun by itself. Finding this opportunity, I would like to express my gratitude to all those who contributed directly or indirectly to the work reported in this thesis.

First of all, I would like to thank my supervisor Stephan Wong who provided me the opportunity to perform research in the Computer Engineering (CE) Lab. His guidance and consistent involvement in all phases of my PhD research project is truly remarkable. We had many brainstorming sessions and long technical meetings that helped me a lot in my work. Special thanks go to the promotor of my thesis Koen Bertels. He always offered his help through out my stay at the university. I am also very grateful to all the faculty members of CE who provided me help and guidance from time to time.

The members of my PhD committee also deserve appreciation. I thank them for devoting some of their time to read my thesis, providing me their valuable comments, and traveling to Delft for the public defense of this dissertation. Special thanks go to Luigi Carro for his discussion and collaboration through out the research project.

The Higher Education Commission (HEC) of Pakistan partially sponsored the research reported in this thesis. I would like to thank all the staff at HEC who was always available whenever I needed their help. I am also very grateful to my former boss Saif shb and colleagues Atif shb, Sajid shb and Yaseen shb for their encouragement and help. In the Netherlands, the NUFFIC and CICAT deserve appreciation. I am very grateful to Loes, Charlene and all other staff at NUFFIC for providing their support. Franca from CICAT deserves special thanks for taking care of my visa related and financial issues.

Appreciation goes to Roel Seedorf, Anthony, Arash, Roel Meuws, Catalin, Dimitri, Yi Lu, Thomas, Zaidi and all other colleagues at the CE Lab for the long discussions we had. I thank them for providing a friendly and research conducive environment. Special thanks go to Motaqi for translating the propositions and summary of this thesis. Mota! You are a great person, always

ready for help. High appreciation goes to the technical and administrative support provided by Bert, Erik, Eef, Lidwina and Monique. I really enjoyed all the social events that are an integral part of life at the CE Lab. Thanks to the organizing members.

I was lucky to have so many Pakistanis around during my stay in Delft. With them I never felt away from home. These include Mehfooz, Hamayun, Laiq, Mazhar, Nadeem, Husnul Amin, Yahya, Hamid, Usama, Umer Ijaz, Saleem, Zubair, Tariq, Hisham, Rafi, Sharif, Umer Naeem, Adeel, Hanan, Fahim, Shah, Rajab, Tabish and all other whose names are not mentioned. Special thanks go to Cheema, Atif, Bilal, Dev, Faisal Kareem, Sandilo, Faisal and Seyab for my early day's help out. We had a very good company living in Poptahof, gossiping and playing cards the whole nights. Great appreciation goes to Imran for his delicious cooking. The social events and get together that we had will always be remembered. Special thanks go to all friends who arrange to play cricket every weekend.

Back in Pakistan, there were many people who encouraged me and prayed for my success. Thank you all. My family deserves great appreciation. Although my mother has been very sick in my absence, but she always prayed for me. I wish she get well soon. My brothers and sisters, cousins and all other family members have supported me well in their capacity. Whenever I spoke to them on phone, I felt myself more energetic. Uneeza, Abru, Manahil, Dua, Fateh, Momin, Aman and Mateen! You kept me alive whenever I spoke to you and visited you. Bushra deserves very special recognition for being very supportive. After she came in my life, fortune has favored me. She always stood with me and that is why I have never let her feel alone. I thank her for her understanding and cooperation whenever I got busy and had little time for her. We have had a very memorable time together. My love and prays are always for her. I would not forget to mention my father here. He is the source of all my inspirations. It was his vision that today I have completed my PhD thesis.

Finally, I would offer my thanks to the Dutch Society in general. People are very friendly and supportive. I really enjoyed my stay in the free and open environment. Going for long biking trips was a fun. Staying at Delft was a fantastic time. I will always remember it.

Fakhar Anjam

Delft,

Eid-ul-fitr, August 08, 2013

Table of Contents

List of Figures

List of Tables

List of Acronyms and Symbols

ALU	Arithmetic Logic Unit
ASIC	Application Specific Integrated Circuit
BRAM	Block Random Access Memory
DLP	Data Level Parallelism
FF	Flip-Flop
FPGA	Field Programmable Gate Array
FSM	Finite State Machine
FU	Functional Unit
GPP	General Purpose Processor
HDL	Hardware Description Language
ILP	Instruction Level Parallelism
IPC	Instruction Per Cycle
ISA	Instruction Set Architecture
ISE	Integrated Software Environment
ISR	Interrupt Service Routine
LUT	Look-Up Table
MT	Multi-threading
PE	Processing Element
RFI	Return From Interrupt
RISC	Reduced Instruction Set Computer
SEU	Single Event Upset
SMT	Simultaneous Multi-threading
SIMD	Single Instruction Multiple Data
TLP	Task Level Parallelism
TMR	Tripple Modular Redundancy
UART	Universal Asynchronous Receiver Transmitter
VEX	VLIW Example
VHDL	VHSIC Hardware Description Language
VHSIC	Very High Speed Integrated Circuits
VLIW	Very Long Instruction Word

1 Introduction

In the current-day world, fixed processors (which cannot change their hardware functionality after fabrication) are the mainstream and are made programmable in order to adapt to a large number of applications. As a consequence, they perform adequately over a wide range of applications, but not efficiently in terms of performance or energy consumption. Application-specific integrated circuits (ASICs) are designed according to the specific requirements of an application, therefore, they are the most efficient implementation and consume very low power. The major problem with an ASIC is that it cannot be adapted for a different application and has a longer and quite expensive development cycle. Reconfigurable hardware, such as field-programmable gate arrays (FPGA) can modify their hardware structure. Hence, efficient systems can be implemented in FPGAs due to the flexibility they offer. In general, FPGAs are programmed using hardware description languages (HDLs), which require the every-day programmers to have intricate knowledge of hardware. Even the use of language translation tools may require rewriting of code leading to longer development time. Now given reconfigurable hardware, can we combine the flexibility of programmable processors with the reconfigurability of FPGAs? Can we design reconfigurable programmable processors that can adapt their functionality to the applications? Can we make designs that can even adapt themselves during run-time? In this dissertation, we try to answer such questions.

The remainder of the chapter is organized as follows. Section 1.1 presents some basic concepts required to understand the questions raised in the dissertation. The scope of the dissertation is discussed in Section 1.2. Some open research questions are formulated in Section 1.3, which are later on answered in the dissertation. Section 1.4 presents the steps that are followed in order to answer the research questions raised in the chapter. Finally, Section 1.5 provides the organization and structure of the dissertation.

1.1 Background

In this section, we provide some background knowledge on programmable processors. We present different processor design architectures, describe different forms of processor parallelism, and then discuss processor architectures that exploit instruction-level parallelism (ILP). Later on, we highlight the benefits of combining programmability and reconfigurability in a single hardware.

1.1.1 General-purpose and Embedded Processors

General-purpose processors (GPPs) are designed without considering the requirements of a specific application or task; rather they are designed to perform adequately over a large number of application domains. Their instruction set is general-purpose rather than specialized for a particular task, therefore, they are not very efficient in terms of performance, power, cost, area, etc., across some or all application domains. In addition, they have support for many different kinds of peripherals. Different software can be put on them and hence can be used for different purposes. Mostly, they can be found in today's PCs, tablets, and servers etc.

Embedded systems include a number of components, where each smaller component provides a service to the large embedding system. An Embedded processor (EP) could be one of the components of the embedded system. EPs are utilized in a large number of chips found in, for example, cellular phones, TVs, automotives, biomedical equipments, game consoles, microwaves, and in many other consumer electronic appliances. Generally, these processors are smaller in size and are customized for a particular application or a domain of applications. They can perform the specific tasks more efficiently compared to a general-purpose processor. The different requirements for embedded processing which are equally important for general-purpose processing could be performance, power consumption, area, cost, cooling system, reliability, dependability, etc.

1.1.2 Processor Design Architectures

A processor (GPP or EP) can be designed according to different architectures/philosophies such as the reduced instruction set computer (RISC), complex instruction set computer (CISC), very long instruction word (VLIW) or superscalar. A RISC processor has simple and fundamental operations set that

operates on simple data kept in registers. The only memory-related operations are load and store operations. All the operations can be executed in a single clock cycle. Code size is large and the compiler has more work to do. Normally, RISC processors can issue a single operation every clock cycle. CISC uses complex operations in addition to the simple ones. A complex operation could be a new operation or may be a combination of few fundamental operations. A *string move* operation, in which a stream of characters stored at a location in memory is moved to another location, is an example of a CISC operation. The execution of an operation may take more than one clock cycles. The assembly code resembles to the high-level code. The compiler has less work to do and the code size is smaller compared to a RISC processor. The *Intel x86* is an example of the CISC architecture.

VLIW and superscalar processors include multiple parallel execution units to exploit instruction level parallelism (ILP). Both these processors can issue multiple operations in a single clock cycle to increase the performance. The major difference between a VLIW and a superscalar processor is that a VLIW processor relies on a compiler to exploit ILP, while a superscalar processor relies on run-time hardware to exploit ILP. Generally, both of these processors have RISC-like instruction set, but superscalar processors with complex instruction set have also been developed. Examples include the in-order superscalar original *Pentium* and the out-of-order superscalar *Cyrix 6x86*. Table 1.1 presents some characteristics of ASICs, RISC (single-issue), CISC, VLIW, and superscalar processors. Each design philosophy has its own advantages and disadvantages.

Table 1.1: Relative characteristics of ASICs, RISC (single-issue), CISC, VLIW, and Superscalar processors.

Type	ASICs	RISC	CISC	VLIW	Superscalar
Hardware Complexity	Medium/High	Medium	Higher	High	Highest
Hardware Area	Small/Medium	Medium	High	High	Highest
Power Consumption	Small	Medium	Medium/High	High	Highest
Performance	Highest	Small/Medium	Medium/High	High	High
Compiler Complexity	No Compiler	Medium	Small/Medium	Highest	Medium/High
Programmable	No	Yes	Yes	Yes	Yes
Code-compatible	No	Yes	Yes	No/Small	Yes

1.1.3 Different Forms of Processor Parallelism

In the domain of processors, parallelism refers to the opportunities in a program to find independent operations and perform them separately in parallel instead of performing them sequentially. There are different forms of processor parallelism which can be thought of as independent of each other. In this section, we briefly discuss the most widely used among them.

ILP: Instruction level parallelism refers to the existence of independent operations in a program which can be executed together in a single clock cycle. Finding some independent operations in a program or a stream of operations is the job of a compiler in case of a VLIW processor or run-time control hardware in case of a superscalar processor. ILP can be combined with any other type of parallelism to further enhance the performance.

DLP: Data Level Parallelism refers to distributing the data across different parallel computing nodes and executing them in parallel. In this case multiple processing nodes receive a part of the total data and they all execute the same operation on this data. The individual results are then finally combined into a single result. Single instruction multiple data (SIMD) is a form of DLP. SIMD operations operate on the standard registers, but treat them as smaller sub-registers. For example, four 8-bit operations can be performed in a single 32-bit operation in 1 clock cycle which would otherwise require 4 clock cycles.

TLP: Task Level Parallelism refers to executing multiple threads of an application on the different processors of a multiprocessor system. A multiprocessor system consists of multiple similar (homogeneous) or different (heterogeneous) processing elements. A program is split into multiple, relatively independent small sub-programs which are executed at the same time on different processors to achieve parallelism. The individual processors may or may not be able to exploit ILP. Programming and compiling for multiprocessors are becoming very complex due to the large number of cores available in today's multiprocessor systems.

MT: Multi-threading refers to a technique where different programs or parts of a program (called threads) are executed one by one on a single hardware to show progress on multiple programs or parts of programs. Threads are very light-weight (in terms of state) and pose less serious problems when they are switched. Different policies can be implemented for the sharing the single hardware, such as round-robin, priority-based, FIFO-based, etc. The shared hardware may also be able to exploit ILP in the individual threads.

SMT: Simultaneous Multi-threading is a special type of multi-threading available in the superscalar processors. A superscalar processor which does not have support for SMT can issue multiple instructions from a single thread every clock cycle. In case of the SMT, the superscalar processor can issue instructions from multiple threads every clock cycle, thus exploiting parallelism available across multiple threads. An example of a processor system which utilizes the SMT technique is graphic processing unit (GPU).

1.1.4 Architectures to Exploit ILP

VLIW and superscalar processors can be used to increase the performance beyond normal RISC architectures. While RISC architectures only take advantage of temporal parallelism (by using pipelining), VLIW and superscalar architectures can additionally take advantage of the spatial parallelism by using multiple functional units (FUs) to execute several operations simultaneously. ILP is determined by considering data dependence in a program and resource availability in hardware. In a superscalar processor, a special control hardware determines the data dependence and resource availability at run-time and then enables the dynamic scheduling of operations. On the other hand, for a VLIW processor, a compiler determines the data dependence and resource availability and statically schedules the operations. In a superscalar processor, the number of issued operations is determined dynamically by the hardware, while the number of issued operations in a VLIW processor is determined statically by the compiler. The window of execution is limited in a superscalar processor which limits the capacity to detect the potentially parallel operations. In case of a VLIW processor, the problem of limited size of execution window does not exist. The compiler of a VLIW processor can potentially analyze the whole program in order to detect parallel operations, hence, increasing the opportunities for finding parallelism. Compared to a VLIW processor, the hardware of a superscalar processor is very complex, larger in size, consumes more power, requires larger design efforts, and hence, becomes costly. According to [5], for the same technology and issue-width, the scheduling logic of a superscalar processor alone consumes more power than the entire VLIW processor. That is why a superscalar processor is less attractive for small embedded applications which require small and energy efficient devices. The hardware of a VLIW processor is relatively simple, and can be easily and quickly adapted from product to product at the expense of a complex compiler.

VLIW processors are designed such that the hardware details are more exposed to the compiler and ILP is made visible in the machine-level program.

ILP cannot be seen in the program that is offered to a superscalar processor; rather the hardware can arrange parallelism at run-time even though it is not exposed in the code itself. One of the advantages of a superscalar processor is that a compiled code for a single-issue scalar RISC processor can be executed on a superscalar processor with the same instruction set architecture (ISA). Hence, different superscalar implementations of the same ISA are object-code compatible. That is why superscalar processors are mostly utilized for general-purpose desktops and servers. Because the ILP is exposed in the program itself, to execute the same application on a VLIW processor, the original source code has to be recompiled for a new implementation/organization of the processor with the same ISA. Table 1.2 presents the major differences between a superscalar and a VLIW processor as described in [4].

1.1.5 Programmability and Reconfigurability Together

ASICs are designed to match exactly the requirements of the target applications. They have the highest-level of performance and consume very low power. When an application changes, for example, a new standard or protocol appears, or certain features need to be enhanced, an ASIC has to be redesigned for the new application. Normally, the development cycle is very long. Few tape-outs are required in order to fully test the complete application and all its requirements, thereby, increasing the development time and cost.

Type	Superscalar	VLIW
Instruction Stream	Instructions are issued from a sequential stream of scalar operations	Instructions are issued from a sequential stream of multiple operations
Instruction Issue and Scheduling	Issued instructions are dynamically scheduled by the hardware	Issued instructions are statically scheduled by the compiler
Issue Width	The hardware determines the number of issued instructions dynamically	The compiler determines the number of issued instructions statically
Instruction Ordering	Dynamic scheduling allows in-order and out-of-order issue	Static scheduling allows only in-order issue

Table 1.2: Differences between superscalar and VLIW processors [4].

Programmability is an important feature and it enhances the productivity of a processing element. It is also referred to as flexibility, i.e., how flexible a processing element is to adapt to a new application. Processors, whether general-purpose or embedded, are made programmable in order to provide maximum flexibility. A processor is designed with a basic instruction set, which it needs to support in hardware. Mostly, programmable processors are made fixed and cannot change their organizations after fabrication. A high-level compiler translates an application written in a high-level language (such as C) to the machine language of a processor. Hence, when an application changes, it is only a matter of compiling the new application and the hardware remains the same. This avoids the required lengthy development cycles and high costs. The major deficiencies of programmable processors include lower performance and higher power consumption compared to a dedicated ASIC.

FPGAs provide design-time as well as run-time configurability. They need to be programmed in HDLs such as VHDL or Verilog. Any kind of digital processing system can be quickly implemented with FPGAs. Initially, FPGAs were small in area/size, slow in speed, and mostly used for prototyping. With the advancement in technology, FPGAs have improved both in area and speed and have become very cheap. Modern FPGAs provide mechanisms to dynamically reconfigure some portions while others are still operational. Compared to ASICs, FPGA-based designs require very short development time, hence, minimizing the overall cost. Unlike ASICs, the development of FPGA-based designs can be immediately started, quickly implemented and shipped to the users. They can be updated in the field by downloading a new bitstream. Feedback from the early design shipments can be used to optimize the final product.

FPGA development requires the knowledge of digital circuits and somewhat low-level HDLs. Most of the high-level language developers/programmers do not have the knowledge of hardware and HDLs. Hence, it is difficult for these developers to design for FPGAs. Nowadays, different language conversion tools are available which convert programs written in a subset of a higher-level language to the HDLs. For example, Handel-C [6] is a subset of C language and the Celoxica DK design tools [7] can convert a program written in Handel-C to a VHDL description, which can then be synthesized for an FPGA or ASIC. The problem with Handel-C type languages is that they are not exactly the same as their higher-level language counter-parts. Hence, programs written in a high-level language first need to be converted manually to these languages, which increases the development time and cost. Additionally, these commercial tools are very costly.

Reconfigurability can also be used in conjunction with programmability. A programmable processor (e.g., a VLIW processor) can be implemented in an FPGA and made reconfigurable. VLIW processors have simple hardware design, consume low power, and can provide high performance. Different parameters of the processor such as issue-width, the number and type of execution units, the type and size of register file, degree of pipelining, size of instruction and data memories, cache parameters, fault tolerance, peripherals implementation, etc., can be made configurable and selectable at design-time. Hence, an optimized processor in terms of performance, area, power/energy consumption, and reliability can be quickly implemented for each application. Additionally, the processor can also be made run-time reconfigurable, where, after the implementation in hardware, certain parameters of the processor can be configured in order to target performance vs. power consumption trade-offs.

1.2 Scope

We foresee that combining programmability with reconfigurability by implementing a reconfigurable programmable VLIW processor in an FPGA will have several advantages such as high design flexibility and rapid application development. This approach allows applications to be developed in a high-level language, such as C, while at the same time, the processor organization can be adapted to the specific requirements of different applications both at design-time as well as at run-time. This dissertation proposes the scheme to combine programmability and reconfigurability which can be more precisely stated as:

We investigate an approach in (but not limited to) the embedded processor design that combines programmability and reconfigurability by implementing a programmable processor on a reconfigurable hardware, where the processor can reconfigure its organization for performance, area, power/energy consumption, and reliability trade-offs.

Consequently, our approach will distinguish itself from other approaches by the following points:

- *reconfigurable programmable VLIW processor:* In order to merge programmability with reconfigurability, we propose a programmable VLIW processor that can be configured/tuned at design-time and/or at run-time.

Statically-scheduled VLIW processors offer improved performance, reduced area footprint, and reduced power consumption compared to a superscalar processor.

- *parametrized design and toolchain:* The design of the proposed VLIW processor is very simple, made parametrized, and can be easily adapted for different applications. The parametrization of the design eliminates the lengthy manual development cycles or the costly C-to-VHDL tools. The availability of the free parametrized compiler-simulator toolchain [1] provides quick design space exploration and code generation.

- *design-time and run-time configurability:* With the proposed scheme, highly-optimized implementations can be generated for individual applications. Additionally, processors can be implemented which can adapt themselves at run-time for performance vs. power consumption trade-offs for different applications or different parts of an application.

- *use as stand-alone processor or co-processor:* The VLIW processor can be used as a stand-alone processor or can be coupled as a co-processor with another processing module (e.g., as in MOLEN paradigm [8]) for off-loading compute-intensive kernels.

- *configurable fault tolerance:* In order to mitigate single event upset (SEU) errors, configurable level of fault tolerance can be implemented. Fault tolerance can be included or excluded at design-time, and enabled or disabled at run-time.

The following assumptions further define the scope of the research described in this dissertation:

- We mainly focus on hardware design and its optimization for performance, hardware area, power/energy consumption, and reliability.

- Both the development toolchain and the processor design are made parametrized. The parametrized compiler can generate optimized code for our configurable VLIW processor. In this thesis, we only consider certain defined values for the different types of parameters.

- We consider FPGAs as the reconfigurable hardware in this thesis. In some cases (Chapter 5 and Chapter 6), we also present implementation results for a standard ASIC technology to show trends in power/energy consumption and hardware area.

- Support for partial reconfiguration is available in some modern FPGAs. We expect that the advances in technology will further simplify partial reconfigurable designs and reduce the reconfiguration times. Furthermore, the proposed design scheme does not necessarily depend on partial reconfiguration. Run-time reconfiguration can also be achieved with virtual reconfiguration schemes, i.e., by re-arranging (turning ON/OFF or multiplexing) the available resources at run-time.
- Custom or user-defined operations can be added to the hardware design and the compiler can generate binary code for them. Currently, the hardware design for user-defined operations has to be developed manually.

1.3 Open Questions

In this thesis, we present one possible approach to merge programmability with reconfigurability. The approach provides opportunity to trade-off between performance, area, power/energy consumption, and reliability for different applications, and hence, optimized solutions can be generated. For the successful merging, the following open questions have to be addressed:

- **Can FPGAs be programmed without knowing HDLs?**

As mentioned earlier, FPGA development requires the knowledge of digital circuits and HDLs. C-to-VHDL tools can be utilized to convert programs written in C to a VHDL description, which can then be implemented in FPGAs. The problems with these tools are that mostly, these are commercial, costly, and not very efficient. In most cases, code re-writing is needed when utilizing these tools, which again restricts their usability. Providing a parametrized/customizable design, where changing certain parameters results in different implementations is one way of avoiding the users to learn HDLs. In this thesis, we will investigate how efficient FPGA designs (programmable processors) can be implemented without knowing much about hardware design and HDLs.

- **Can we design flexible and reconfigurable processors which can adapt their functionality to the requirements of applications?**

Most of the available embedded programmable processors are made fixed in implementation and cannot change their hardware after fabrication. Many different applications exist which require different characteristics of the processing elements for efficient execution. A single fixed implementation cannot

perform well for all applications across different dimensions such as performance, power/energy consumption, area, code size, etc. In this thesis, we will investigate how we can design flexible processors which can be easily adapted to match the requirements of different applications.

- **Can we make these designs dynamic so that they can adapt themselves during run-time?**

With design-time configurability, optimized instance-specific implementations can be generated. However, when the number of applications to be executed is large or an application consists of several sub-applications, generating, implementing, and maintaining a large number of hardware configurations each tuned to a particular application becomes difficult or even impossible. In this thesis, we will investigate how we can create hardware designs that provide sufficient performance and reduced power/energy consumption for a large number of applications by reconfiguring their organizations at run-time to match the requirements of the applications.

- **Can we develop simple techniques for core-morphing and run-time code migration among different cores?**

Multi-core systems have multiple cores which can be used in different configurations. To exploit thread level parallelism, multiple threads of an application or multiple independent applications can be run on the individual cores. Some multi-core systems allow combining certain cores together to exploit ILP. Similarly, power can be reduced by turning off the un-used cores. In this thesis, we will investigate, how multiple cores in a multi-core processor can be combined/split at run-time and how a task running on a core can be migrated to a different core for performance improvement or power reduction.

- **What is the impact on performance and energy consumption when both the instruction cache and the processor's issue-width are simultaneously reconfigured?**

Memory system plays an important role in the performance and power consumption of a processor system. When the processor is reconfigured (e.g., issue-width is changed), the memory (caches) may also need to be reconfigured for improved performance or reduced power consumption. In this thesis, for a run-time adaptable processor, we will investigate the effect of simultaneous reconfiguration of the issue-width and instruction caches on the performance,

dynamic energy consumption, and energy-delay product (EDP) for different benchmark applications.

- **Are the implemented designs easily extendable?**

User-defined operations can increase the performance and/or reduce the power consumption of a processor. Before implementing a custom operation, a simple method of profiling and simulation to measure its performance is necessary. Because processors are implemented using HDLs, adding a custom operation requires the knowledge of hardware and HDLs. Providing a library of different design-time selectable custom operations and a simple methodology to implement additional custom operations increase the productivity. In this thesis, we will investigate how custom operations can be profiled and simulated at higher level (C language), added to a processor hardware design, and the binary code generated for them.

- **Can we implement fault tolerance techniques that are design-time as well as run-time configurable?**

In general, hardware-based fault tolerance techniques utilize additional hardware to detect and correct faults. This result in increased area, increased power consumption, and reduced performance. In order to optimize these characteristics, a processor should be able to include/exclude or enable/disable fault tolerance when required. In this thesis, we will investigate how we can develop hardware-based configurable fault tolerance techniques for our configurable processor for hardware area, performance, and power consumption trade-offs.

1.4 Methodology

In this section, we propose the different steps needed to combine programmability with reconfigurability to achieve a trade-off between hardware resources, performance, power/energy consumption, and reliability. These steps are:

- **Investigate and propose a parametrizable/customizable design of a programmable VLIW processor that can be configured at design-time to match the specific requirements of each application.** Implementing such a processor in a reconfigurable hardware, such as FPGAs means that applications can still be written in a high-level language, while taking advantages of the reconfigurability provided by an

FPGA. Multiple parameters and their implementation in different mechanisms allow a trade-off between hardware resources, performance, and power/energy consumption. Utilizing a parametrized/customizable design avoids to use any C-to-VHDL tool, provides high design flexibility and rapid application development.

- **Investigate and propose the parameters for the proposed VLIW processor that can be reconfigured at run-time to match the specific requirements of a running application.** Parameters such as issue-width, number and type of different FUs, register file size, etc., effect the performance, hardware area requirement, and power/energy consumption of an application. We will investigate and propose run-time techniques that allow running tasks to migrate from one core to another core in order to improve performance or power consumption characteristics at run-time. Additionally, we will investigate the effect of simultaneous reconfiguration of issue-width and instruction cache on the behavior of different applications.

- **Investigate and propose configurable fault tolerance techniques for the proposed VLIW processor in order to mitigate SEU errors.** We will investigate and propose hardware-based techniques which allow fault tolerance in a processor to be included/excluded at design-time and/or enabled/disabled at run-time in order to trade-off between hardware resources, performance, power consumption, and reliability.

1.5 Dissertation Organization

The remainder of this dissertation is organized in several chapters. Following, we present a brief summary of each chapter.

Chapter 2 – Background

Chapter 2 presents the background and motivations for the adaptable VLIW processor system needed for combining programmability and reconfigurability. The chapter highlights the VEX system which includes the VEX ISA, the VEX C compiler, and the VEX simulator. An earlier design of a VLIW processor is presented and its limitations are listed, which are later on, addressed in the thesis. Finally, the chapter presents some previous work related to the state-of-the-art in reconfigurable processors.

Chapter 3 – Design-time Configurable Processor

Chapter 3 presents the design and implementation of a parametrized and configurable VLIW processor based on the VEX ISA. The parameters include the processor's issue-width, the type and number of different FUs, type and size of register files, etc. These parameters can be configured/customized at design-time before implementing the processor in hardware.

Chapter 4 – Run-time Reconfigurable Processor

When the characteristics of an application are not known at the design-time, efficient processor's organization may not be selected for it, resulting in reduced performance and/or increased power consumption. In Chapter 4, we extend the processor design presented in Chapter 3 to make it run-time reconfigurable in order to meet the requirements of the running application(s).

Chapter 5 – Configurable Fault Tolerance

Chapter 5 presents hardware-based configurable fault tolerance techniques for our configurable processor. At design-time, users can choose between the standard non fault-tolerant design, a fault-tolerant design where the fault tolerance is permanently enabled, and a fault-tolerant design where the fault tolerance can be enabled and disabled at run-time. These options enable a user to trade-off between hardware resources, performance, power consumption, and reliability characteristics.

Chapter 6 – Experimental Results

Chapter 6 evaluates the effectiveness of our (re)configurable processors presented in the previous chapters. The hardware area/resources and the critical path delay (maximum clock frequency) were evaluated in these chapters. In this chapter, different metrics such as, performance (execution cycles, IPC), power/energy consumption, and EDP are utilized for different configurations of the proposed processors and different benchmark applications.

Chapter 7 – Conclusions

Chapter 7 summarizes the work presented in this dissertation and describes the main contributions of the research. Finally, several open issues and future work directions are listed.

2
Background

*In **Chapter 1**, we discussed the advantages and disadvantages of VLIW and superscalar processors in detail. Both processors have multiple parallel execution units to exploit ILP. In case of a VLIW processor, a compiler is responsible to find independent operations in a program and issue them together in a single clock cycle. For a superscalar processor, hardware determines operation dependence and resource availability at run-time. Therefore, the design of a VLIW processor is simpler compared to that of a superscalar processor at the expense of a complex compiler. Because a superscalar processor requires larger die size and consumes more power, it is not suitable for embedded systems which require area and power consumption as small as possible. Building a production-quality, high-performance optimizing VLIW compiler requires large effort, therefore, when considering the space of possible VLIW processor designs, it is always recommended to start with an available ISA and compiler, not the available hardware. Based on this, we started our research by utilizing one available compiler toolchain rather than building a new one. In this chapter, we provide some background information for the work carried out in this dissertation.*

The remainder of the chapter is organized as follows. Section 2.1 presents the motivations for an adaptable VLIW processor, discusses the VEX system, introduces an initial design of the ρ-VEX processor and lists its limitations. Some previous work related to the state-of-the-art in softcore and configurable/fixed processors is presented in Section 2.2. Finally, Section 2.3 concludes the chapter with a summary.

2.1 Adaptable VLIW Processor

An adaptable processor can adapt its organization according to the requirements of an application. This adaptability can be achieved at design-time, i.e., before an application starts execution or even at run-time when the application is running on the processor. In this thesis, we present an adaptable VLIW processor and highlight its benefits. The processor is based on the VEX ISA [4] and a toolchain [1] (C compiler and simulator) is freely available for architectural exploration and code generation. The processor combines both the programmability and reconfigurability to achieve high flexibility and high performance at the same time. It provides opportunities to compare performance, hardware resources, power/energy consumption, and reliability trade-offs.

2.1.1 Motivations

As discussed in Chapter 1, our proposal for combining programmability and reconfigurability requires an adaptable/reconfigurable VLIW processor. Instead of the other design philosophies mentioned in Section 1.1.2, we chose a VLIW processor as the starting point because of the following advantages:

- **increased performance:** Compared to a single-issue RISC processor, a VLIW processor can provide improved performance by exploiting ILP. While RISC architectures can only benefit from temporal parallelism by utilizing pipelining, VLIW architectures can additionally benefit from spatial parallelism by utilizing multiple FUs concurrently. A VLIW processor can potentially provide more performance compared to a same-issue superscalar processor due to the larger room for compiler optimizations.

- **reduced power consumption:** Because a superscalar processor utilizes complex control hardware for run-time scheduling of instructions, it consumes more power than a VLIW processor. According to an estimate by [5], the scheduling logic of a superscalar processor alone consumes more power than an entire VLIW processor of the same issue-width.

- **simple hardware:** The compiler takes care of all the dependencies and scheduling in case of a VLIW processor, while a run-time hardware does the same job for a superscalar processor. Therefore, the hardware of a VLIW processor is very simple and straight-forward at the expense of

a complex compiler, and hence, can achieve higher clock frequencies to further improve the performance.

- **availability of existing tools:** The compiler for a VLIW processor is very complex and requires significant efforts and time to develop from scratch. Fortunately, for the VEX ISA, a toolchain is freely available from HP. The VEX toolchain [1] includes a parametrized C compiler and simulator which can be used for design space exploration and code generation for different implementations of the VEX processor. Other open-source compilation frameworks such as Trimaran [9] etc., could also be easily adapted.

- **no need for language translations:** As stated earlier, designing for FPGAs requires the knowledge of hardware and HDLs. Most of the high-level language programmers do not have this knowledge. High-level-to-HDL translation tools are used, which place some restrictions on high-level languages and in most cases code rewriting is required when using such tools. With VLIW processor and its toolchain, programs can still be written in high-level languages (such as C), while taking advantages of the reconfigurability provided by an FPGA.

Apart from these basic advantages of a VLIW processor, following are the reconfigurability-specific benefits:

- **static reconfigurability:** Static reconfigurability means that the processor can be customized for a particular application before it is implemented in hardware. With the help of the simulator, processor parameters most suited for the targeted application(s) can be evaluated and determined. Hence, optimized designs can be implemented for each application.

- **dynamic reconfigurability:** Dynamic reconfigurability allows the processor to adapt its organization after it is implemented in hardware. When multiple applications need to be run, or the application's precise characteristics are not known at design-time, a single implementation cannot be optimized for them. In this case, the processor can be designed such that it can change some of its parameters (e.g., issue-width, number of registers and different execution units, cache size, etc.) at run-time to match the specific requirements of the running application(s).

The fixed nature of traditional VLIW architectures has certain intrinsic disadvantages which prevented them to become mainstream processors. These

disadvantages can be mitigated by implementing a VLIW processor on reconfigurable hardware. In the following, we highlight the most important problems that arise from the fixed design of a VLIW processor and their solutions:

- **different instruction lengths:** As stated earlier, different applications have different level of parallelism, and require different instruction word widths for efficient execution. A fixed processor may not exploit different level of parallelism very efficiently. This problem can be dealt with by implementing a parametrized and reconfigurable VLIW processor. Different instruction decoders can be instantiated/configured to provide different instruction word widths by either reconfiguring the issue-slots or sharing the unused issue-slots among other cores.
- **high number of NOPs:** A fixed VLIW processor may not meet the requirements of an application parallelism, and hence, a large number of NOPs may be scheduled. This scenario results in under-utilization of the available hardware resources. A parametrized/reconfigurable processor can adapt its organization/issue-slots to match the requirements of the application and avoid this under-utilization.
- **unavailable FUs per issue-slots:** NOPs are scheduled when issue-slots do not have the required FUs, thus increasing the under-utilization. With reconfigurable implementation, the required FUs can be added per application basis or even per phase of an application.
- **backward compatibility:** Code recompilation is needed when new versions of a VLIW processor is released. The reason could be a new organization of the FUs or a different set of added instructions. Backward compatibility can be relaxed by providing dedicated organizational features in the reconfigurable hardware for particular already-compiled code. Similarly, rarely used instructions can be instantiated when needed to support a legacy code.

Having stated how a parametrized and reconfigurable VLIW design can overcome the traditional shortcomings of a VLIW processor, in the following, we present the two most likely used scenarios for such a processor:

1. **stand-alone processor:** In this scenario, complete applications are compiled and they (or their threads) run on the VLIW processor. The processor can be configured at design-time to suit a particular application. Additionally, it can be reconfigured at run-time to suit multiple applications or multiple code portions of an application.

2. **application-specific co-processor:** In this scenario, only compute-intensive kernels are compiled to the VLIW processor while the remaining part of the application runs on another type of processing element. Hence, there is no need for code rewriting, complex tools such as C-to-VHDL translators, and manual design of accelerators, as in the case of a MOLEN processor [8].

2.1.2 The VEX System

The VEX (VLIW Example) system is developed by Hewlett-Packard (HP). It includes three basic components: (1) the VEX ISA, (2) the VEX C compiler, and (3) the VEX simulation system. A VEX software toolchain including the compiler and simulator is made freely available by the HP [1].

The VEX Instruction Set Architecture The VEX ISA is a scalable and customizable 32-bit clustered VLIW ISA [4]. It is modeled on the ISA of HP/ST Lx (ST200) family of successful VLIW embedded processors [10]. The VEX ISA is scalable because different parameters of the processor such as the number of clusters, issue-width per cluster, the number and type of different FUs and their latencies, and the number of read/write ports and size of register file, etc., can be changed. The ISA is customizable because special-purpose instructions can be defined in a structured way. It includes many features for compiler flexibility and optimization.

The VEX C Compiler The VEX C compiler [1] is derived from the *Lx/ST200 C compiler*, which itself is derived from the *Multiflow C compiler* [11], and includes high-level optimization algorithms based on *trace scheduling* [12]. It has the robustness of an industrial compiler, has a command line interface and is available as closed source (binary form). Because the VEX ISA is scalable and customizable, the compiler also supports the scalability and customizability. A flexible machine model determines the target architecture, which is provided as input to the compiler in the form of *machine model configuration (fmm)* file. Hence, without the need to recompile the compiler, architectural exploration of the VEX ISA is possible with the compiler and simulator. To add a custom operation, the application code is annotated with pragmas. Different compiler pragmas and optimization options are available for performance improvement [4]. Applications can be compiled with *profiling flags*, and the *GNU gprof* can also be utilized to visualize the profiled data.

The VEX Simulation System The VEX simulator [1] is an architectural-level simulator that uses *compiled simulator* technology to achieve faster execution. With this simulator, C programs compiled for a VEX configuration can be simulated on a host workstation for performance analysis and architectural exploration. The VEX simulator first translates the VEX binaries to C, and then using the host C compiler generates a host executable. The simulator provides a set of POSIX-like *libc* and *libm* libraries (based on the GNU *newlib* libraries), a simple built-in cache simulator (level-1 cache only) and an application program interface (API) that enables user-defined memory interfaces and other plug-ins for modeling the memory system [4]. To model L1 instruction and data caches, a cache simulation library is provided, which can also be replaced by a user-defined library. After simulation, the simulator generates a log file with different statistics such as the number of execution cycles, stall cycles, total executed operations, IPC, total branches (taken and not taken), total memory accesses, total misses, total NOPs, etc.

2.1.3 The Initial Design of ρ-VEX VLIW Processor

The ρ-VEX is a 32-bit 4-issue softcore VLIW processor based on the VEX ISA [4]. The processor is implemented in VHDL language and has a Harvard architecture employing separate memories for instructions and data. It has two different versions, namely, the *non-pipelined* [13] [14] and the *pipelined* [15] versions. Both versions have the same general features such as the issue-width of 4 and the same number of FUs and register files. The processor has 1 *branch unit (CTRL)*, 1 *memory or load/store unit (MEM)*, 4 *arithmetic logic units (ALUs)*, 2 *multipliers (MULs)*, a 64×32-bit 4-write-8-read (4W8R) ports *general register file (GR)* and an 8×1-bit 4W4R ports *branch register file (BR)*. The BR register file is used to store branch conditions, predicate values, and the carries from arithmetic operations. In the non-pipelined design, a new instruction is fetched only when the previous instruction is completely executed and the results written back to the target register file. For the pipelined design, a new instruction can be fetched every clock cycle while the previous instructions are in-flight and even not yet completely executed. Hence, each unit/stage is active at every clock cycle and working towards the completion of different instructions, making the overall program execution faster.

The non-pipelined design has 4 stages called *fetch*, *decode*, *execute* and *writeback*, while the pipelined design consists of 5 stages called *fetch*, *decode*, *execute 0*, *execute 1* and *writeback*. For the non-pipelined design, each stage is implemented as a finite state machine (FSM). The fetch stage is responsible for

address generation and instruction fetching from the attached instruction memory. A fetched long instruction is passed on to the decode stage, which splits it into four 32-bit syllables and decodes them in parallel. The decode stage also fetches the required operands for the operations from register files. Branch and other control related operations are performed by the branch unit. The actual operations (ALU, MUL, and load/store) are performed in the execute stages depending upon the latency of the operations. A 32×32-bit MUL operation is performed by two 16×32-bit MUL operations and then adding the partial products. All write activities are performed in the writeback stage to ensure that all targets are written back at the same time. Different write targets are the GR and BR register files (both designs), as well as the data memory and the PC for the non-pipelined design. The 4-issue non-pipelined ρ-VEX processor is depicted in Figure 2.1. The VEX compiler is used to compile a C application and generate VEX binaries, which are then assembled by an assembler to generate VHDL instruction ROM and an initialized data memory.

Limitations The initial design of ρ-VEX processor suffers from the following limitations, which are addressed in this dissertation.

1. Different parameters for extensibility (such as issue-width, type of GR register file, etc.) are not explored/implemented. These parameters for evaluating the performance, hardware area, and power/energy consumption trade-offs.

2. The GR register file design requires considerable area (FPGA's configurable resources such as slices, LUTs, flip-flops) for its implementation.

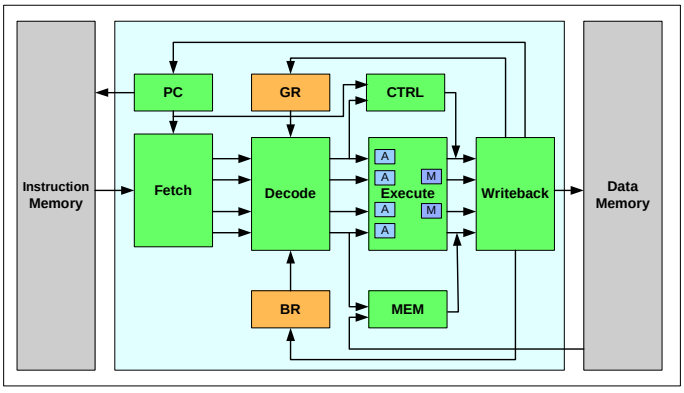


Figure 2.1: 4-issue non-pipelined ρ-VEX VLIW Processor.

Efficient register file (BRAM-based and run-time reconfigurable register file) designs are missing.

3. The execution units (ALUs, MULs) having considerable area are under-utilized in the non-pipelined design. This area may be shared among other instances of the processor to improve the under-utilization and power consumption.

4. The current design of the multiplier (MUL) unit requires large area (slices or hardwired DSP modules) when implemented in FPGAs. Area and performance optimized multiplier (MUL) design is missing.

5. Although both designs utilize multiple execution units but they are single-core in the control-flow. Multi-core systems can be implemented and different units may be shared to reduce area and improve performance by exploiting both fine-grained and coarse-grained parallelism.

6. There is no mechanism to provide run-time reconfiguration of the parameters such as issue-width, GR register file size, increasing/decreasing the number of different execution units etc. This is important to adapt the hardware to a running application and optimize performance and/or power/energy consumption at run-time.

7. There is no support for interrupts and exceptions handling. These are important building blocks on a processor and provide an advanced level of control to applications and operating systems.

8. There is no support for run-time task migration among the different cores in the ρ-VEX based multi-core system. This is important to trade-off between performance and power consumption at run-time. It can also be used for fault tolerance.

9. Cache reconfiguration analysis regarding different issue-width for performance and energy consumption is missing. This is important to determine specific configurations to optimize performance and/or power/energy consumption. The statistics can be provided to the run-time reconfiguration algorithms to optimize these parameters.

10. The opcode space is very tight due to the utilized instruction encoding scheme. There is hardly any available free opcode left. This means that user-defined or custom operations could not be added restricting the processor's extensibility.

11. A systematic way of adding a custom operation is missing. Because there is no free opcode available, it is not possible to add a custom instruction. Providing a simple methodology for implementing a custom operation will increase the productivity.

12. Any mechanism to enhance the reliability and dependability of the processor is missing. Apart from providing fault tolerance, the processor should be able to turn off the circuit in case the application does not require fault tolerance at some point in time. This can result in huge power savings at run-time.

2.2 Related Work

In this section, we highlight several approaches that have been proposed in literature for designing programmable processors that are configurable to some extent as well. Some processors (with multi-issue support) which target high performance are unfortunately fixed in nature and cannot be reconfigured. We discuss configurable softcores (RISC and VLIW) and some hard-wired cores (adaptable and fixed).

2.2.1 Configurable RISC Softcore Processors

Soft means the processor core is not fixed in silicon, rather available in a synthesizable form and can be implemented in any technology (FPGA or ASIC). For some softcore processors, certain parameters can be configured at design-time, and hence, can be easily adapted for different applications. These are single-issue cores and an issue-width wider than one is architecturally not supported, hence, restricting higher performance.

MicroBlaze: MicroBlaze [16] is a 32-bit RISC softcore processor from Xilinx Inc. Parameters such as 3-stage or 5-stage design, implementing hardwired multiplier and divider units, cache parameters, and connection to different peripherals can be configured at design-time utilizing the Xilinx Embedded Development Kit (EDK) software. The IBM CoreConnect [17] processor local bus (PLB) architecture is used for connecting peripherals to the MicroBlaze. A fully featured application development toolchain is available from Xilinx. The MicroBlaze is offered as closed-source, requires a license to use, and can only be implemented in Xilinx FPGAs.

Nios-II: Nios-II [18] is a 32-bit RISC softcore from Altera Inc., and has similar features like the MicroBlaze. Altera provides the Nios-II Embedded Design Suite (EDS) software development tools including the GNU C/C++ toolchain. Parameters such as implementing hardwired multiplier/divider units, cache parameters, and connection to different peripherals can be customized at design-time utilizing the EDS. The Nios-II is offered as closed-source, requires a license to use, and can only be implemented in Altera FPGAs.

LatticeMicro32: LatticeMico32 [19] is a 32-bit Harvard, RISC softcore microprocessor, freely available from Lattice Semiconductor Inc., with an open IP core licensing agreement. The processor provides the visibility, flexibility, and portability that can be expected in an open source hardware design. The Lattice Semiconductor provides software development tools (LatticeMicro System) and evaluation boards with FPGAs for developing systems with the processor. Different WISHBONE [20] compatible peripheral components can be integrated with the core in order to accelerate the development of microprocessor systems.

LEON: LEON [21] is a 32-bit synthesizable processor core based on the SPARC V8 architecture, and is managed by the Aeroflex Gaisler Inc. Caches can be configured for the 5-stage, 7-stage, and radiation-hardened designs. The LEON processor is distributed as part of the GRLIB IP library (an integrated set of reusable IP cores) designed for system-on-a-chip (SoC) development. The bus architecture used in the LEON processor is the AMBA AHB/APB [22]. Aeroflex Gaisler provides a complete development toolchain for the LEON project.

OpenRISC 1200: OpenRISC 1200 is a 32-bit, 5-stage open source scalar RISC softcore processor [23]. It has special units such as multiply-accumulate (MAC) unit and configurable caches. The processor is designed with emphasis on performance, simplicity, low power requirements, scalability and versatility. The processor supports WISHBONE SoC Interconnection Rev. B [20] compliant interface for connection to peripherals.

2.2.2 Configurable VLIW Softcore Processors

The processors mentioned in this section are VLIW processors having support for issue-width larger than one. In literature, very few VLIW softcore processors with complete toolchain can be found. In most cases the issue-width is fixed and the extensibility is not very comprehensive. The absence of complete toolchain restricted the usability of some designs. Because most of the

designs do not have a specific name in literature, we refer them by their inventor's/designer's names.

Spyder: Spyder [24] [25] is the first reported VLIW softcore processor found in existing literature. Spyder had three reconfigurable execution units. A compiler toolchain was available, which was used to decide about the configurations of the execution units. Custom configurations could also be added to the library base. The Spyder marked the beginning of more (reconfigurable) VLIW softcore processor designs. It did not evolve extensively because both the processor architecture as well as the compiler was designed from scratch. The designers had to work on the development and improvements of both the architecture and the toolchain which were time-consuming.

Brost VLIW: In [26], a customizable design of a VLIW processor is presented by Brost et al. Certain parameters of the processor architecture could be configured in a modular fashion. Algorithms are programmed in C as if they were to be executed on a DSP processor and compiled with a VLIW DSP compiler. The code is analyzed automatically, and an optimal DSP VHDL model with a variable instruction set is generated, which can be implemented in hardware. The DSP model utilized is the Texas Instruments (TI) TMS320C6201 [27], which is an 8-issue VLIW processor. The presented design is an instance-specific implementation of a DSP processor, and hence, does not represent a general VLIW processor system.

Lodi VLIW: A VLIW processor with a reconfigurable instruction set is presented in [28] by Lodi et al. The XiRisc, which is a 5-stage pipelined 2-issue VLIW processor, is tightly coupled with a pipelined run-time configurable datapath (PiCo gate array, or PiCoGA). Regular FUs perform typical DSP calculations such as 32-bit MAC, SIMD ALU operations, etc. The reconfigurable PiCoGA is utilized to extend the processor instruction set with application-specific multi-cycle instructions. The processor instruction set has been extended with two types of instructions; one to reconfigure the PiCoGA, and the second to execute the configured function. A GNU gcc toolchain is available for programming and benchmarking. The complete system is implemented in a 0.18-μm CMOS technology.

Jones VLIW: An FPGA-based design of a VLIW softcore processor is presented in [29] by Jones et al. The processor is based on an ISA that is binary-code compatible with the Altera Nios-II ISA [18]. The compilation scheme consists of Trimaran [9] as the front-end and the extended Nios-II as the back-end. An application is profiled and compute-intensive kernels are selected. The kernels are translated to VHDL and the remaining code is compiled for

the VLIW processor. The kernels are implemented in hardware and attached in parallel with the regular FUs. Utilizing multiplexers, the register file ports are shared between the hardware kernels and the regular FUs. Due to the licensed Nios-II core and ISA, this VLIW design is not much flexible, and the parametric extensibility is not possible.

Grabbe VLIW: An instance specific VLIW processor for elliptic curve cryptography is presented in [30]. The processor can perform basic field operations in parallel as well as complex instructions needed for the specific application. In order for the instruction set to be easily modified or extended, the control path is microcoded. The modular datapath structure and the FPGA-optimized design facilitate the adaptation to various requirements of different applications. The presented design is a direct implementation of a specific application, and does not represent a general VLIW processor.

Koester VLIW: In [31], a hardware compilation flow to generate instance-specific VLIW cores is presented by Koester et al. The application is described in ANSI C and then translated to a VLIW-style hardware targeting ILP. The front-end consists of the VEX compiler [1] which translates a sequential C program to the VEX assembly. The back-end consists of the Celoxica DK Design Suite [7], which converts a Handel-C [6] description to a synthesizable VHDL description. The AS2HCC tool converts the code generated by the VEX compiler to the Handel-C description, which is translated to VHDL code by the DK Design Suite. Hence, instance-specific optimized VLIW cores can be generated. The disadvantages of the design are that it requires commercial C-to-VHDL tools, the generated architectures are instance-specific, and it does not represent a general VLIW processor.

Saghir VLIW: In [32], the architecture and micro-architecture of a customizable soft VLIW processor is presented by Saghir et al. The processor executes a basic set of integer operations that resemble MIPS R2000 instructions [33]. The datapath is configurable and can include `16`, `32`, or `64`-bit regular FUs as well as custom computational units (CCUs) to execute user-defined operations. The processor has three configurable distributed register files instead of a single unified multiported register file. Processor specifications and an assembly program are provided to a tool which generates the VHDL model. There is no compiler for any high-level language, and hence, applications have to be written in assembly which restricts the usability of the processor design.

EPIC: Based on the Explicitly Parallel Instruction Computing (EPIC) ISA [34], a design of a customizable `2`-stage pipelined VLIW processor is presented in [35]. The EPIC architecture can exploit ILP by issuing multiple op-

erations per clock cycle. Possible customizations include varying the number of issues, registers, and FUs; all of which are specified at compile-time. Development tools include a compiler and an assembler based on the Trimaran framework [9] and a commercial C-to-VHDL tool. The processor is described in Handel-C language [6], which is then translated to VHDL description by the Celoxica C-to-VHDL tool [7].

Seshasayanan VLIW: In [36], a design of a low-power 16-bit, 6-issue VLIW test processor supporting a small number of operations is presented by Seshasayanan et al. For 64-point pipelined fast Fourier transform (FFT), the processor employs four radix-4 processing elements (PEs) in each stage. Both floating-point and fixed-point operations are supported. The system has two modules; one is the VLIW processor and the other is a hybrid dynamic voltage scaling (DVS) module. The hybrid DVS module is used to quickly adjust the processor's operating voltage or frequency at run-time, while maintaining the minimum level of performance an application requires. In this way, the application energy efficiency can be maximized. Main limitations of the processor are the absence of a rich set of instructions and a compiler toolchain. Additionally, there is no parametric extensibility available for the processor.

ADRES: The ADRES (architecture for dynamically reconfigurable embedded system) [37] couples a VLIW processor with a coarse-grain-array (CGA) accelerator, through a shared central register-file. This architecture has many advantages such as improved performance, a simplified programming model, reduced communication costs, and substantial resource sharing. The reconfigurable FUs are used to accelerate certain compute-intensive kernels, while the VLIW processor is used to improve the performance of the remaining part of the code by exploiting ILP. ADRES is supported by the DRESC compiler [38], which includes an XML architecture template to describe the functionality of the CGA accelerator, and to define the communication topology, supported operation set, resource allocation, and timing of the target architecture. The compiler generates a machine code to be executed on the ADRES processor, a simulation file for cycle-accurate simulation, and a synthesizable VHDL file for hardware implementation.

XIMD: The variable instruction multiple data (XIMD) [39] architecture structurally resembles VLIW architecture and can dynamically partition its resources to support concurrent execution of multiple instruction streams. The number of streams can vary from cycle to cycle to best suit each portion of an application. When all the sequencers read from the same location in the instruction memory, a XIMD processor operates exactly like a VLIW processor.

It can exploit both the instruction level as well as data parallelism. Although the XIMD provides interesting features, the architecture did not evolve extensively because of the absence of a good compiler.

OptimoDE: The ARM OptimoDE [40] technology is a system for analyzing and generating optimized instance-specific architectures for high-performance embedded signal processing applications. The technology is licensable intellectual property and is offered with an associated tool environment, which can be used for configuration and customization of the datapath resources. It allows a user to customize instructions per application basis instead of using a standardized ISA. It analyzes an application source code and then finds and selects optimal configurations (issue-width, FUs, storage sizes, interconnect topology, etc.) of the architecture. Custom units if any have to be generated manually by the user, while the standard FUs are inserted by the tools.

Tensilica Xtensa LX4: The Tensilica's Xtensa LX4 [41] is a configurable and extensible processor template. The major difference between the OptimoDE and the Xtensa LX4 is that the former allows a user to fully customize the instructions, while the latter uses a standardized ISA as well as user-defined functions. The template can be configured as a multi-issue VLIW processor with user-selectable 5- or 7-stage pipeline depth. Using the provided tools, customized solutions can be generated with a wide range of options including DSP units, local memories, I/Os etc.

CLAW: The clustered length-adaptive word (CLAW) processor is an 8-issue, 4-cluster VLIW processor [5], where each cluster has two issues. The issue-width of the processor can be configured at design-time as well as at run-time. The clustered approach scales down the resources when the issue-width is increased. Instead of having a large global register file, each cluster has its own local register file. Special channels and instructions are provided for inter-cluster communication. The compiler is used to capture ILP at compile-time and provide hints for the processor to shutoff certain clusters to reduce unwanted power consumption at run-time. The shutoff is done by a software instruction with a small latency. The processor is implemented in Verilog. The parametric extensibility with machine model parameters is absent. A multi-cluster organization although efficient in resource scaling is in-efficient in performance compared to a single-cluster VLIW processor [10]. Inter-cluster communication channels increase the critical path and inter-cluster copy operations reduce the performance and increase the code size.

KAHRISMA: The KAHRISMA architecture [42] utilizes different coarse-grained and fine-grained FUs and a run-time adaptable inter-communication

network. An application is partitioned, different operations are selected and implemented in the FUs, and the independent code-sections are compiled to a RISC or a fixed-issue-width VLIW processor, and at run-time, connections can be adapted to configure these two different modes. Instruction- , data-, and thread-level parallelism can be exploited. Inter-cluster communication is required between different clusters, as there is no global shared register file. KAHRISMA enables out-of-order execution, due to which it utilizes extra hardware. It uses the dynamic operation execution model, i.e., all operations of one instruction need not be issued at the same time [43]. The KAHRISMA ISA is comparable to clustered-VLIW processors, but its micro-architecture is similar to superscalar architectures with dynamic scheduling but without a dispatcher. In contrast to a VLIW program, a KAHRISMA program has unit-assumed latencies (UAL), and the latencies are not exposed to the compiler.

MOVE32INT: Transport triggered architecture (TTA) [44] is a class of statically programmed ILP architectures, and is programmed by specifying data transports instead of operations. A program specifies only the data transports to be performed by the interconnection network and operations occur as "side-effect" of the transports. Operands enter FUs through ports and one of the ports acts as a trigger. An operation is executed, whenever data is moved to the trigger port. The architecture can be tailored by adding or removing resources (FUs, registers, interconnects, etc.) and is particularly suited for application-specific purposes. MOVE32INT [45] is an instance of the TTA architecture which is implemented in a 2.0μ CMOS Sea of Gates technology. It is a 32-bit pipelined processor running at 80 MHz, with several FUs operating concurrently. Up to four concurrent data transports per clock cycle are possible.

2.2.3 Fixed Hardwired VLIW Processors

This section presents some widely used industrial hardwired VLIW processors. The distinguishing factors among them include the number and type of FUs and register files, the way in which the global control flow is maintained, and the amount of on-chip memory and/or caches. These processors are fixed in nature and cannot change their organizations/architectures.

STMicroelectronics ST231: ST231 [46] is a 32-bit 4-issue VLIW processor from STMicroelectronics. The processor is a single cluster implementation of the Lx architecture [10], and is used in several successful consumer electronics products. It is a 5-stage pipelined integer VLIW processor with multiple FUs and a multiported register file. The processor has a 32 Kbyte direct mapped L1 instruction cache and a 32 Kbyte 4-way set associative L1 data cache. Due to

simple logic, the processor consumes very low power. A complete toolchain including C compiler, debugger, etc., is available for application development.

Philips Trimedia TM1000: TM1000 [47] is a 32-bit 5-issue VLIW processor specially designed for real-time multimedia processing. It has 27 FUs split over 5 issue-slots, a 16 Kbyte data cache, and a 32 Kbyte instruction cache. Operations requiring more than two inputs and producing more than one outputs are supported by combining two issue-slots together. The processor has a very rich instruction set and supports up to four 8-bit or two 16-bit partitioned operations. Programmers can specify such operations at high-level with specific library calls and the compiler takes care of the rest. A complete toolchain (compiler, debugger, etc.) is available for application development.

Fujitsu FR500: FR500 [48] is a 32-bit 4-issue VLIW processor from Fujitsu Limited. There are two integer units, two floating-point units, a general register file, and a floating-point register file. The floating-point unit also performs MAC operations with 40-bit accumulation and partitioned arithmetic operations on 16-bit data. Both the instruction and data caches for the processor are 16 Kbyte 4-way set associative.

Texas Instruments TMS320C6211: TMS320C6211 [49] is a 32-bit 8-issue VLIW DSP architecture. The processor has 2 clusters, each with 4 FUs and a multiported register file. Each register file provides an additional read port for inter-cluster communication. It has a rich instruction set especially suited to target DSP algorithms. Up to two 16-bit partitioned operations are supported in some ALUs. To handle I/O data transfers, the processor features a programmable direct memory access (DMA) controller combined with two 32 Kbyte on-chip data memory blocks. A complete toolchain including compiler, debugger, etc., is available from the Texas Instruments.

Hitachi/Equator Technologies MAP1000: MAP1000 [50] is a 32-bit, 2-cluster 4-issue VLIW processor. Both clusters are similar, each having an integer ALU (IALU), an integer floating-point graphics ALU (IFGALU), 64 general registers, 16 predicate registers, and a pair of 128-bit registers. The IFGALU can perform advanced operations such as 64-bit partitioned operations, sigma operations on 128-bit registers, various formatting operations, and floating-point operations including division and square root. The processor has an on-chip programmable DMA controller, a 16 Kbyte 4-way set-associative data cache, a 16 Kbyte 2-way set-associative instruction cache. The data cache can also be used as on-chip memory. A complete toolchain is available for application development.

Transmeta's Crusoe TM5400: TM5400 [51] is the only known VLIW processor targeted to be used in general-purpose PCs and workstations. The 32-bit 4-issue processor when used in conjunction with the Transmeta's x86 code morphing software, provides x86-compatible software execution utilizing a technique called dynamic binary code translation. The processor together with the code morphing software can execute all standard x86-compatible operating systems and applications, including the Microsoft Windows and Linux. The processor has a 64 Kbyte 16-way set-associative L1 data cache, a 64 Kbyte 8-way set associative L1 instruction cache, a 256 Kbyte L2 cache, and a PCI port. The processor's control logic is kept simple and instruction scheduling is controlled by the running software. The processor has a 7-stage integer pipeline and a 10-stage floating-point pipeline. It consumes very low power compared to superscalar processors.

2.2.4 Our Proposal

In this thesis, we present a adaptable softcore VLIW processor called ρ-VEX. The processor is based on the VEX ISA developed by HP. A toolchain (parametrized C compiler and cycle-accurate simulator) is publicly available from HP, which can be used for architectural exploration and code generation. The ρ-VEX processor can be customized and different parameters such as issue-width, number of FUs, register file size, etc., can be selected at design-time to match the specific requirements of an application. Different types of multiported register files, interrupt systems, and custom operations are evaluated and can be selected for implementation depending upon the available hardware resources/area. A design methodology is presented to implement the processor with any required functionalities. Custom operations can be easily added to the hardware design and the compiler can generate binary code for them. Additional to the static features, the ρ-VEX processor is run-time reconfigurable. Parameters such as issue-width, number of FUs, register file size, etc., can be adapted at run-time to target performance vs. power consumption trade-offs. Multiple smaller issue-width cores can be combined/split at run-time to target ILP/TLP. Running tasks can be migrated from one core to another. The effect simultaneous reconfiguration of issue-width and instruction cache is evaluated to optimize the design. Hardware-based fault tolerance techniques are implemented for the processor which can be included/excluded at design-time and enabled/disable at run-time. All these options allow users to trade-off between hardware area, performance, power/energy consumption, and reliability. The ρ-VEX processor is publicly available as open-source.

2.3 Summary

After defining the goal of the thesis as to implement a programmable VLIW processor on a reconfigurable hardware for performance vs. power/energy consumption trade-offs in Chapter 1, this chapter highlighted the available techniques and tools for programmable and configurable processors. The chapter discussed the motivations behind an adaptable VLIW processor and the VEX system which includes the VEX ISA, the VEX C compiler, and the VEX simulator. An initial design of the ρ-VEX VLIW processor was presented and its limitations listed, which are later on, addressed in the thesis. The chapter ends with presenting state-of-the-art in configurable and/or programmable RISC and VLIW processors.

3

Design-time Configurable Processor

This chapter *presents an open-source design-time configurable softcore VLIW processor called ρ-VEX. The processor design is made parametrized and can be easily adapted for different applications before it is implemented in hardware. The parameters include the processor's issue-width, the type and number of different execution units and their latencies, the type and size of register files and the number of read/write ports, size of instruction and data memories, type of interrupt and exception systems, selection of default custom operations, datapath sharing, etc. The chapter presents a methodology to implement and utilize the processor. Applications have to be profiled and simulated to determine the suitable parameters for the processor. The parameters can be set in a configuration file before the processor is synthesized. Program binaries can be generated by utilizing the VEX compiler. Hence, trade-off between performance, hardware resource utilization, and power consumption can be easily made, and optimized solutions can be generated. Following are the contributions of the chapter:*

- *An open-source parametrized softcore VLIW processor is presented that can be configured before implemented in hardware. The synthesizable VHDL design is made parametrized, and hence, optimized solutions can be generated without using any C-to-VHDL tools. Applications can be developed in C, while taking advantages of the reconfigurability provided by an FPGA.*
- *Different types of multiported register files are implemented in order to optimize the hardware utilization and dynamic power consumption.*
- *Different types of interrupt handling systems are implemented to trade-off between hardware resources and interrupt response time.*
- *An optimized instruction encoding scheme is implemented to increase*

the available opcode space. A methodology to extend the instruction set of the processor is presented.

- *A datapath sharing mechanism is implemented to share the hardware resources between multiple instances of the processor.*

The remainder of the chapter is organized as follows. Section 3.1 presents the design-time configurable ρ-VEX processor and its implementation methodology. Different types of FPGA-based multiported register files for the ρ-VEX processor are presented in Section 3.2. Interrupt support for the ρ-VEX processor is discussed in Section 3.3. A new instruction encoding scheme to optimize the opcode space and remove certain compatibility issues is presented in Section 3.4. Section 3.5 provides a design methodology to extend the instruction set of the ρ-VEX processor. A datapath sharing technique to optimize the hardware resource utilization among multiple instances of the ρ-VEX processor is discussed in Section 3.6. Finally, the chapter is concluded by presenting a summary in Section 3.7.

3.1 Design-time Configurable ρ-VEX VLIW Processor

In this section, we present a parametrized, extensible, and design-time configurable softcore VLIW processor called the ρ-VEX. The processor is based on the VEX ISA [4] and has been implemented in VHDL. The technology-independent implementation allows the processor to be synthesized for any FPGA or ASIC technology. The processor is made parametrized and different parameters and constants can be provided in a configuration file before synthesizing the processor. The 5 stages/units of the processor are *fetch*, *decode*, *execute 0*, *execute 1*, and *writeback*. The fetch stage generates the instruction memory addresses and fetches a long instruction which is passed onto the decode stage. In the decode stage, operations are decoded in parallel and operands are fetched from the register files. Branch/control related operations are handled by the CTRL unit. The ALU, MUL, and load/store operations take place in the execute stages. The writeback stage ensures that all the write targets are written back together at the same time. The processor has a 64×32-bit multiported general-purpose (GR) register file and an 8×1-bit multiported branch (BR) register file. The BR registers are used to store branch conditions, predicate values, and carries from arithmetic operations. The number of the GR and BR registers can be decreased from 64 and 8, respectively,

at design-time. Different types of GR register files can be selected to optimized the hardware area/resources, performance, and power consumption. An interrupt handling system has been implemented for the processor in four different mechanisms offering trade-offs between hardware resources/area, performance, and power consumption. The processor can be interrupted, its state saved, and an interrupt service routine (ISR) can be executed. Exceptions can also be handled with the help of the interrupt system. An optimized instruction encoding scheme has been implemented to increase the opcode space which can be utilized for extending the instruction set of the processor. A VEX development toolchain [1] including a parametrized C compiler and a cycle-accurate simulator is free available from Hewlett-Packard that can be used for architectural exploration and code generation. Details about the VEX system can be found in Section 2.1.2. Following we discuss a methodology to generate an instance of the ρ-VEX processor and generate binary code for it.

Methodology to Generate a ρ-VEX VLIW Processor Figure 3.1 depicts the methodology/steps required to generate and utilize the ρ-VEX processor. The process starts with a C application which is simulated and profiled to generate different statistics. The VEX simulator reads a machine configuration file (created by the user/designer) describing the processor's configuration parameters such as the number of clusters, the issue-width per cluster, the type and number of different FUs (ALUs, MULs, MEMs, and custom units), latencies for different FUs, and the size of the GR and BR register files. It simulates the application on the configured/desired processor and then generates a detailed log file with different statistics, such as execution cycles, total executed operations, total branches (executed and not executed), stall cycles, memory operations (total accesses, total misses), instruction per cycle (IPC), and other function profiles. Custom or user-defined operations if any can also be specified at C language level, and the simulator is able to simulate them. The process can be repeated until an optimized processor configuration is obtained.

The optimized processor parameters are then utilized to generate a synthesizable VHDL description for the processor. We do not use any commercial tools for VHDL generation; rather we have a parametrized VHDL description for the ρ-VEX processor. The parameters passed to the compiler/simulator are described in a separate file called "*Processor Core Description*" as depicted in Figure 3.1. Based on these parameters the compiler generates assembly code when compiling an application. Another set of parameters is described in a different file called "*Processor Core Optimization*". These parameters include the types of GR register file as discussed in Section 3.2, the types of interrupts and

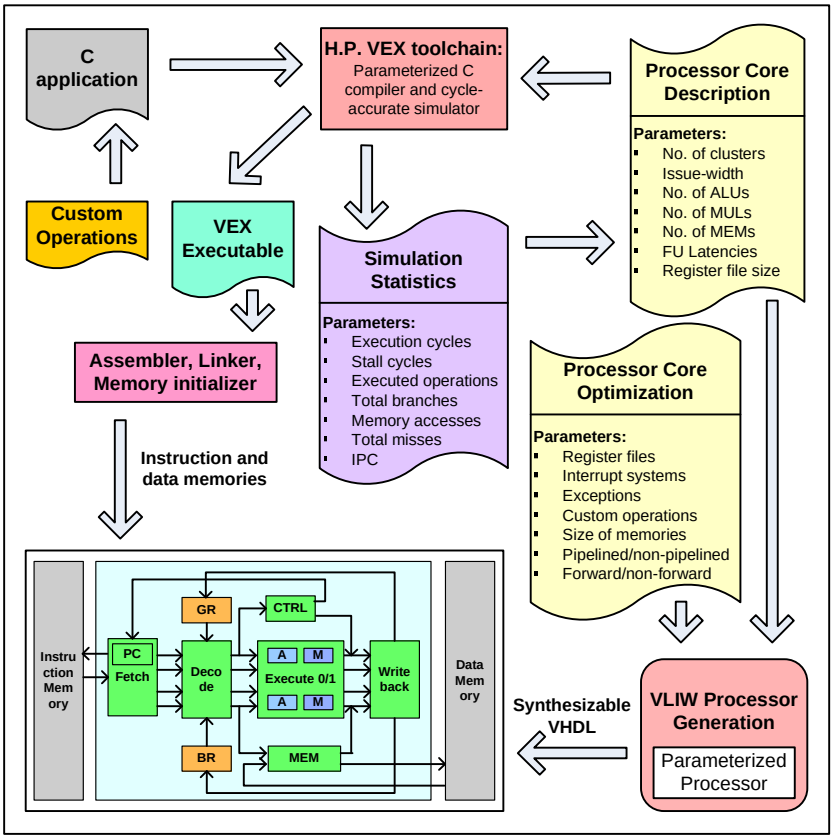


Figure 3.1: Methodology to generate an instance of the ρ-VEX processor.

exception systems as discussed in Section 3.3, types of default custom operations as discussed in Section 3.5, size of instruction and data memories, and the options for pipelined, non-pipelined, forwarding, and non-forwarding. These parameters can be used to optimize the selected processor core design. Both of the configuration files are included with the processor design files when the processor is being synthesized and implemented. Although, the processor design can be extended for any arbitrary configuration, the current version of ρ-VEX can be configured in issue-width to be 2, 4, and 8 only. Additionally, we consider only single-cluster implementations. Custom operations (other than those provided as default) require an additional step, and have to be added manually to the design. A simple methodology is provided which can be used to implement a custom operation for the ρ-VEX processor. The VEX compiler can generate code for custom operations (See Section 3.5). The processor can

be implemented in FPGAs or ASIC. The compiler is also provided with the same processor description file (*Processor Core Description*) and it generates code for the application to be executed on the ρ-VEX processor. The compiler generated code is passed through the low-level development tools (assembler, linker, etc.) to generate/initialize the instruction and data memories for the ρ-VEX processor. Hence, an optimized processor for an application can be implemented in an FPGA very quickly, shortening the development time and reducing the associated costs. In the following, we discuss different types of register files, different implementations of interrupt systems and the supported exceptions, methodology for adding custom operations, and datapath sharing for the ρ-VEX processor. All of these are design-time parameters for the ρ-VEX processor and can be used to trade-off between hardware resources/area, performance, and power consumption.

3.2 Multiported Register Files

The shared multiported register file is one of the most resource-consuming modules of a VLIW processor, and its resource requirement grows exponentially with increasing the issue-width. Since different issue-width processors require register files with different number of read and write ports, this section explores different register files, especially targeted for different types of FPGAs. As the current state-of-the-art FPGAs do not provide multiport memories, therefore, multiported register files are created with the FPGA's configurable resources such as look-up tables (LUTs), slices, and flip-flops (FFs) as well as the configurable resources plus the hardwired BRAMs. The 2-issue, 4-issue, and 8-issue ρ-VEX processors require multiported GR register files with 2-write-4-read (2W4R), 4W8R, and 8W16R ports, respectively. For this section, each processor has a 64×32-bit multiported GR register file, an 8×1-bit multiported BR register file, and a single MEM unit. The number of ALUs is same as the processor issue-width. The 2-issue, 4-issue, and 8-issue processors utilize 2, 2, and 4 MUL units, respectively. We utilized the Xilinx ISE release version 13.2 and the Virtex-4 *XC4VFX100-11FF1152* and Virtex-6 *XC6VLX240T-1FF1156* FPGAs for the implementation. The Virtex-4 and Virtex-6 FPGAs have 4-input and 6-input LUTs, respectively. Table 3.1 presents the details of the GR register files that we have implemented for our multi-issue ρ-VEX processors. These designs are evaluated in Section 3.2.3.

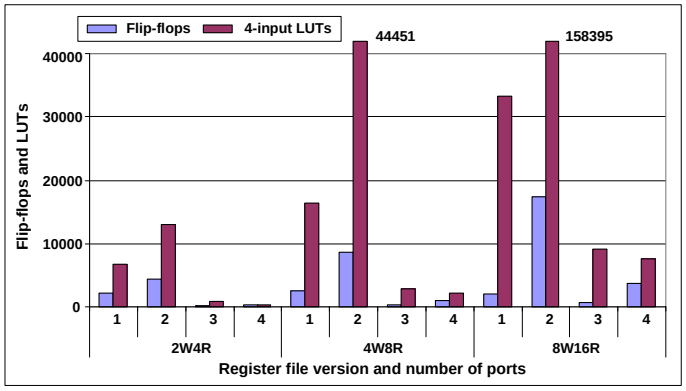
Table 3.1: Implementation types for GR register files

Version	Implementation detail
1	Straight-forward behavioral implementation requiring large combinational address decoders, and multiplexers/de-multiplexers utilizing LUTs + FFs.
2	Utilizes distributed memory (LUTRAMs) instead of BRAMs as in version 3, and LUTs + FFs.
3	Implemented utilizing banking and replication with BRAMs, and LUTs + FFs.
4	Similar to version 3, but running the internal ports of the BRAMs at twice high the frequency of the external ports.
5	Similar to version 3, but avoiding the use of the Direction table. The conflicts associated with the write ports are resolved by compile-time register renaming in the executable code.

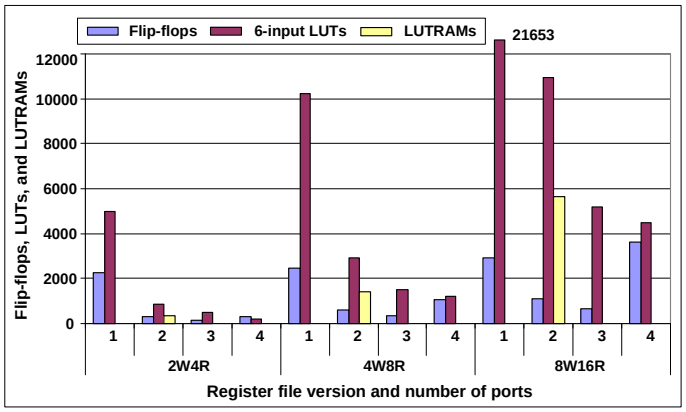
3.2.1 Register Files with FPGA's Configurable Resources

In this case, the GR register files for the ρ-VEX processors are implemented utilizing the FPGA's configurable LUTs, slices, and flip-flops, and the distributed memory called LUTRAMs. The size of all register files is configurable, but here we consider a 64×32-bit size. Each register can be accessed by each FU of the processor. In general, each of the regular FUs has 1 write port to access the register file for storing a data in a register and 2 read ports to read data from two different registers at the same clock cycle.

Register File Version 1 This is a simple and straight-forward design of the register file. Within the register file, there is a write address decoder, a read address decoder, and two read ports for each FU. It utilizes large combinational multiplexers/de-multiplexers and encoders/decoders to support multiple ports. These components are implemented utilizing LUTs with flip-flops as the storage elements. Due to its design, the distributed memory available in the FPGAs cannot be used, hence the 4-input LUTs usage becomes large compared to the 6-input LUTs. Figure 3.2 and Figure 3.3 depict the hardware implementation results for the different GR register files and multi-issue ρ-VEX processors, respectively, for the Virtex-4 and Virtex-6 FPGAs. It can be observed from the figures that less number of LUTs are required to implement the register files and the processors when the number of inputs on the LUTs increases.



(a) Virtex-4 FPGA (4-input LUTs)



(b) Virtex-6 FPGA (6-input LUTs)

Figure 3.2: Hardware results for different versions of the 64×32-bit GR register files with different ports. In addition to the mentioned resources, version 3 of the 2W4R, 4W8R, and 8W16R register files also utilize 8, 32, and 128 RAMB18s, respectively. Similarly, version 4 of the 2W4R, 4W8R, and 8W16R register files utilize 2, 8, and 32 RAMB18s, respectively.

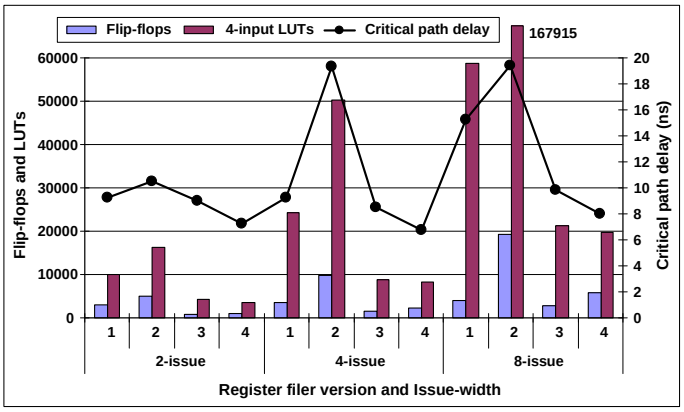
Register File Version 2 Here, banking and replication technique is used to implement the register file. A 64×32-bit base register file with 1W1R ports is implemented, which is then banked multiple times for the number of write ports and replicated inside each bank for the number of read ports. The product of the number of read and number of write ports on the actual register file determines the number of times the base register file is replicated. A small direction table keeps track of the write port number for each location of the register file which is implemented with the base register file. The table is

itself implemented utilizing FPGA's configurable resources and has the same number of ports and depth as the actual multiported register file. Figure 3.2 and Figure 3.3 depict the hardware implementation results for the different GR register files and multi-issue ρ-VEX processors, respectively, for the Virtex-4 and Virtex-6 FPGAs. As can be observed from the figures, compared to version 1, the version 2 register files utilize more flip-flops and LUTs in case of the Virtex-4 FPGAs. Considering the Virtex-6 FPGA, the version 2 register files utilize far less flip-flops and LUTs compared to the version 1 register files. The reason is that in case of the Virtex-6 FPGAs, the base 64×32-bit 1W1R register file is mapped to the distributed memory available in the FPGA. Some of the LUTs in almost all of the Xilinx FPGAs can be configured to be used as memory called the distributed memory or LUTRAMs. The base register file in our case has simple dual port (SDP) mode. In case of the Virtex-4 FPGAs, the available distributed memory can not be used as an SDP memory, therefore, the version 2 register files are implemented on standard LUTs in the Virtex-4 FPGA. In case of the Virtex-5 and latest FPGAs, the available distributed memory can be configured in the SDP mode, therefore, for the Virtex-6, the base register file in our case is mapped to the LUTRAMs, hence reducing the number of LUTs and flip-flips for the version 2 register files.

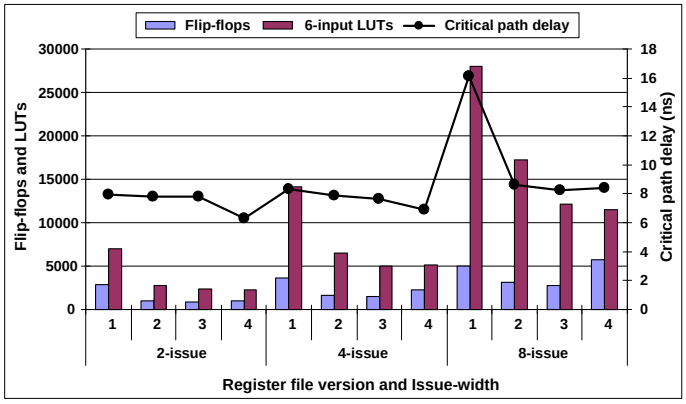
3.2.2 Register Files with FPGA's Embedded BRAMs

As stated in Section 3.2.1, increasing the number of ports on a register file increases the resources (LUTs and flip-flops) and reduces the frequency. To avoid this problem, multiported register files can be implemented utilizing the configurable resources plus the dual-ported BRAMs available in large numbers in the modern FPGAs. Following we discuss few such designs that we implemented for the ρ-VEX processors.

Register File Version 3 Register file version follows similar design to version 2. Instead of implementing the base register file with 1W1R ports with configurable resources, it is implemented with the fixed BRAMs. This design is also called as the single-pumped design, where the register file utilizes the same clock frequency as that of the processor. It utilizes BRAMs in order to reduce the utilization of configurable resources (flip-flops, LUTs, and slices) without using the register renaming technique as in the case of the version 5 register file. The implementation is based on the designs presented in [52] [53], which utilize a mechanism of port indirection. A table keeps track of the write port number for each location of the multiported memory that is implemented



(a) Virtex-4 FPGA (4-input LUTs)



(b) Virtex-6 FPGA (6-input LUTs)

Figure 3.3: Implementation results for multi-issue pipelined ρ-VEX processors with different versions of the GR register files. In addition to the mentioned resources, version 3 of the 2-issue, 4-issue, and 8-issue processors also utilize 8, 32, and 128 RAMB18s, respectively. Similarly, version 4 of the 2-issue, 4-issue, and 8-issue processors utilize 2, 8, and 32 RAMB18s, respectively. The 2-issue, 4-issue, and 8-issue processors also utilize 4, 4, and 8 DSPs modules, respectively.

utilizing BRAMs. The table is itself implemented utilizing FPGA's configurable resources and has the same number of ports and depth as the actual multiported memory. Figure 3.4 depicts the organization of a 4W8R ports register file. Here, each BRAM is configured as a 64×32-bit 1W1R ports memory block. For a register width of 32-bit, each BRAM can provide up to 512 such registers. To support multiple ports, the BRAMs are organized into banks and data is duplicated across various BRAMs within each bank. The number of

write ports defines the number of banks and the number of read ports defines the number of BRAMs per bank. Each bank holds a separate write port to update all the BRAMs in that bank, each of which can then be read by a separate read port. In this manner, different registers can be read from different BRAMs simultaneously. Multiplexers driven by the direction table outputs are utilized to provide access to the registers stored within each register bank. Figure 3.2 and Figure 3.3 depict the hardware implementation results for the different GR register files and multi-issue ρ-VEX processors, respectively, for the Virtex-4 and Virtex-6 FPGAs. From the figures, we can observe that the single-pumped BRAM-based register file design considerably reduces the flip-flops and LUTs utilization at the expense of BRAMs. The maximum clock frequency remains the same for both designs.

Register File Version 4 A single-pumped register file (version 3) runs at the same clock frequency as that of the processor. For W write and R read ports, a single-pumped multiported register file requires $W \times R$ BRAMs. In order to reduce the number of BRAMs for the multiported register files, we designed a register file where the internal ports run at twice higher frequency than the external ports. This means that the register file has to be run at twice the clock frequency of the processor. Basically, it is a multi-pumped design with multiplexing. A similar design for a quad-port memory is presented in [54]. The

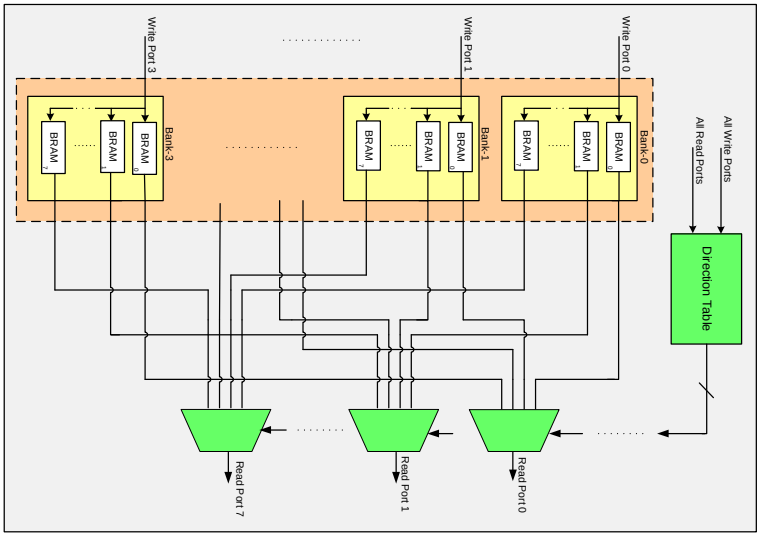


Figure 3.4: A single-pumped 4W8R ports BRAM-based register file.

double-pumped design reduces the number of BRAMs by a quarter compared to the single-pumped design. A double-pumped register file with W write and R read ports requires $1/4 \times W \times R$ BRAMs instead of $W \times R$ BRAMs. Implementation results for the different GR register files and multi-issue ρ-VEX processors are depicted in Figure 3.2 and Figure 3.3, respectively. From the figures, it can be observed that the double-pumped register file design reduces the required number of BRAMs by a quarter compared to the single-pumped register file design.

Register File Version 5 Multiported memories implemented utilizing BRAMs in banking and replication [55] have inherent conflicts associated with the write ports, and hence cannot be utilized as true multiported memories unless a technique called *register renaming* [56] [57] is applied, or additional hardware logic is utilized for port indirection. In this section, we present a register renaming technique that is applied at compile (assemble) time. The register file is implemented utilizing BRAMs based on the design presented in [55]. We implemented a register renaming technique to avoid write port conflicts and save considerable resources [58]. As a case study, we implemented a register renaming technique for a 4-issue ρ-VEX processor.

In order to support multiple ports, multiple BRAMs (with 1W1R ports) are organized into banks and data is duplicated across various BRAMs within each bank. For a register file with 4W8R ports, 32 BRAMs are distributed across 4 banks with 8 BRAMs per bank. Here, the number of write ports defines the number of banks and the number of read ports defines the number of BRAMs per bank. For a 32-bit register, each BRAM can provide up to 512 such registers. Because register banks hold mutually exclusive sets of registers, they can be updated independently. Each register bank holds a separate write port, which can write to the registers dedicated to that bank. In order to provide multiple read ports, multiple BRAMs are utilized within each register bank to store duplicate copies of the corresponding register subset.

A limitation of this design is that instructions cannot be scheduled to execute in parallel if they produce results in registers that belong to the same register bank. Hence, in any given instruction only one of the registers from a given bank can be written. We remove this limitation by applying *register renaming* technique at software level, after the code is generated by the compiler. The advantage is that it does not involve any hardware cost or compiler modification. All data dependencies are handled by the compiler and only register scattering needs to be done to avoid the write ports conflicts.

Each bank provides 64 registers that are logically renamed per bank, hence, the total number of registers is 256. This representation guarantees that we always have free registers to rename. Based on the application (data and control dependencies, and the available ILP), for a 4-issue ρ-VEX processor, the VEX compiler can generate an assembly code with 4, 3, 2, or 1 operation(s) per instruction. The register renaming ensures that within any instruction, no two operations should write to registers from a single register bank. The compiler utilizes 64 registers to generate the code, but our register file provides 256 registers with 64 registers per bank.

We developed a register renaming tool using C language. It takes the VEX assembly code as input and generates a register-renamed VEX assembly as output. Multiple passes are made in order to cover all possible conflicting conditions. The tool reads an instruction and parses its operations. It searches for the source and destination registers for all the operations of an instruction. It renames the destination registers for different operations in an instruction such that each operation could write to a separate register bank. A source register is renamed in a following instruction only, if that same register was renamed in an earlier instruction. While renaming the registers, the algorithm takes care of the different conflict conditions that result from the number and position of the different FUs and the compiler generated VLIW instructions.

Table 3.2 presents the hardware utilization for the 64×32-bit 4W8R ports register file with register renaming and the 4-issue ρ-VEX processor for the same Virtex-4 FPGA. Compared to the design presented in Section 3.2.1, this design requires considerably less number of flip-flops and LUTs at the expense of 32 BRAMs, while the frequency remains the same.

3.2.3 Evaluation of the Register File Designs

In the previous sections, we presented different implementation styles for multiported register files. These designs utilize specific hardware resources in FPGA. We selected two different families of the Xilinx FPGAs which

Table 3.2: Implementation results for 64×32-bit 4W8R ports register file with register renaming and 4-issue ρ-VEX VLIW processor.

Module	Flip-flops	LUTs	BRAMs	DSPs
Register file	521	477	32	0
Processor	3208	6514	32	4

are Virtex-4 (*XC4VFX100-11FF1152*) and Virtex-6 (*XC6VLX240T-1FF1156*). The Virtex-4 and Virtex-6 FPGAs have 4-input and 6-input LUTs, respectively. They have different number/quantity of flip-flops, LUTs, BRAMs, and distributed memory (LUTRAMs).

The designs (version, 3, 4, and 5) which utilize BRAMs for their implementation produce similar results for both the considered FPGAs. Version 4 reduces the number of BRAMs by 1/4, compared to the version 3 and version 5 designs, but it requires two different clock frequencies for its operation. The number of the required BRAMs remains the same for both the considered FPGAs. The number of required LUTs is different merely because of the size of the LUTs in the two families. When considering the version 1 and version 2 register files for the two types of FPGAs, there is a big difference in the hardware utilization. These designs are implemented utilizing Flips-flops and LUTs only and no hardwired BRAMs. In Virtex-6, the version 2 design can be efficiently mapped to LUTRAMs instead of implementing on the general LUTs. Therefore, the hardware utilization for this design is very small compared to the version 1 design. The critical path delay is also smaller. Because the LUTRAMs in the Virtex-4 family cannot be configured to be used in simple dual port (SDP) mode, both the version 1 and version 2 designs are mapped on to the general LUTs. Due to specific design, version 2 utilizes large number of LUTs compared to version 1 for the Virtex-4 FPGA. The critical path delay for version 2 is also longer compared to version 1 design.

Hence, given an FPGA, the designer has two multiple choices for efficient implementation of the register file and the ρ-VEX processor by considering the different versions that we presented. If there are more available BRAMs, the designer can choose to implement a BRAM-based design (version 3, 4 or 5). If multiple clock sources are available, version 4 is a better option as its requires less number of BRAMs. If there is limited number of BRAMs, the designer can choose to implement a LUT-based design (version 1 or 2). For the Virtex-5, Virtex-6, and other recent families, version 2 is a better option as it can be mapped more area efficiently on the available distributed memory. For the Virtex-4 and older families, version 1 is a better option due to its efficient implementation on the general LUTs.

3.3 Support for Interruptability

Certain critical tasks require that the processor should respond to them within a certain time limit. With an interrupt system, a processor can be interrupted,

its execution state saved, and a different task can be executed. The exception handling system ensures that the computed result is correct. The interrupt and exception handling systems are important building blocks on a processor for running an operating system on it. Features like multi-tasking, multi-threading, and task migration are facilitated by an interrupt system. In this section, we present the design and implementation of interrupt and exception handling systems [59] [60] for the ρ-VEX processor. The interrupt system is made parametrized and implemented in four different mechanisms with respect to interrupt latency, hardware utilization, and stress on the compiler and/or related toolchain. The exception handling system utilizes the interrupt system for its implementation.

3.3.1 Interrupt Handling System

The interrupt system that we implemented is called the *interrupter* and can be easily plugged in and out of the ρ-VEX core. Figure 3.5 depicts the interrupter embedded into a 4-issue pipelined ρ-VEX processor. The interrupter receives input signals from interrupt pins and then generates control signals to the fetch stage to reschedule instructions such that an *interrupt service routine (ISR)* could be executed. At the same time, the necessary context is stored in the data memory. When a *return from interrupt (RFI)* instruction is decoded, a signal is passed to the interrupter to indicate the end of an ISR. After that, the context is restored back to the core which then resumes the original execution. To generate a software interrupt, the *INT_SOFT* custom operation is implemented. The *INT_MASK* custom operation is implemented to enable and disable the individual interrupts. Following we discuss two sub modules of the interrupter called the *interrupt scheduler* and the *interrupt controller*.

Interrupt Scheduler The interrupt scheduler is configurable and the following parameters can be configured at design-time: (1) number of interrupt vectors, (2) interrupt priority for each vector, and (3) ISR location address in the instruction memory. The interrupt scheduler is responsible for: (1) receiving interrupt input signals from different sources, (2) scheduling different tasks into the task queue, and (3) enabling interrupt requests to the interrupt controller when priority of the requested task is higher than the current task. The interrupt scheduler dataflow is depicted in Figure 3.6(a). There are two inputs for the interrupt scheduler: external *interrupt in* signals from outside world and the internal *clear interrupt* flag signal from the interrupt controller. The former adds tasks to the task queue while the latter removes it. Based on the interrupt

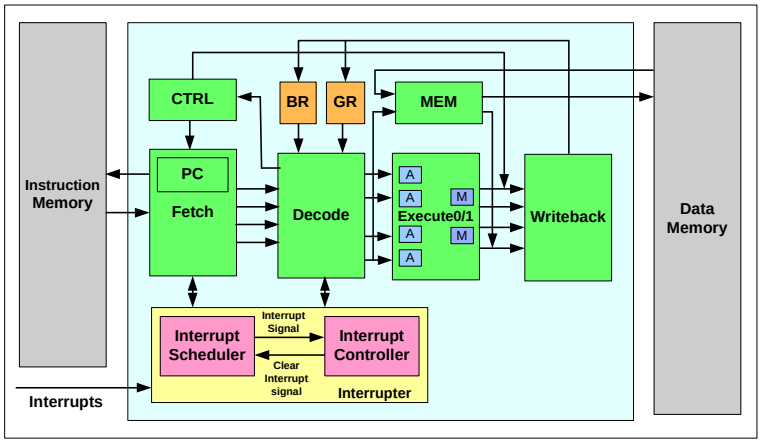
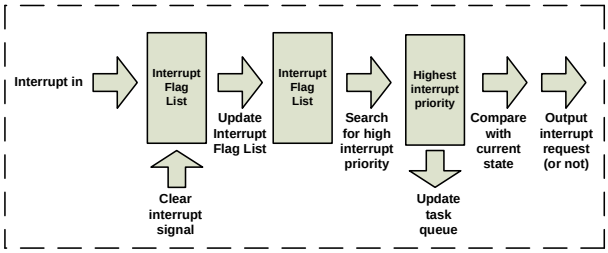


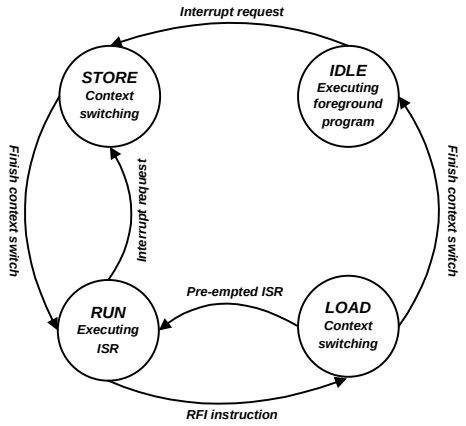
Figure 3.5: A 4-issue ρ-VEX processor with the interrupter.

priority, the scheduler decides whether to raise an interrupt to the interrupt controller or not. Only if an interrupt with higher priority comes in, or a higher priority task is finished, a waiting task can then become active. The interrupt vector table records all information that is necessary for scheduling different tasks. These include interrupt vectors (type of interrupts) and their priorities, interrupt flags which show the status of each interrupt request, ISR address (predefined or user defined), and interrupt enable bits to mask the interrupts.

Interrupt Controller The interrupt controller is responsible for context switching and ISR execution. It's main jobs are: (1) receiving interrupt request signals from the interrupt scheduler, (2) storing the context, (3) loading the ISR address, (4) restoring the context, and (5) restarting the main program again from the point where it was left before the interrupt. The efficiency of the control logic determines the length of interrupt latency. Unlike the interrupt scheduler, the interrupt controller cannot be designed pipelined, since the next state at each clock cycle is determined by the previous state. We designed the interrupt controller as one finite state machine (FSM) which is depicted in Figure 3.6(b). The input (i.e., interrupt request or RFI) signal determines the next state based on the current one. The interrupt system is pre-emptive and a currently executing ISR can be preempted to start a new one based on its priority. An interrupt queue is implemented to record information, such as, ISR addresses, return addresses, and interrupt vectors received from the interrupt scheduler along with the interrupt request signal.



(a) Interrupt scheduler dataflow



(b) Interrupt controller FSM

Figure 3.6: Dataflow and FSM in the interrupter.

3.3.2 Implementation Styles for the Interrupt Controller

We implemented the interrupt controller in four different methods in order to match different application requirements and hardware resource utilization. These implementations differ by the way the context (the GR and the BR registers) is stored and restored. The first 3 methods utilize the ρ-VEX processor whose register file is implemented with the FPGA's configurable resources (slice registers and slice LUTs), called here ρ-VEX type 'a'. The 4^{th} method utilizes the ρ-VEX processor whose register file is implemented with the FPGA's embedded BRAMs, called here ρ-VEX type 'b'.

Directly Switching Context Method: In this method, the context is stored/restored through dedicated paths without utilizing the processor pipeline. Here, the GR registers, the BR registers, and the data/shared memory are directly connected to the interrupt controller for context switching. Extra

multiplexers are utilized to select the path from either the pipeline or the interrupt controller to the register files and data memory. The advantage is that the ISR code becomes smaller, as instructions for context switching are not needed in the ISR.

Hardware Instructions Switching Context Method: Instead of using dedicated paths, this method utilizes the processor pipeline for context switching. The instructions needed for context switching are generated by the interrupt controller hardware and inserted into the pipeline. The advantage is that the ISR code becomes smaller, as instructions for context switching are not required in the ISR. Additionally, a hardware *monitor* is introduced which records the maximum index of registers at run-time in order to reduce the size of the context to be stored.

Software Instructions Switching Context Method: Here, the processor pipeline is utilized, and the instructions needed for context switching are generated in software. Hence, the complexity is shifted from hardware to software. Additional hardware for context switching is not required, however, we lose the possibility of only switching a subset of registers at run-time, as the instructions for context switching are fixed after compilation. Additionally, this method introduces extra overhead for the size of the ISR code.

Page-able Register File Method: Instead of storing/restoring the context, here, the page of the register file is switched before/after executing an ISR. Here, we utilized the ρ-VEX processor whose register file is implemented with BRAMs (ρ-VEX type 'b' instead of the ρ-VEX type 'a' as in the first 3 methods). A ρ-VEX processor requires a 64×32-bit register file. Multiple BRAMs are utilized to implement the register file as discussed in Section 3.2. The 18 Kbits BRAM-based register file can provide up to 512×32-bit registers or up to 8 copies/pages of 64×32-bit register file. We modified the register file design to exploit the unused registers as multiple sets of the register file.

3.3.3 Interrupt Latency and Response Time

The following key metrics determine the performance of an interrupt system: **Interrupt latency** - The time from when an interrupt is first generated to when the processor responds to the interrupt, i.e., the time when the processor is ready to start storing the context. **Interrupt response time** - The time from when an interrupt is first generated to when the processor runs the first instruction in an ISR. It includes the interrupt latency plus the time required for context storing and calling an ISR. Table 3.3 lists the implementation types,

Table 3.3: Implementation version, interrupt response time, and the worst-case interrupt latencies for the four types of interrupt system for the ρ-VEX processor.

Version	Description	Response time	Latency
1	Directly switching context	76 cycles	5 cycles
2	Hardware instructions switching context	17 – 76 cycles	5 cycles
3	Software instructions switching context	76 – 78 cycles	5 cycles
4	Page-able register file	2 – 6 cycles	2 cycles

the interrupt response time, and the worst-case interrupt latencies for the four types of our interrupt system with ρ-VEX processor.

In the ρ-VEX architecture, the GR register number `0` (*$r0.0*) is hardwired to value zero, therefore, it is not stored during context store. For version `1` of the interrupter, the interrupt response time of `76` cycles includes `4` cycles for completing the currently fetched instruction and stopping the pipeline, `1` cycle for scheduling the interrupt, `63` cycles for moving the GR registers and `8` cycles for moving the BR registers. For version `2`, a hardware monitor records the maximum index of the registers used in a running program before the processor is interrupted. Therefore, the interrupt response time depends on when the currently executing program is interrupted. The worst case could be `76` cycles. The best case could be `17` cycles (`4` cycles for completing the currently fetched instruction and stopping the pipeline, `1` cycle for scheduling the interrupt, `12` cycles for moving the GR registers (*$r0.1* to *$r0.11*, and *$r0.63*)). These registers have special purposes in the ISA and are mostly utilized in a program [4]. For version `3`, the interrupt response time is pre-determined at compile time. Still, there could be two scenarios. First, when the context storing routine is placed within the body of the ISR, the interrupt response time is `76` cycles. Second, when the context storing routine is placed at a separate location and is called from within the ISR, the interrupt response time is `78` cycles, as there would be an extra `2` cycles branch latency. In the latter case, the size of the ISR code is reduced. The interrupt response time for version `4` of the interrupter is `2` clock cycles. One cycle for scheduling the interrupt and another for switching the register file page. When implementing this method, the first and the last `4` instructions in the ISR should not read and write data from/to registers, respectively, in order to allow the currently fetched instruction to be passed through the pipeline. This is reasonable because at the beginning of a program (ISR), variables are normally initialized first before they can be read, and at the end of a program, the already computed data is consumed or spilled to memory instead of generating new data (writing to registers). If this assumption is not

valid then some NOP instructions (maximum 4) should be added at the start of the ISR code. In this case, the interrupt response time ranges from 2 to 6, depending upon how many NOP instructions are added to the code.

3.3.4 Exceptions Handling System

The difference between interrupts and exceptions is that interrupts are utilized to handle external events (serial ports, buttons etc.) while exceptions are used to handle internal instruction faults (arithmetic overflow, illegal opcode etc.) The exceptions handling system mainly relies on the interrupts system for its implementation. Unlike the interrupts which can occur asynchronously, exceptions occur synchronously when an instruction is decoded or executed. Different conditions are tested at decode and execute stages and internal interrupt is raised whenever there is an exception. Following we discuss different exceptions implemented for the ρ-VEX processor. The system can be easily extended with other types of exceptions.

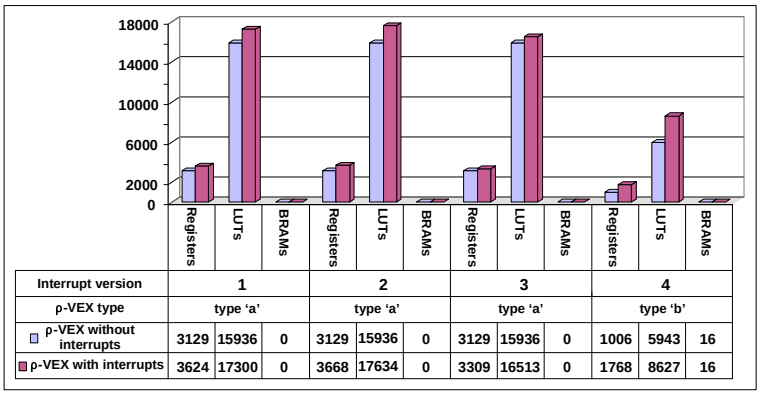
Arithmetic Overflow: Arithmetic overflow occurs when the result of an arithmetic/multiplication operation becomes larger than the size of the register used to hold it. Because these operations takes place in the execute stage, therefore, the overflow exception is detected in the execute stage. For an *arithmetic* operation, the maximum result of two 32-bit operands can be 33-bit, therefore, we can simply test the leftmost bit as the overflow flag and use it as an exception signal. Similarly, for *multiplication* operations of 32×16-bit, the leftmost 16 bits are tested for the overflow condition. When an overflow exception occurs, an internal interrupt is raised to the interrupter and an exception handling routine can thus be called.

Invalid Opcode: When a non-supported opcode is fetched into the processor pipeline, it triggers an invalid/illegal opcode exception. This condition can be detected in the decode stage as opcodes are decoded here. The opcode is tested in all issue lanes and then exception signals are generated to the interrupter, and an exception handling routine can thus be called.

Unavailable Hardware Unit: Different executions units are distributed over different issue lanes in a ρ-VEX processor, therefore, not all the operations can be executed in every lane. If an operation is assigned to a lane, which does not have the hardware unit to execute it, an exception signal is generated to the interrupter, and an exception handling routine can thus be called. This exception is also detected in the decode stage.

3.3.5 Implementation Results

Figure 3.7 depicts the hardware utilization for the 4 types of interrupt system with 4-issue ρ-VEX processor (3 with ρ-VEX type 'a' and 1 with ρ-VEX type 'b'). We utilized the Xilinx ISE release version 13.2 and the Virtex-6 *XC6VLX240T-1FF1156* FPGA for the synthesis and implementation. The ρ-VEX processor has 4 ALUs, 2 MULs, and 1 MEM unit, and can run up to 100 MHz in the Virtex-6 FPGA. A RAMB36E1 is equal to two RAMB18 BRAMs in the Virtex-6 FPGA. As can be observed from the figure, each implementation method of the interrupter requires different amount of hardware resources. The first 3 version of the interrupter are implemented for the ρ-VEX type 'a'. Compared to the ρ-VEX without interrupter, the ρ-VEX with interrupter version 1 requires 15.82% more registers and 8.56% more LUTs, the ρ-VEX with interrupter version 2 requires 17.26% more registers and 10.66% more LUTs, and the ρ-VEX with interrupter version 3 requires 5.76% more registers and 3.62% more LUTs. The version 4 of the interrupter is implemented for the ρ-VEX type 'b'. Compared to this version of the ρ-VEX without interrupter, the ρ-VEX with interrupter version 4 requires 75.75% more registers and 45.16% more LUTs. As the multiported register file for the type 'b' has 8 copies of the register set (512×32-bit in total), the direction table becomes large, and hence requires more slice registers of the FPGA for its implementation. The number of BRAMs remains the same. Remember that the Virtex-6 *XC6VLX240T-1FF1156* FPGA has 301440 slice registers and 150720 LUTs, and that all our designs utilize only a small portion of the device.



Interrupt version	1			2			3			4		
ρ-VEX type	type 'a'			type 'a'			type 'a'			type 'b'		
ρ-VEX without interrupts	3129	15936	0	3129	15936	0	3129	15936	0	1006	5943	16
ρ-VEX with interrupts	3624	17300	0	3668	17634	0	3309	16513	0	1768	8627	16

Figure 3.7: Implementation results for the 4 types of interrupt system with 4-issue ρ-VEX processor (3 with ρ-VEX type 'a' and 1 with type 'b') for a Virtex-6 FPGA. Each ρ-VEX processor (with/without interrupts) also utilizes 4 DSP48E1 modules.

3.4 Instruction Encoding Scheme

The issue-width of the ρ-VEX processor can be statically or dynamically selected to be 2, 4, or 8. A 2-issue, 4-issue, or 8-issue processor can execute 2, 4, or 8 operations per clock cycle, respectively. An operation is encoded in 32 bits, and multiple operations make a VLIW instruction which can be issued in a single clock cycle. Each operation is executed by an available FU in an issue-slot. Addresses of 6-bit and 3-bit are required to access the GR and BR registers, respectively. The regular FUs are ALUs, MULs, CTRL, and MEM. Each ρ-VEX processor can utilize a different mix and number of these FUs, except the CTRL, which is only one per processor. Increasing the number of these FUs increases the hardware resources, but simplifies the sorting of operations in the low-level development tools (assembler). Each FU requires at maximum two 32-bit inputs and generates one 32-bit output. Both of the input operands could be register values or one of them could be a register value and the second one an *immediate (IMM)* value. A short IMM (up to 9-bit) is encoded in the same 32-bit operation. When there is a long IMM (up to 32-bit), it cannot be encoded in the same operation. Hence, an additional operation space or issue-slot is required to carry the long IMM. For the ρ-VEX processor, a custom operation called *Syllable_Follow (S_F)* is implemented to carry a long IMM value for an operation. Hence, long immediates are handled in the same instruction utilizing multiple issue-slots. We designed a new encoding scheme for the ρ-VEX processors to increase the available opcodes from 128 to 256 which can be utilized for ISA extension. Additionally, the new encoding scheme defines proper positions for the regular FUs which makes the assembler tool simpler and uniform. With these positions, code generated for any of the 2-issue, 4-issue, or 8-issue standalone ρ-VEX processors can be executed correctly on the 2-4-8-issue run-time reconfigurable processor presented in Chapter 4.

3.4.1 Design of the New Encoding Scheme

Table 3.4 presents the old [14] [15] and the new encoding schemes for the ρ-VEX processor. The first 2 bits of the 32-bit operation encoding are reserved for multi-clustering and NOP-folding purposes. The ISA [4] includes operations which require two source and one destination GR registers and one BR register at the same time, therefore, the next 21 bits are needed for these registers addressing. In the old encoding scheme, 2 bits are utilized to encode the IMM type (“00” is no IMM, “01” is short IMM, and “1x” is long IMM). The

Table 3.4: The old and the new encoding schemes. IMM is flag for immediate types. Short IMM and long IMM are the values of the short and long immediates, respectively. S_F means Syllable_Follow custom operation.

31 25	24	23	22 17	16 11	10 5	4 2	1	0
Opcode Old	IMM Old		Dest. GR address	Src1 GR address	Src2 GR address	BR address	-	-
Opcode Old	IMM Old		Dest. GR address	Src1 GR address	Short IMM / Long IMM-1 (9-bit)		-	-
Opcode New		IMM New	Dest. GR address	Src1 GR address	Src2 GR address	BR address	-	-
Opcode New		IMM New	Dest. GR address	Src1 GR address	Short IMM / Long IMM-1 (9-bit)		-	-
S_F Old	Long IMM-2 (23-bit) Old						-	-
1 0 0 0 * * *	*	Long IMM-2 (23-bit) New					-	-
Opcode Old	Branch offset (20-bit) Old						-	-
Opcode New		Branch offset (19-bit) New					-	-

last 7 bits are left for opcode encoding, and hence, only 128 different operations could be implemented. Short IMM operands (up to 9-bit) are encoded in the same operation. When an operand is a long IMM, the first 9 bits of the IMM are carried by the same operation and the last 23 bits are carried by the *S_F* operation in a different issue-slot. Hence, an operation with a long IMM requires two issue-slots. With the old encoding scheme, there was hardly any free opcode left to extend the ISA.

The new encoding scheme utilizes 8 bits for opcodes and a single bit for the IMM type. "0" for IMM means no immediate operand, while "1" means an immediate operand, whose type (short or long) is determined by the *S_F* opcode in the same instruction as discussed in Section 3.4.2. With the new encoding scheme the opcode space is increased from 128 to 256. It utilizes an uniform approach for sorting operations for the 2-issue, 4-issue, and 8-issue stand-alone processors, and the 2-4-8-issue run-time reconfigurable processor (presented in Chapter 4), makes the assembler tool simple and uniform, and solves the problem of code versioning.

3.4.2 Borrowing Scheme and Instruction Mapping

Borrowing refers to the issue-slot, on which an operation can find the last 23 bits of its long IMM. Although, the number, type, and position of the regular FUs per issue-slot is a design-time configurable parameter, here, we consider a default number of FUs for each type of the ρ-VEX processor. For every type of the processor, we consider one CTRL unit, one MEM unit, and the same number of ALUs as the issue-width. We consider the number of MULs for

the stand-alone 2-issue, 4-issue, and 8-issue processors to be 2, 2, and 4, respectively. These numbers provide enough performance without exceeding the hardware resources. The 2-4-8-issue run-time reconfigurable ρ-VEX processor utilizes 8 MULs. In general, the more the number of the individual FUs, the simpler the borrowing scheme becomes.

Tables 3.5 and 3.6 present the positions of different FUs and borrowing schemes for the 2-issue, 4-issue, 8-issue, and 2-4-8-issue ρ-VEX processors, and their instruction mapping schemes. The left-most column presents the possible combination of operations (with/without long immediates) making different VLIW instructions. Mapping scheme presents the possible instructions and how they can be accommodated in the available issue-slots. ALU operations are not considered as an ALU is available in every issue-slot. Branch immediate is restricted to 19-bit maximum and requires a single issue-slot with the CTRL unit. An operation with a short IMM is scheduled on a single conflict-free issue-slot, while that with a long IMM is scheduled on two issue-slots. The first 9 bits of the long IMM are carried by the operation slot, while the last 23 bits are carried by an *S_F* operation scheduled on a different issue-slot. The *S_F* is a custom operation with the new opcode of: $1000----$ (8-bit), utilizing a space of 16 opcodes. Bit 0 of the *S_F* opcode is used to carry the last bit of a long IMM (bit 32). Bits 3 to 1 of the *S_F* opcode represent the number of the issue-slot for which the *S_F* operation is carrying the last 23 bits of the long IMM. Because the *S_F* opcode reserves 16 opcodes, the total number of additional free opcodes provided by the new encoding scheme is 128 - 16 = 112.

3.5 ISA Extension Support

In this section, we provide a design methodology to extend the instruction set and generate binary code for user-defined/custom operations for the ρ-VEX processor. With the new encoding scheme, there are 112 free opcodes that can be utilized to extend the ISA. The VEX compiler can generate binary code for custom operations that are defined in a C application. Following our methodology, it is very easy to implement the hardware for a custom operation for the ρ-VEX processor. Additionally, users can select to add certain common custom operations (e.g., *abs*, different *sub-word operations*, etc.) at design time to the ρ-VEX processor.

Table 3.5: Position of FUs, borrowing scheme for long IMM, and instruction mapping for the 2-issue and 4-issue ρ-VEX processors. Here, AU, MU, MM, CT, S, and L mean ALU, MUL, MEM, CTRL, short, and long, respectively.

Slot number	1	0
Functional units	AU MU MM	AU MU CT
Borrowing scheme	0	1
Mapping scheme for instructions		
MU L	MU L	S_F(1)
MM L	MM L	S_F(1)
MU1 S , MU2 S	MU2 S	MU1 S
MU S , MM S	MM S	MU S
MU S , CT	MU S	CT
MM S , CT	MM S	CT

(a) 2-issue ρ-VEX processor

Slot number	3	2	1	0
Functional units	AU MM	AU MU	AU MU	AU CT
Borrowing scheme	0 , 2	3 , 1	2 , 0	1 , 3
Mapping scheme for instructions				
MU1 L , MU2 L	S_F(2)	MU2 L	MU1 L	S_F(1)
MU1 L , MM L	MM L	S_F(3)	MU1 L	S_F(1)
MU1 L , MM S , CT	MM S	MU1 L	S_F(2)	CT
MU1 L , MU2 S , MM S	MM S	MU2 S	MU1 L	S_F(1)
MU1 L , MU2 S , CT	S_F(2)	MU1 L	MU2 S	CT
MU1 S , MU2 S , MM L	MM L	MU1 S	MU2 S	S_F(3)
MU1 S , MU2 S , MM S , CT	MM S	MU1 S	MU2 S	CT
MU1 S , MM L , CT	MM L	S_F(3)	MU1 S	CT

(b) 4-issue ρ-VEX processor

3.5.1 Binary Code Generation for Custom Operations

The VEX compiler has support for user-defined operations at C language level with the help of special intrinsic called *_asm()*. When a call is inserted to _asm() in a C program with proper parameters, the operation is scheduled and registers are allocated by the compiler. Hence, C variables for operands and destinations can be referred. Calls to _asm() are interpreted in a special way. The "vexasm.h" header file [1] includes the implicit function prototypes for the _asm() intrinsics, as presented in Figure 3.8.

The opcode argument is a numeric identifier for the operation. Operations defined with _asm() intrinsic can have up to 8 optional arguments after opcode that represent the values read by the operation. These operations can have no or up to 4 return values. Taking the address of an _asm*() function is illegal. Figure 3.9 presents an example of the _asm() usage implementing a division (DIV) function and its VEX assembly code for a 2-issue processor. In this example, the intrinsic *0x01* implements the division of two numbers. It is called with two arguments and stores its result in a third variable. This is the simplest implementation for a DIV operation and cannot handle the case when a division by zero occurs. This example is just for illustration purpose to

Table 3.6: Positions of FUs, borrowing scheme for long IMM, and instruction mapping for the 8-issue and 2-4-8-issue ρ-VEX processors. Here, AU, MU, MM, CT, S, and L mean ALU, MUL, MEM, CTRL, short, and long, respectively.

Slot number	7	6	5	4	3	2	1	0
Functional units	AU , MM	AU , MUL	AU , MUL	AU	AU	AU , MUL	AU , MUL	AU , CT
Borrowing scheme	4 , 6	7 , 5	6 , 4	5 , 3	4 , 2	3 , 1	2 , 0	1 , 3
Mapping scheme for instructions								
MUL1 L , MUL2 L , MUL3 L , MUL4 L	S_F(6)	MUL4 L	MUL3 L	S_F(5)	S_F(2)	MUL2 L	MUL1 L	S_F(1)
MUL1 L , MUL2 L , MUL3 L , MM L	MM L	S_F(7)	MUL3 L	S_F(5)	S_F(2)	MUL2 L	MUL1 L	S_F(1)
MUL1 L , MUL2 L , MUL3 S , MUL4 S , MM L	MM L	MUL4 S	MUL3 S	S_F(7)	S_F(2)	MUL2 L	MUL1 L	S_F(1)
MUL1 L , MUL2 L , MUL3 S , CT , MM L	MM L	S_F(5)	MUL2 L	S_F(7)	S_F(2)	MUL1 L	MUL3 S	CT
MUL1 L , MUL2 S , MUL3 S , MUL4 S , CT , MM L	MM L	MUL4 S	MUL3 S	S_F(7)	S_F(2)	MUL1 L	MUL2 S	CT
MUL1 S , MUL2 S , MUL3 S , MUL4 S , MM L , AU L	MM L	MUL4 S	MUL3 S	S_F(7)	S_F(0)	MUL2 S	MUL1 S	AU L
MUL1 L , MUL2 L , MUL3 S , MUL4 S , MM S , CT	MM S	MUL4 S	MUL2 L	S_F(5)	S_F(2)	MUL1 L	MUL3 S	CT
MUL1 S , MUL2 S , MUL3 S , MUL4 S , MM L , CT , AU S	MM L	MUL4 S	MUL3 S	S_F(7)	AU S	MUL2 S	MUL1 S	CT

(a) 8-issue ρ-VEX processor

Slot number (2-issue)	1	0	1	0	1	0	1	0
Slot number (4-issue)	3	2	1	0	3	2	1	0
Slot number (8-issue)	7	6	5	4	3	2	1	0
Functional units	AU , MUL , MM	AU , MUL , CT	AU , MUL , MM	AU , MUL , CT	AU , MUL , MM	AU , MUL , CT	AU , MUL , MM	AU , MUL , CT
Borrowing scheme	Same as for 2-issue, 4-issue, and 8-issue. Any issue-width code will execute correctly.							
Mapping scheme	Same as for 2-issue, 4-issue, and 8-issue. Any issue-width code will execute correctly.							

(b) 2-4-8-issue ρ-VEX processor

```
/* From "<vex>/usr/include/vexasm.h" */

typedef unsigned int                              __vexasm1;
typedef struct {unsigned int n0, n1;}             __vexasm2;
typedef struct {unsigned int n0, n1, n2;}         __vexasm3;
typedef struct {unsigned int n0, n1, n2, n3;}     __vexasm4;

void        _asm0 (int opcode, ...);
__vexasm1   _asm1 (int opcode, ...);
__vexasm2   _asm2 (int opcode, ...);
__vexasm3   _asm3 (int opcode, ...);
__vexasm4   _asm4 (int opcode, ...);
```

Figure 3.8: Prototypes for the _asm() intrinsics [1].

show how a user-defined operation can be defined and compiled with the VEX compiler. The compiler schedules the code around the intrinsic call *_asm()* and operates the usual optimizations and register allocation tasks. The *_asm* intrinsic is distinguished by the opcode, which is 1 in this example. The latency and the number of occupied issue-slots for an *_asm()* operation can be set in the machine configuration file which is provided to the compiler.

3.5.2 Methodology to Extend the ISA

Custom operations are defined at the C language level in the source code. The modified C source code with the custom operations can be simulated with the VEX simulator [1] for performance analysis. Figure 3.10 depicts a methodology/flowchart that can be used to implement a custom operation for the ρ-VEX

```
#include <vexasm.h>
#define DIV(x, y) ((int)_asm1(0x01, (x), (y)))

void main () {
        int x, y, z;
        x = 2 ; y = 4;
        z = DIV(x, y); }
```

```
;;
c0 mov $r0.3 = $r0.0
c0 mov $r0.2 = 2
;;
c0 asm,1 $r0.2 = $r0.2, 4
c0 return $r0.1 = $r0.1, (0x0), $l0.0
;;
```

Figure 3.9: The _asm() usage example for implementing a division (DIV) function and its VEX assembly code for a 2-issue ρ-VEX processor.

processor. First of all select an opcode from the available unused opcodes and add it to the opcode_pkg.vhd file. This package file contains the opcode constants and parameters. After this, determine whether the only decoding the new operation is enough or execution is also needed. If execution is also required determine which of the available FUs (ALU, MUL, CTRL, or MEM) will execute the new operation. In the selected FU, add code to select the result based on the new operation. This will mainly comprise of adding an *elsif statement* to the selected FU design file (e.g., for ALU, this file is the alu.vhd). Also add the prototypes and the function definitions related to the functionality of the new operation to the package file for the selected FU (e.g., for ALU, this file is alu_operations.vhd). Finally, adjust the decoder (decode.vhd) by adding the decode logic and elsif statement to generate and select the required signals and values for the new operation. For operations which only require decoding (such as an operation for masking an interrupt), there is no need to update the execute unit. Modifying the decoder is enough to generate the required signals. Instead of adding to the already available standard FUs, operations can also be added as separate custom FUs. In this case, the custom FU is placed in a lane in parallel to the standard FUs. The final result is selected either from the custom FU or the regular FUs depending upon the signals from the decoder. A custom operation requiring more than two inputs and/or generating more than one output can be expanded over multiple execute lanes. One execute lane can execute an operation with at maximum two inputs and one output. The opcode will be decoded by the selected decode lanes simultaneous each accessing a different set of registers for the operation expanded over multiple lanes.

3.5.3 Design-time Selectable Custom Operations

We provide some commonly used operations that are not part of the VEX ISA as design-time custom operation for the ρ-VEX processor, as listed in Table 3.7. These operations can be enabled to be included in a processor by setting a bit in the rVEX_package at design time. Sub-word operations utilize a 32-bit operation slot for either two 16-bit operations or four 8-bit operations. These operations are very common, especially in multimedia applications such as pixel manipulation. Utilizing these sub-word operations, the throughput for operations operating on bytes and double-bytes can be increased. Figure 3.11 presents the hardware results for these operations for a *4*-issue pipelined ρ-VEX processor with 4 ALU, 2 MUL, and 1 MEM units. The operations are included in the regular/default ALU. In Figure 3.11, the *Reference* design represents the base 4-issue ρ-VEX processor, to which the custom operations

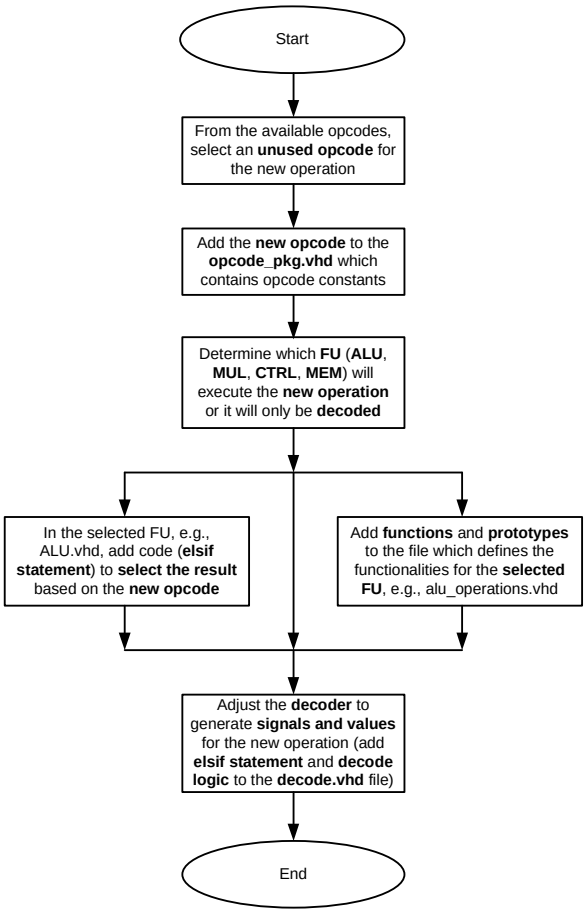


Figure 3.10: Methodology/Flowchart for implementing a custom operation.

are then added. These operations reduce the maximum clock frequency in the range of 3% to 14%, and utilize moderate hardware resources depending upon the number of ALUs containing the operations. With the help of the VEX simulator, the total gain (in terms of executions cycles) due to adding a custom operation to the design can be determined in advance. If the clock cycle reduction is more than the degradation due to the critical path increase, the designer can choose to add the custom operation by setting a bit in the rVEX_pkg.vhd file.

Table 3.7: List of design-time available custom operations.

Custom operation	Description
VECT2ADD16	two 16-bit additions in a single 32-bit slot
VECT4ADD8	four 8-bit additions in a single 32-bit slot
VECT2SUB16	two 16-bit subtractions in a single 32-bit slot
VECT4SUB8	four 8-bit subtractions in a single 32-bit slot
VECT2SHR16	two 16-bit shift left in a single 32-bit slot
VECT4SHR8	four 8-bit shift left in a single 32-bit slot
VECT2SHL16	two 16-bit shift right in a single 32-bit slot
VECT4SHL8	four 8-bit shift right in a single 32-bit slot
PACK16HIGH	packs the higher 16 bits of two operands
PACK16LOW	packs the lower 16 bits of two operands
ABS	absolute of a 32-bit number
VECT2ABS16	two 16-bit absolute in a single 32-bit slot
VECT4ABS8	four 8-bit absolute in a single 32-bit slot

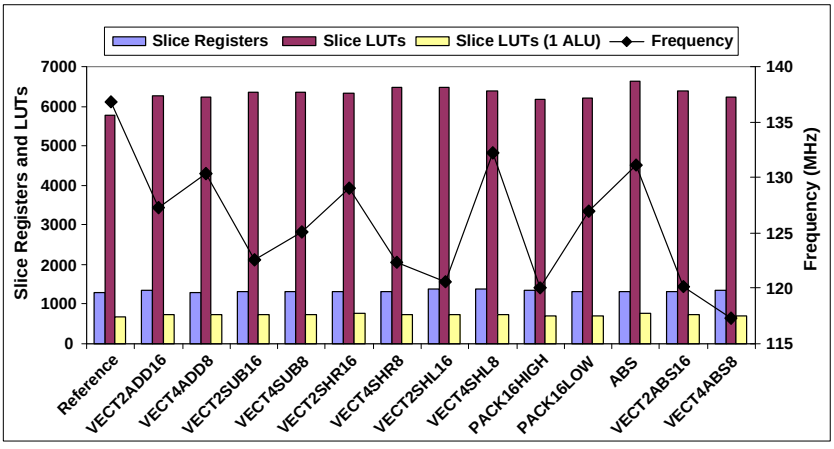


Figure 3.11: Implementation results for the custom operations listed in Table 3.7 for a 4-issue ρ-VEX processor with 4 ALU, 2 MUL, and 1 MEM units for the Virtex-6 FPGA. The processor also requires 32 RAMB18s and 4 DSP48E1s modules.

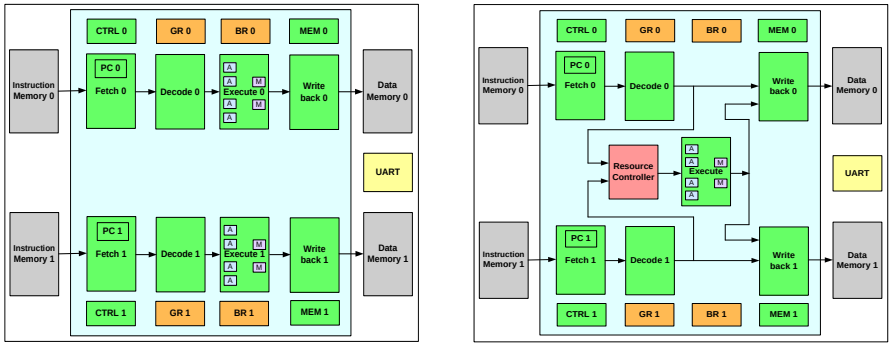
3.6 Datapath Sharing

The higher performance of VLIW processors does not come for free as their resources do not scale well. As stated previously in the Section 3.2, the hardware/area requirement for a multiported register file is directly proportional to the number of read and write ports, and these parameters do not scale to a large extent. To reduce the pressure on the number of read and write ports of the register file, a *clustered* architecture is used. A cluster is a collection of a register file and a set of tightly coupled FUs. A multi-cluster processor has multiple clusters, but a single execution thread, while a multiprocessor has multiple processors and may have multiple execution threads. Clustered VLIW processors do not scale well in terms of performance due to the inter-cluster communication. The delay resulting from inter-cluster communication reduces the machine performance. For example, a 16-unit/2-cluster processor performs roughly like a 12-unit/1-cluster processor and an 8-unit/2-cluster processor like a 6-unit/1-cluster processor [10].

To avoid this problem, we designed a dual-processor system utilizing the non-pipelined ρ-VEX processor as its base. Parameters for each base processor such as the issue-width, the number and type of FUs, supported instructions, type and size of register file, etc., can be selected at design-time. The base processor has a multi-cycle design and consists of *fetch*, *decode*, *execute*, and *writeback* stages/units. During execution of a code, only one unit of the base processor is active per clock cycle, hence, FUs can be shared among different instances of the processor. A VLIW multiprocessor system (where each processor is a VLIW processor) can exploit both fine-grain (instruction level) as well as coarse-grain (data level) parallelism.

3.6.1 Dual-processor System

Figure 3.12(a) depicts a dual-processor system implemented with two non-pipelined 4-issue ρ-VEX processors. Each base processor can access its own instruction and data memories. The dual-processor system can target TLP or DLP, while the individual processor can exploit ILP. For example, if we need to encrypt 100 Kbytes of data according to advanced encryption standard (AES) algorithm, we run the application code on both processors each encrypting 50 Kbytes of data and then combine the result. Thus, we can achieve almost twice the performance of a single-processor system.



(a) Non-shared datapath **(b)** Shared datapath

Figure 3.12: VLIW dual-processor systems.

3.6.2 Datapath-shared Dual-processor System

Figure 3.13 presents the implementation results (FPGA slices) for the base processor for the Virtex-II Pro *XC2VP30-7FF896* FPGA. As can be observed from the figure, the *execute* unit and the GR register file version `1` (`64` registers) require `35.50`% and `59.02`% slices, respectively, of the total processor slices. Apart from the slices, the processor/execute unit also utilizes `14` *MULT18X18s*. We modified the design of the dual-processor system presented in Section 3.6.1. Instead of implementing an execute unit in each base processor, we developed a scheme to share it [61]. Because the base processor has a non-pipelined design, we can share the execute unit between two processors. In a non-pipelined processor, a new instruction is only fetched when the older one gets executed and results written back. Hence, the execute unit is not active all the time and can be utilized by the second processor. Both processors execute their own threads sharing a single execute unit, thus reducing hardware area and power consumption. In the current design of ρ-VEX, only two cores can share a single execute unit. If a larger multiprocessor is needed, e.g., a quad-processor system, two dual-processor systems each with a single execute unit can be combined. Figure 3.12(b) depicts our datapath-shared dual-processor system.

We designed a *Resource Controller* unit for the datapath-shared dual-processor system to share the single *execute* unit. It takes inputs from the *decode* units of the two processor cores, resolves some conflicts, multiplexes, and provides these inputs periodically to the single execute unit. Output from the shared execute unit are supplied to the *writeback* units of both the cores at the same time.

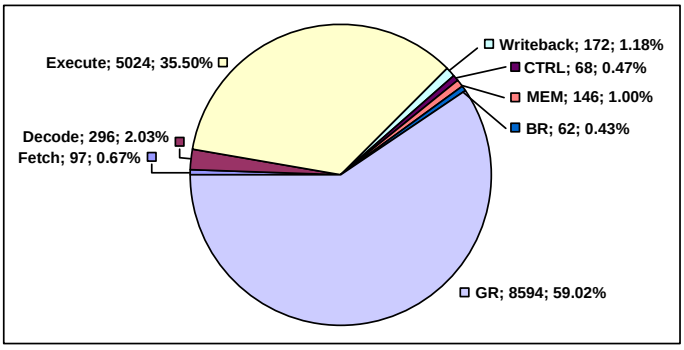


Figure 3.13: Implementation results (slices) for the base 4-issue non-pipelined ρ-VEX processor's modules for the Virtex-II Pro FPGA. The complete processor requires 14561 slices and 14 MULT18X18s. The register file is 64×32-bit.

The intended writeback unit writes the results based on the input signals from its corresponding decode unit. The *Resource Controller* utilizes 1606 slices and runs at a maximum frequency of 168.87 MHz. It reduces the critical path by avoiding the logic and connections of the additional execute unit, thereby, increasing the clock frequency of the datapath-shared system compared to the non-shared datapath system.

3.6.3 Implementation Results

Figure 3.14 presents the implementation for our dual-processor systems (shared and non-shared) for the same Virtex-II Pro FPGA. Although the ρ-VEX processor is parametrized, the base processor used in the dual-processor systems has 4 issue-slots with 4 ALUs, 2 MULs, 1 MEM unit, a 64×32-bit 4W8R ports GR register file, and an 8×1-bit 4W4R ports BR register file. From Figure 3.13 and Figure 3.14, we can observe that the hardware utilization becomes double for the dual-processor system compared to a single-process system, as expected. The datapath-shared dual-processor system reduces this hardware utilization by sharing the execute unit between two processor cores. We can observe a similar trend in hardware resources, when the number of multiported registers is increased from 8 to 64. To reduce the resources utilized by the register file version 1 (which requires more than 59% of the total base processor slices), we utilized the BRAM-based register file version 5 (register renaming) presented in Section 3.2.2. The results for the dual-processor systems (shared and non-shared) with the BRAM-based register file are depicted in Figure 3.14. Compared to the version 1 register file, the dual-processor

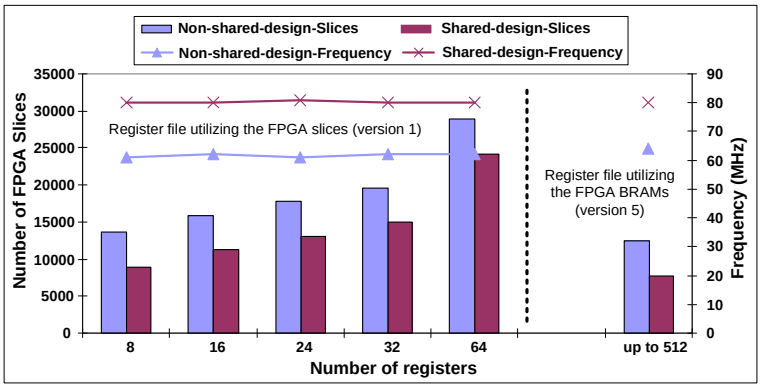


Figure 3.14: Implementation results for the dual-processor system (shared and non-shared) for a Virtex-II Pro FPGA. Apart from the slices, the datapath-shared and non-shared dual-processor systems also require 14 and 28 MULT18X18s, respectively. The BRAM-based design also utilizes 64 RAMB18s.

systems with version 5 register file considerably reduces the slice utilization at the expense of BRAM utilization. Consequently, we have two alternate designs for the dual-processor systems. If the designer has extra area/slices, he/she can instantiate the slice-based design. If slices are limited, the designer can instantiate the BRAM-based design. The datapath-shared dual-processor system runs at higher clock frequency compared to the non-shared-datapath dual-processor system.

3.6.4 Related Work

Softcore multiprocessor systems as found in literature are mostly based on either the MicroBlaze or the Nios-II softcore processors. Altera provides a tutorial [62] for creating a multiprocessor system utilizing the Nios-II processor. The tutorial provides a complete design flow from hardware building to software programming. A design of a symmetric multiprocessing on programmable chips utilizing the Nios-II softcore as the basic building block is presented in [63]. Similarly, different designs of MicroBlaze-based multiprocessor systems are available [64] [65] [66] [67] [68]. The main drawback of all these designs is that they are using proprietary softcores which are not open-source. Additionally, the Nios-II and the MicroBlaze are single-issue processor cores and cannot exploit ILP like a VLIW processor.

3.7 Summary

In this chapter, we presented a methodology to implement an instance of the open-source design-time configurable ρ-VEX processor. Configuration files describing different types of parameters for the ρ-VEX processor, such as issue-width, types of FUs, register file size, etc., are utilized for architectural exploration. These files are provided to the VEX compiler/simulator for performance analysis and code generation. The same files provide input to the parametrized VHDL description for a ρ-VEX processor generation. Hence, without having any knowledge of the HDLs, a user can generate a desired/optimized ρ-VEX processor. The compiler generated code is assembled into instruction and data memory files which can be synthesized together with the rest of the processor design files. The chapter furthers by presenting different types of multiported register files and different types of interrupt and exception systems to match different application requirements. Additionally, a new instruction encoding scheme and a methodology to add user-defined operations to a ρ-VEX processor has been presented. In the end, a datapath sharing mechanism has been explored in a dual-processor system to reduce its hardware utilization.

Note.

The content of this chapter is partially based on the following papers:

F. Anjam, S. Wong, and M.F. Nadeem. A Multiported Register File with Register Renaming for Configurable Softcore VLIW Processors. In *International Conference on Field Programmable Technology (FPT)*, pp. 403–408, 2010.

F. Anjam, S. Wong, and M.F. Nadeem. A shared Reconfigurable VLIW Multiprocessor System. In *International Parallel and Distributed Processing Symposium (IPDPS-RAW)*, pp. 1–8, 2010.

F. Anjam, Q. Kong, R.A.E. Seedorf, and S. Wong. A Run-time Task Migration Scheme for an Adjustable Issue-slots Multi-core Processor. In *International Symposium on Applied Reconfigurable Computing (ARC)*, pp. 102–113, 2012.

F. Anjam, Q. Kong, R.A.E. Seedorf, and S. Wong. On the Implementation of Traps for a Softcore VLIW Processor. In *HiPEAC Workshop on Reconfigurable Computing (WRC)*, 2012.

4

Run-time Reconfigurable Processor

Issue-width is *an important parameter for a VLIW processor. Increasing the issue-width can improve the performance of an application by exploiting the increased ILP. On the other hand, larger issue-width processors consume more power due to increased datapath. Therefore, VLIW processors whose datapath can be reconfigured at run-time are needed to target performance vs. power consumption trade-offs. In Chapter* 3*, we presented a design-time configurable VLIW processor that could change its organization before it is implemented in hardware. In this chapter, we extend that design to make it run-time reconfigurable. The run-time reconfigurable processor utilizes multiple* 2*-issue* ρ*-VEX cores each of which can run independently. If not in use, each core can be taken to a lower power mode by gating off its source clock. Multiple* 2*-issue cores can be combined at run-time to form a variety of configurations of VLIW processors. The run-time reconfigurable parameters include the issue-width, the number and type of FUs, and the size of the general register file. The processor can target a variety of applications having instruction and/or data level parallelism. Following are the contributions of the chapter:*

- *A run-time reconfigurable multi-core processor is presented. The smaller cores can be utilized independently to exploit thread level parallelism or can be combined at run-time to form larger issue-width cores to exploit ILP. Performance vs. power consumption trade-offs can be achieved at run-time.*
- *A mechanism for run-time task migration among different cores of the multi-core processor is implemented to improve the performance or reduce the power consumption of the processor at run-time.*
- *A setup for analysis of simultaneous reconfiguration of issue-width and instruction cache for the run-time reconfigurable processor is presented.*

The remainder of the chapter is organized as follows. Section 4.1 presents the design and implementation of our run-time reconfigurable processor. The design of a dynamically reconfigurable register file is discussed in Section 4.2. Section 4.3 presents a run-time task migration scheme to shift a task running on one type of core to another for performance improvement or power reduction. Section 4.4 presents a setup to analyze the effect of simultaneous reconfiguration of issue-width and instruction cache on performance improvement and energy consumption of the run-time reconfigurable processor. Finally, the chapter is concluded by presenting a summary in Section 4.5.

4.1 Run-time Reconfigurable/Adaptable Processor

Figure 4.1 depicts the execution cycles normalized to an 8-issue core for *matrix multiplication*, *secure hash algorithm (SHA)*, and *quick sort (Qsort)* applications. It can be observed from the figure that increasing the issue-width from 2 to 4 and 8 increases the performance considerably for the matrix multiplication. For the SHA, the change in issue-width from 2 to 4 or 2 to 8 increases the performance considerably, but going from 4 to 8-issue only produces a small 10% increase in performance. For the Qsort, there is almost no change for different issue-widths. All these three applications have different ILP, and hence, a specific issue-width processor can provide the maximum possible performance at reasonable power budget. For a non-reconfigurable VLIW processor, the configuration and issue-width of the processor are fixed at design time. Therefore, the issue-width cannot be adjusted to suit a different set of applications after fabrication.

Utilizing the design presented in Chapter 3, we implemented two versions of run-time reconfigurable/adaptable/adjustable issue-slots VLIW processors called *2-4-issue* [69] and *2-4-8-issue* [70] processors. The issue-width and the number and type of FUs in these processors can be reconfigured at run-time while the processor is active and running. The 2-4-issue processor has two 2-issue ρ-VEX cores, which can be used independently or combined together to form a 4-issue processor at run-time. The processor is implemented utilizing the Xilinx partial reconfiguration flow. With the help of a small set of external control signals and loading a small partial bitstream, the processor issue-width can be reconfigured. The 2-4-8-issue processor has four 2-issue ρ-VEX cores, which can be used independently or multiple 2-issue cores can be combined together to form larger issue-width processors at run-time. The processor is implemented utilizing the virtual reconfiguration flow. A set of external sig-

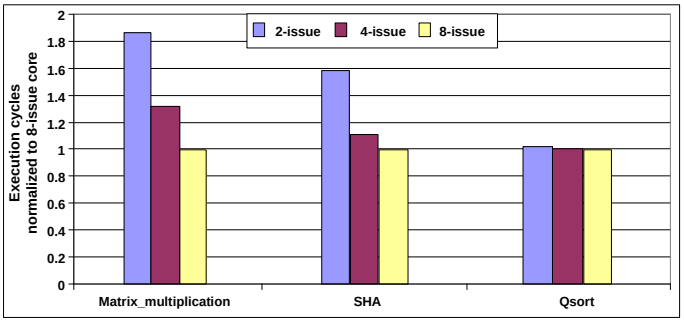


Figure 4.1: Execution cycles for matrix multiplication, SHA, and Qsort applications.

nals controls the configuration of the issue-width, and there is no need for partial bitstream. If not in use, each 2-issue core in the 2-4-issue and 2-4-8-issue processors can be taken to a lower power mode by gating off the source clock, and hence, the total power consumption of the processors can be reduced at run-time. Before an application starts execution, the machine's organization can be adjusted to suit the application requirements. Applications with more fine-grain (instruction level) parallelism can be run on the larger issue-width cores for better performance, while applications with more coarse-grain (data level) parallelism can be run on multiple 2-issue cores with the data divided among the cores for faster execution. Performance vs. power consumption trade-offs can be achieved for different applications at run-time.

4.1.1 Reconfiguration Flows

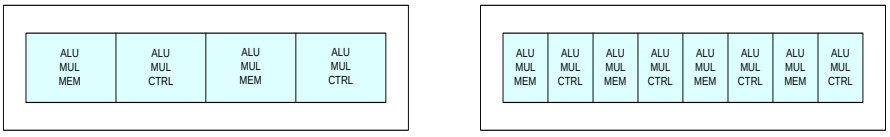
In this section, we discuss the two reconfiguration flows that are utilized to design the 2-4-issue and 2-4-8-issue reconfigurable processors.

Virtual Reconfiguration Flow In virtual reconfiguration, all the hardware resources required for a design implementation are made available. The design is pre-placed and the reconfiguration is provided by utilizing different multiplexers and turning certain modules ON and OFF. External/internal signals driven by configuration register bits control the reconfiguration/re-adjustment of the running system. Designs with virtual reconfiguration flow are simple to implement and require only few cycles for reconfiguration. The disadvantage is that all the required resources have to be made available all the time even when those are not in use. Our 2-4-8-issue reconfigurable processor is implemented utilizing the virtual reconfiguration flow.

Partial Reconfiguration Flow Partial reconfiguration is utilized to time share certain hardware resources among different modules of a design at run-time. According to the Xilinx *early access partial reconfiguration (EAPR)* design methodology [71], a design is split into *static* and *reconfigurable* regions. The static region contains those parts of the design which do not require run-time reconfiguration. The reconfigurable region contains the modules which require run-time reconfiguration. The communication between the static and reconfigurable regions is provided by special modules called *bus macros*. Separate bitstreams are generated and downloaded to an FPGA at run-time in order to change the functionality of a reconfigurable region. Our 2-4-issue processor is designed according to the partial reconfiguration flow, although it also utilizes the virtual reconfiguration flow for some parts of the design. The advantage of partial reconfiguration flow is that resources/area can be shared among different modules. The disadvantage is that it takes longer (in the range of milliseconds) to reconfigure a module in the current FPGA technology.

4.1.2 Design of the Run-time Reconfigurable Processors

In this section, we present the design of the 2-4-issue and 2-4-8-issue reconfigurable processors. Each processor consists of multiple 2-issue base ρ-VEX cores. The base processor is pipelined consisting of *fetch*, *decode*, *execute0*, *execute1/memory,* and *writeback* stages/units. Figure 4.2 depicts the FUs available per issue-slot in the 2-4-issue and 2-4-8-issue processors. Each 2-issue base core has two ALUs, two MULs, a MEM and a CTRL unit. Therefore, every issue-slot in the 2-4-issue and 2-4-8-issue processors has an ALU and a MUL, while MEM and CTRL units are available in alternate issue-slots. A 4-issue core has double the resources of a 2-issue core except that only one of the CTRL units is utilized when the two 2-issue cores are combined. Similarly, an 8-issue core has double the resources of a 4-issue core except that only one of the CTRL units is active when multiple 2-issue cores are combined.

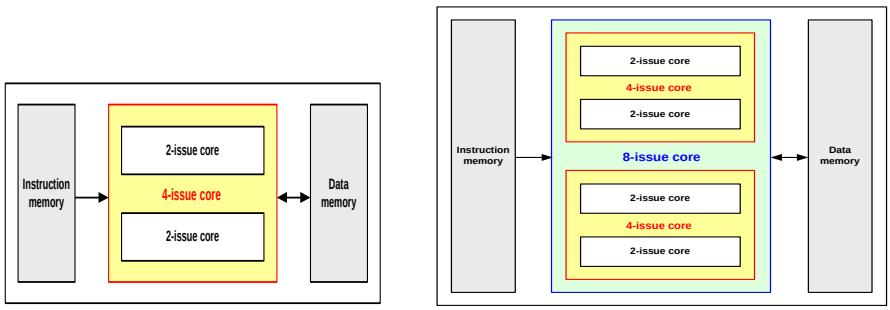


(a) The 2-4-issue processor. **(b)** The 2-4-8-issue processor.

Figure 4.2: Execution units in different issue-slots.

A signal called *issue_ctrl* manages the issue-width reconfiguration/adjustment. The signal is controlled by dedicated bits in the configuration registers of the processors. The 2-4-issue processor utilizes a single bit *issue_ctrl* signal. When this signal is at logic *low*, the two 2-issue cores can be utilized independently. When this signal is at logic *high*, the two 2-issue cores are combined and they behave like a single 4-issue core with double the resources of a 2-issue core. The 2-4-8-issue processor utilizes a two-bit *issue_ctrl* signal. When the *issue_ctrl* bits are "00", the system behaves as four independent 2-issue cores, when "01", the system behaves as two 2-issue and one 4-issue cores, when "10", the system behaves as two 4-issue cores, and when "11", the system behaves as an 8-issue core. After these bits are written into the configuration register, the configuration and the issue-width are changed in a single cycle. The unused FUs and parts of the processors are clock gated to reduce the dynamic power consumption. Both the 2-4-issue and 2-4-8-issue processors consist of different units, namely *fetch*, *decode*, *execute*, and *write-back*. In order to make the processors run-time reconfigurable, we combined these units into two modules called *frontend* and *backend*. Figure 4.3 depicts the general views of the 2-4-issue and 2-4-8-issue processors.

Frontend The frontend of the 2-4-issue and 2-4-8-issue processors requires reconfiguration/adjustment for changing the configuration and issue-width at run-time. It consists of the fetch and decode units, and the GR and the BR register files. For the 2-4-issue processor, the decode unit is the only module that is reconfigured by loading a partial bitstream to switch between two 2-issue cores to one 4-issue core or vice versa. The other modules are controlled by the *issue_ctrl* signal, and they do not require a partial bitstream for



(a) The 2-4-issue processor. **(b)** The 2-4-8-issue processor.

Figure 4.3: General view of the run-time reconfigurable issue-slots processor.

reconfiguration. This is done in order to minimize the number of resources to be reconfigured and hence, minimize the size of the partial bitstream. This resulted in reduced configuration time as well as reduced memory storage for the partial bitstreams. The 2-4-8-issue processor utilizes virtual reconfiguration and does not require a partial bitstream for reconfiguration. All modules in the frontend are reconfigured utilizing the *issue_ctrl* signal and all the resources for the processor are already available and pre-placed. It is only a matter of turning ON and OFF some of the resources in order to change the processor configuration. Following we discuss all modules of the frontend for both the 2-4-issue and 2-4-8-issue processors.

Fetch Unit A 2-issue fetch unit splits an incoming long instruction into two syllables (operations for individual execution units), and then passes them to the decode unit. Therefore, multiple 2-issue fetch units can be combined together to form a combined fetch unit to behave like a larger issue (up to 4-issue for the 2-4-issue processor and up to 8-issue for the 2-4-8-issue processor) fetch unit. Every 2-issue fetch unit has a *program counter (PC)*, which generates the next address for the instruction memory. The only module of the fetch unit that needs to be reconfigured is the PC. If multiple fetch units are combined to form a larger issue-width core, only one of the PCs is running and other PCs in that specific larger issue-width core are stopped. The signal *issue_ctrl* is utilized for this purpose.

Decode Unit Multiple 2-issue decode units can be combined together to form a decode unit for a larger issue-width processor. The *branch/CTRL* unit which calculates the offset and the branch target addresses is included in every 2-issue decode unit, but only one branch unit is working when multiple 2-issue decode units are combined. Each 2-issue decode unit decodes its own long instruction (64-bit) and raises high its own *done* signal when the last VLIW instruction in the program (*STOP* instruction) is executed and the last result is written back. When a core is configured as a larger issue-width core, the combined decode unit provides only one branch unit and one *done* signal. The other *done* signals are tied to logic low. The signal *issue_ctrl* controls the mechanism. For the 2-4-issue processor, the decode unit is reconfigured by loading a partial bitstream. Separate bitstreams are utilized to switch the two 2-issue cores to one 4-issue core or vice versa.

General-Purpose Register File We implemented the GR register files for the 2-4-issue and 2-4-8-issue processors utilizing BRAMs. The register files can provide access to multiple configurations of our reconfigurable processors. For the 2-4-issue processor, the register file is designed such that the single register file can provide access to a 4-issue core or two 2-issue cores at the same time. Register file for the 2-4-8-issue processor is depicted in Figure 4.4. It can provide access to an 8-issue core or two 4-issue cores or one 4-issue and two 2-issue cores or four 2-issue cores at the same time.

The register files are based on the version 3 design presented in Section 3.2.2, utilizing the 18 Kbits embedded BRAMs. Each BRAM is configured in simple dual port (SDP) mode with 1W1R port. In order to provide multiple ports, the BRAMs are organized into multiple banks and data is duplicated across various BRAMs. The register file for the 2-4-8-issue processor has 8W16R ports utilizing 128 BRAMs each providing 256 registers of 32 bits each. The distribution of the registers and ports for the different types of cores is presented in Table 4.1. Each of the active processor requires a maximum of 64 multiported registers requiring 6-bit address, but the combined register file has to have 256 registers requiring 8-bit address. Each BRAM in the register file has 8-bit address to provide 256 registers. The signal *issue_ctrl* and a small *control logic* are utilized inside the register file to generate the 7^{th} and 8^{th} bits of the BRAM addresses. The GR register file for the 2-4-issue processor has 4W8R ports utilizing 32 BRAMs each providing 128 registers of 32 bit each. If the processor is configured as a 4-issue core, all of the ports and the lower 64 registers are utilized. If the processor is configured as two 2-issue cores, half of the ports are utilized by the first core and the second half by the other core. The lower 64 registers are utilized by one core and the upper 64 by the other core. Each of the active processor requires a maximum of 64 multiported registers requiring 6-bit address, but the combined register file has to have 128 registers requiring 7-bit address. Each BRAM in the register file has 7-bit address to provide 128 registers. The signal *issue_ctrl* is utilized inside the register file to generate the 7^{th} bit for the BRAM addresses. By utilizing this mechanism, we avoided the register files to be reconfigured by loading the partial bitstreams and hence, reduced the size of the partial bitstreams required to alter the organization of the processors.

Branch Register File The VEX ISA specifies a 1-bit 8-element multiported BR register file for a multi-issue VLIW processor. The 2-issue, 4-issue, and 8-issue ρ-VEX processors require BR register files with 2W2R ports, 4W4R ports, and 8W8R ports, respectively. Since the size of this register file is small,

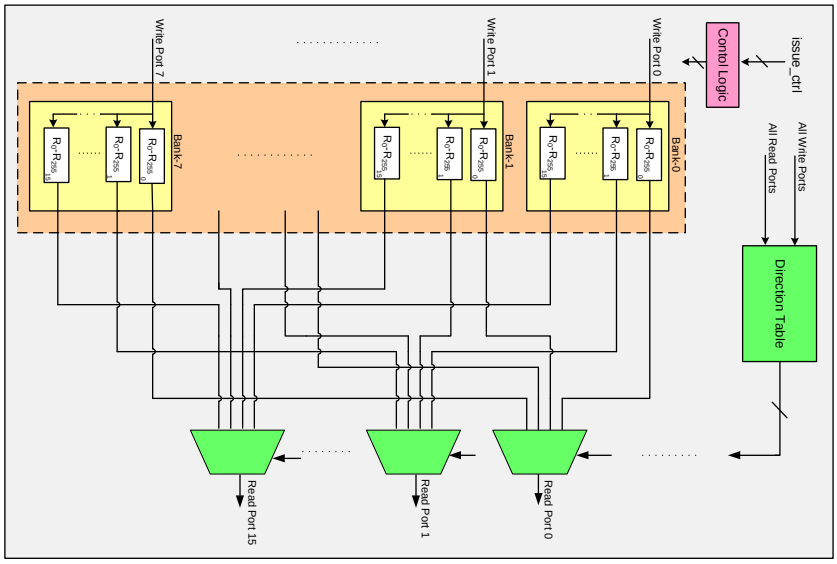


Figure 4.4: 256×32-bit 8W16R ports register file for the 2-4-8-issue processor.

it is implemented utilizing the FPGA's slice flip-flops and slice LUTs instead of BRAMs. For the 2-4-issue processor, we implemented a 16×1-bit BR register file with 4W4R ports. Utilizing the *issue_ctrl* signal, we partition the register file among the configured cores. When the processor is configured as one 4-issue core, all of the ports and the lower 8 registers are utilized. When the processor is configured as two 2-issue cores, half of the ports and the lower 8 registers make the BR register file for one core and the other half ports and the upper 8 registers make the BR register file for the second core. The signal *issue_ctrl* controls this mechanism. For the 2-4-8-issue processor, we implemented a 32×1-bit BR register file with 8W8R ports. The signal *issue_ctrl* is used to share the register file among the configured cores. The distribution of registers and ports for the register file is similar to that of the GR register file.

Backend The backend of the 2-4-issue and 2-4-8-issue adaptable processors remains fixed and does not change when the issue-width is changed. It consists of the execute and writeback units. Multiple 2-issue writeback units (four for the 2-4-8-issue processor and two for the 2-4-issue processor) are combined together. Each writeback unit can serve a 2-issue core and multiple writeback units can make a writeback unit for a larger issue-width core. Each lane of the writeback unit can write to its corresponding port on the GR and BR register files. Since these register files can handle the processor's issue-width by

Table 4.1: Distribution of registers and ports for the 256×32-bit 8W16R ports GR register file for the 2-4-8-issue processor.

Processor configuration	Write ports	Read ports	Registers
One 8-issue	0 - 7	0 - 15	0 - 63
Two 4-issue	0 - 3	0 - 7	0 - 63
	4 - 7	8 - 15	64 - 127
One 4-issue and two 2-issue	0 - 3	0 - 7	0 - 63
	4 - 5	8 - 11	64 - 127
	6 - 7	12 - 15	128 - 191
Four 2-issue	0 - 1	0 - 3	0 - 63
	2 - 3	4 - 7	64 - 127
	4 - 5	8 - 11	128 - 191
	6 - 7	12 - 15	192 - 255

themselves, the writeback unit does not need to take care of that, and hence, does not need reconfiguration/adjustment in order to combine or split the issue-slots. Additionally, the backend consists of all the execution units which are distributed across different issue-slots. For the 2-4-8-issue processor, 8 ALUs, 8 MULs, and 4 MEM/LS units make up the backend, while the backend for the 2-4-issue processor consists of 4 ALUs, 4 MULs, and 2 MEM/LS units.

4.1.3 Memory System

In this section, we explain how the instruction and data memories for our run-time adaptable processors can be set and reconfigured. Here, we only consider instruction memories that are locally connected to the cores. We do not discuss caches here. We discuss the memory system for the 2-4-8-issue processor only, while the 2-4-issue processor follows similar organization. Figure 4.5 depicts the memory organization for the 2-4-8-issue processor. Every 2-issue core has its own instruction memory to provide a 64-bit instruction per clock cycle. To generate the next address for an instruction memory, every 2-issue core has a *PC* in the fetch unit. If multiple 2-issue cores are combined to form a larger issue-width core, only one of the PCs is running and other PCs in that specific larger issue-width core are stopped. The *address generation unit (AGU)* receives input from all the PCs, and based on the *issue_ctrl* signal, generates the next addresses for all the instruction memories. The AGU keeps the next addresses for all the instruction memories in synch and drives them in lockstep according to the desired configuration scheme of the processor.

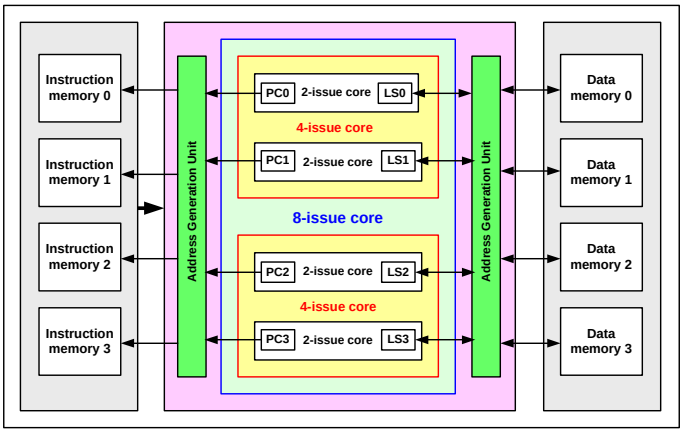


Figure 4.5: Instruction and data memories for the 2-4-8-issue processor.

Every PC can be initialized with a user-defined instruction address value. When the processor is configured as four 2-issue cores, each instruction memory receives its next address value from its corresponding PC. A program for a 2-issue core can be placed at any desired location in its corresponding instruction memory. Before executing the program, the PC of that core is loaded with the starting address of the program, and in the next cycle, the first instruction of the program is fetched in. When two 2-issue cores are combined to form a 4-issue core, the PC of the first core is active while that of the second core is stopped. Based on the *issue_ctrl* signal, the address generation unit drives the input addresses for the two instruction memories in lockstep. The program for the 4-issue core is split such that the two operations of the long instructions are placed in the first instruction memory, while the last two operations in the second instruction memory for the combined core. The partitioned program can be placed in the two instruction memories starting at the same or different location addresses. If the programs are placed at the different locations, the AGU should know the offset in the two addresses. The same technique is utilized when four 2-issue cores are combined to form an 8-issue processor.

Every 2-issue core has a MEM or load/store (LS) unit and a separate data memory. When multiple 2-issue cores are combined to form larger issue-width cores, the individual data memories can be combined together to provide a single larger data memory. The larger issue-width core can also utilize the additional LS units to increase the data transport from/to the memory. The *AGU* receives the effective addresses from the individual LS units, and based on the configuration bits, adapts the connection for the individual data memories.

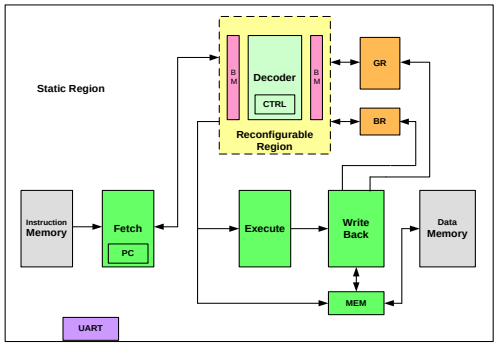
4.1.4 Mechanism for Issue-width Adjustment

Each of the 2-issue cores has an input signal called *run*. When this signal for a core is at logic *high*, the core starts fetching its VLIW instructions. When this signal is at logic *low*, the PC for that core is stopped. The *run* signal is also utilized to gate the source clock for a core. If a 2-issue core is not executing any application, it can be taken to a lower power mode by gating off its source clock, and hence, the dynamic power of that core is reduced resulting in a reduced total power consumption of the system. When a core finishes its execution, it raises its *done* signal. The *done* signal is utilized in order to schedule new code on a core. The signal *issue_ctrl* controls the organization and issue-width of the cores. For example, when the *issue_ctrl* signal is at logic *low*, the 2-4-issue and 2-4-8-issue processors behave as two and four independent 2-issue cores, respectively. The *issue_ctrl* controls the PCs for the cores that are combined to form a larger issue-width core and drives them in lockstep. It additionally controls the organization of the register files. In order to group or ungroup certain cores to change the issue-width, the selected cores are first stopped utilizing their *run* signals. In the next cycle, the *issue_ctrl* bits are modified to adjust the issue-width of the resulting core/cores. In the next cycle, the *run* signals are asserted and the cores start fetching their VLIW instructions. For 2-4-8-issue processor, only the signals *issue_ctrl*, *run*, and *done* are needed for the processor reconfiguration. For the 2-4-issue processor, apart from controlling the signals *issue_ctrl*, *run*, and *done*, a partial bitstream is also loaded for the processor reconfiguration. The signals *issue_ctrl*, *run*, and *done* are controlled through a configuration register.

4.1.5 Implementation Results

In this section, we present the implementation details and results for the 2-4-issue and 2-4-8-issue adaptable processors.

2-4-issue processor The 2-4-issue processor is implemented utilizing the Xilinx EAPR partial reconfiguration methodology [71]. The design is split into two regions, called *static* and *reconfigurable*, as depicted in Figure 4.6(a). The processor consists of frontend, backend, instruction memory, data memory, and a UART module. Except the decode unit in the frontend, all other modules in the frontend, the backend, memories, and UART are placed in the static region as they do not need partial dynamic reconfiguration. The decode unit is placed in the reconfigurable region. The two regions are connected



(a) Statically and partially reconfigurable regions.

Region	Slices	BRAMs
Reconfigurable region	560	0
Bus macros	230	0
Register file	1060	32
Backend	8265	0
Static region	9944	32

(b) Hardware utilization.

Figure 4.6: Design and hardware resource utilization for the 2-4-issue reconfigurable processor for the Xilinx Virtex-II Pro XC2VP30-7FF896 FPGA.

through *bus macros* [71]. Figure 4.6(b) presents the implementation results for the 2-4-issue processor. We used the Xilinx ISE version 9.2.04i_PR14 and the Virtex-II Pro *XC2VP30-7FF896* FPGA for the synthesis and implementation. The processor can run up to a maximum of 70 MHz.

Using the EAPR design methodology, partial bitstreams for the decode units are generated and downloaded to the FPGA. The partial bitstream size for the reconfigurable region is 59 Kbytes, and is about 24 times smaller than the full bitstream size which is about 1415 Kbytes. The width of internal configuration access port (ICAP) in the Virtex-II Pro and Virtex-4 FPGAs is 8 bits and 32 bits, respectively. The maximum frequency for the ICAP in the Virtex-II Pro and Virtex-4 FPGAs is 66 MHz and 100 MHz, respectively. The minimum time needed to switch from two 2-issue cores to one 4-issue core or vice versa is 0.893 and 0.148 milliseconds for Virtex-II Pro and Virtex-4 FPGAs, respectively. These values do not include the time needed for accessing the memory in which the bitstreams are placed. These represent the time needed for the SelectMAP or ICAP to configure an FPGA. At 50 MHz clock, these reconfiguration times translate to a total of 44650 and 7400 clock cycles for Virtex-II Pro and Virtex-4 FPGAs, respectively. It is to note that the Virtex-II Pro and Virtex-4 FPGAs have almost similar structure. For the purpose of illustration, we estimated the reconfiguration time for Virtex-4 FPGAs.

2-4-8-issue processor The 2-4-8-issue processor is implemented utilizing the virtual reconfiguration flow. All the required resources are available, and with the help of the *issue_ctrl* signal, the configuration and issue-width of the

processor can be adapted. There is no need for downloading partial bitstreams. Table 4.2 presents the implementation results for the 2-4-8-issue processor. We utilized the Xilinx ISE release version 13.2 and the Virtex-6 *XC6VLX240T-1FF1156* FPGA for the synthesis and implementation. Each 2-issue or any larger issue-width processor in our design can run up to a maximum clock frequency of 110 MHz. It is to note that Virtex-II Pro and Virtex-4 FPGAs have 4-input LUTs while Virtex-6 FPGAs have 6-input LUTs. The DSPs elements in Virtex-II Pro and Virtex-4 FPGAs are smaller compared to that in Virtex-6 FPGAs. Additionally, we designed a new MUL unit for the ρ-VEX processor, which is more efficient and requires less hardware resources compared to the old one. We utilized different families of the Xilinx FPGAs (Virtex-II Pro, Virtex-4 and Virtex-6) to show the effectiveness of our design under different FPGA families. For example, the register file implementations presented in Section 3.2 result in different hardware utilization for these different FPGAs.

4.1.6 Related Work

Voltron [72] combines small cores and the on-chip memory to make larger issue-width cores at run-time to exploit instruction, data, or thread-level parallelism. It exploits VLIW-style ILP by lock-stepping the individual cores like a multi-clustered VLIW processor. A network for inter-cluster communication is provided which is orchestrated by the compiler. *RAW* [73] has a grid of identical tiles connected through a mesh of scalar operand networks. Each title is a single-issue core with on-chip caches/memories, and using the operand network, intermediate register values can be transported. RAW supports instruction, data, and thread-level parallelism by using the software-controlled routing network between the tiles. *TRIPS* [74] is a reconfigurable architecture that enables the available out-of-order processing cores and the on-chip memory system to be configured and combined in different modes for instruction, data, or thread-level parallelism. TRIPS implements a custom ISA and micro-architecture, and relies heavily on compiler support for scheduling instructions

Table 4.2: Implementation results for the 2-4-8-issue processor for the Virtex-6 *XC6VLX240T-1FF1156* FPGA.

Module	Slice registers	Slice LUTs	DSP48E1s	RAMB18s
Register file	820	5887	0	128
Backend	988	8022	16	0
Processor	3187	16790	16	128

to extract ILP. *Core fusion* [75] provides mechanisms to combine small out-of-order cores to make larger issue-width cores at run-time or utilize them as independent smaller cores. A pair of instructions is used to fuse and split the available cores in order to exploit ILP and TLP. Core fusion is implemented in a simulator. *Smart memories* [76] has many processing tiles, each containing local memory, local interconnect, and a processor core. The user can program the wires, the memory, and the processor in order to match the architecture to the application. The *M-Machine* multiprocessor system [77] provides direct inter-processor communication channels between the register files of the available processor cores. By controlling the communication channels, the M-Machine can be used to exploit ILP and TLP. *XIMD* [39] is a superset of VLIW paradigm and can exploit both control flow parallelism as well as data parallelism. The XIMD architecture can dynamically partition its resources to support concurrent execution of multiple instruction streams. It has multiple FUs running in locked step, with each FU controlled by its own sequencer. The KAHRISMA architecture [42] utilizes different coarse-grained and fine-grained FUs. By means of a run-time adaptable inter-communication network, the FUs can be connected in different manners to emulate different processing modes (e.g. RISC and VLIW). The KAHRISMA ISA is comparable to clustered-VLIW processors, but its micro-architecture is similar to superscalar architectures with dynamic scheduling but without a dispatcher [43].

4.2 Run-time Reconfigurable Register File

Data in [29] show that increasing the number of read/write ports or increasing the issue-width of the VLIW processor results in an exponential increase in resources. Similarly, in [78], the design and implementation of a 3-issue VLIW microprocessor is presented. The processor datapath is 64-bit and it supports only 16 operations. The multiported register file for the processor containing 16, 64-bit registers each having 3W6R ports require 7172 Logic Cells of the Altera Stratix *EP1S25F1020C* FPGA, which is more than the area taken by the rest of the design. Figure 4.7 presents the hardware utilization (FPGA slices) for a 4-issue non-pipelined ρ-VEX processor (4 ALUs, 2 MUL, 1 MEM unit) and 4W8R ports register file version 1 for the Virtex-II Pro *XC2VP30* FPGA. From the figure, we can observe that when the number of registers is 64, the total number of slices utilized by the register file exceeds the total slices taken by all other modules of the processor. Because not all applications require 64 registers, implementing 64 registers for a ρ-VEX processor would mean a lot

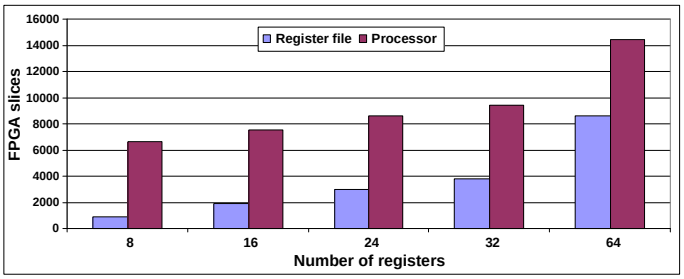
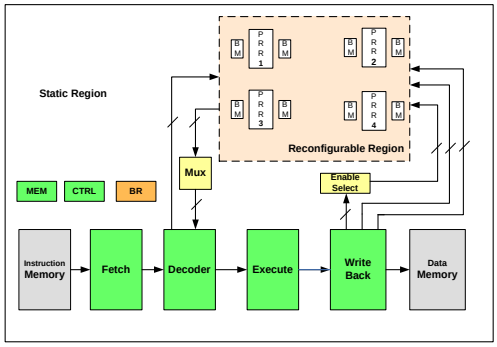


Figure 4.7: Virtex-II Pro FPGA's slice utilization for 64×32-bit 4W8R ports register file and 4-issue non-pipelined ρ-VEX processor.

of wasted resources as well as wasted power. On the other hand, implementing a smaller number of registers, for example, 8 would require less resources, but may degrade the performance of an application, as memory swapping (through load/store) may then be needed more frequently. We designed a run-time reconfigurable register file [79] and a ρ-VEX processor that supports partial dynamic reconfiguration allowing the creation of dedicated register files for different applications. Therefore, valuable area can be freed and shared with other implementations (such as timers, UARTs, another ρ-VEX core, etc.) on the same FPGA when not all of the 64 multiported registers are needed. Power consumption can be reduced by not configuring the un-necessary registers.

4.2.1 Case Study for 4-issue ρ-VEX Processor

For the 4-issue non-pipelined ρ-VEX processor, the total number of inputs/outputs (I/Os) of the register file is 460. Each register is 32-bit, having 4W8R ports. We utilized the Xilinx EAPR methodology [71] for designing our partial reconfigurable processor. For partial reconfiguration, we split the processor into two regions: *static* and *reconfigurable* as depicted in Figure 4.8(a). To simplify the design and quickly verify the idea of dynamically reconfigurable registers, we restrict the total number of registers to be 32, divided in 4 smaller register files or groups. The number of I/Os for each smaller register file is 424 that would cross the boundary between static and reconfigurable portions on the FPGA, and would require *bus macros* [71]. Four reconfigurable regions for register banks are connected to static region of the processor using asynchronous bus macros. The static region contains all of the processor modules namely, *fetch*, *decode*, *execute*, *writeback*, *memory unit*, *control unit*, *branch registers*, *instruction* and *data memories*, except the general-purpose register file, which is implemented in the reconfigurable region. The granu-



(a) Statically and partially reconfigurable regions.

Region	Slices
Bank of 8 registers	924
Bus macros	848
Static region	5727

(b) Hardware utilization.

Figure 4.8: Design and hardware resource utilization for the dynamically reconfigurable ρ-VEX processor. Apart from the slices, the static region also utilizes 14 *MULT18X18s*, and some BRAMs for instruction and data memories.

larity level is 8 for the reconfigurable register file, i.e., registers can be added in the increments of 8 up to maximum of 32 registers. Using the EAPR design methodology, partial bitstreams for the register banks are generated and can be downloaded to the FPGA. An application can be profiled with the VEX toolchain and the optimum number of registers can be determined. This information can then be communicated to the decoder by means of a custom instruction, which can be used to direct the reconfiguration controller to reconfigure the required number of registers before the application starts execution.

Figure 4.8(b) presents the hardware utilization for our reconfigurable processor. For implementation, we used the Xilinx Virtex-II Pro *XC2VP30* FPGA and the ISE release version 9.2.04i_PR14. The size of the partial bitstream for a reconfigurable bank of 8 registers and the full bitstream are 85 Kbytes and 1415 Kbytes, respectively. The time needed to configure a bank of 8 registers is 1.29 milliseconds, while that for the full bitstream is 21.44 milliseconds. The reconfiguration time mentioned does not include the time needed for accessing the memory in which the bitstreams are stored. It is the time needed for the *SelectMAP* or *ICAP* to configure the FPGA.

4.3 Run-time Task Migration

Building on the interrupt system presented in Section 3.3, we developed a run-time task migration scheme [60] for the 2-4-8-issue multi-core processor. With the task migration scheme, a code running on a core can be shifted at run-time

to a larger or a smaller issue-width core for increasing the performance or reducing the power consumption of the whole system, respectively. The cores can be combined or split even when they are not idle. All the cores can be utilized in an efficient manner, as a core needed for a specific job can be freed at run-time by shifting its running code to another core. Figure 4.9 depicts the timeline for a task migration example. At a time instance, *core1*, a 2-issue core is running *task1* and requires time *t1* to finish the task. *Core2*, which is a 4-issue core is running *task2* and requires time *t2* to finish the task. At *t2*, *core2* is free, and in a time Δt, *task1* can be migrated from *core1* to *core2*. Since *core2* is a larger issue-width core, it can boost the performance and hence, finish *task1* at $t3 < t1$. Similarly, shifting from a larger issue-width core to a smaller issue-width core at run-time and turning off the larger issue-width core can reduce the power consumption.

4.3.1 Design of the Task Migration Scheme

We implemented a run-time task migration scheme for the 2-4-8-issue adaptable multi-core processor as depicted in Figure 4.10. The methodology utilized in Section 4.1 is that cores can only be combined or split when they are idle (i.e., have finished their current execution). In this section, we present another level of control for the 2-4-8-issue adjustable processor with the development of interrupt system. Every 2-issue core or the combined larger issue-width cores can now be individually interrupted. Each core is now able to pass on its environment (execution state) to another core of the same or different type. We can now combine or split cores that are even not idle. We implemented an environment shifting or task migration mechanism for the cores utilizing the interrupt system. The environment shifting is needed in different situations. For example, if a larger issue-width core becomes available, it might be needed to switch an application running on a smaller issue-width core to the larger issue-width core for performance reasons. Similarly, one might need to switch a code running on a larger issue-width

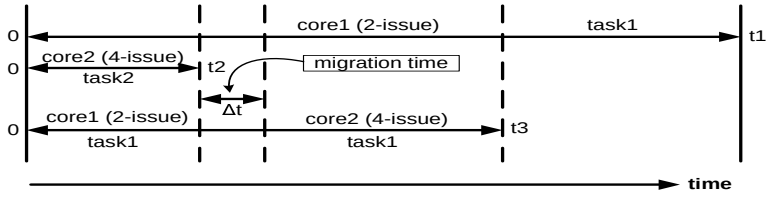


Figure 4.9: A task migration example.

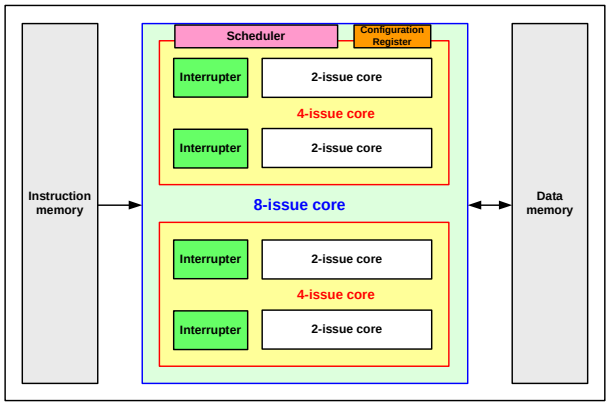


Figure 4.10: The 2-4-8-issue adaptable processor with the task migration support.

core and turn the larger issue-width core off to reduce the dynamic power consumption of the whole system at run-time.

The issue-width or the organization of the 2-4-8-issue processor can be changed by writing dedicated bits to the configuration register of the processor. This register can be accessed by decoding a custom instruction on the processor. This instruction can be manually placed at the specific points in the executable code, where an issue-width change is required. The configuration register can also be implemented in the global space accessible to other dedicated hardware/software controllers. In this case, the reconfiguration process can be initiated by some external agents/controllers based on certain run-time metrics such as hardware utilization, power/energy considerations, arrival of other tasks, cache related statistics, We utilized the *generic binaries* scheme [80] to generate the binary code for our variable issue-width processor. In this scheme, an application is compiled such that the same binary code can be executed correctly on different issue-width VLIW processors with some performance degradation. The advantage is that the same binaries can be utilized when switching the processor issue-width and there is no need for loading/accessing multiple binaries. More information about the generic binaries scheme can be found in Section 6.4.

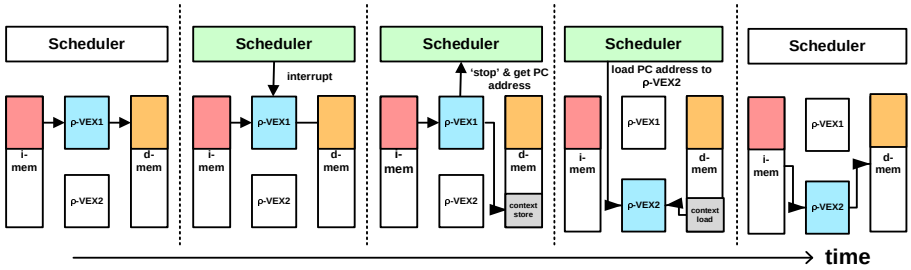
Figure 4.11 depicts the mechanism for migrating a task from one type of core to another (say ρ-VEX1 to ρ-VEX2) in the 2-4-8-issue adaptable processor. Here ρ-VEX1 and ρ-VEX2 could be any issue-width cores (2-issue, 4-issue, or 8-issue). A hardware scheduler controls the process of task migration. When shifting a code running on ρ-VEX1 to ρ-VEX2, the scheduler performs the following steps as depicted in Figure 4.11:

- generate an interrupt on ρ-VEX1 core
- a special ISR is called and executed on ρ-VEX1 that stores the context into the data memory (shared memory accessible to all cores), and the PC address with respect to a defined switching point where the currently running program was stopped is recorded
- reconfigure the issue-width of the core (merge or split cores) if required (now called ρ-VEX2) by changing the configuration register values
- generate an interrupt on ρ-VEX2 core
- a special ISR is called and executed on ρ-VEX2 that restores the context from the data memory
- load the PC address into ρ-VEX2
- start ρ-VEX2 to resume execution of the remaining code

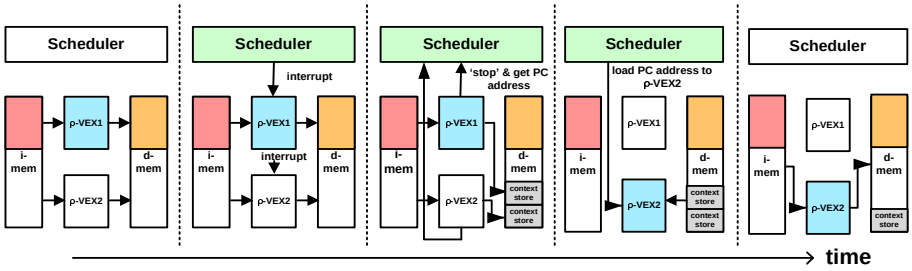
Here, we only store/restore the content of the GR and the BR register files. We implement the stack in the data memory accessible to both cores and hence, do not store/restore the stack while moving the task from one core to another. The cores should know the address in the data memory where the stack is implemented, and it is done at compile/assemble time. This reduces the task migration time between different cores.

4.3.2 Implementation Results

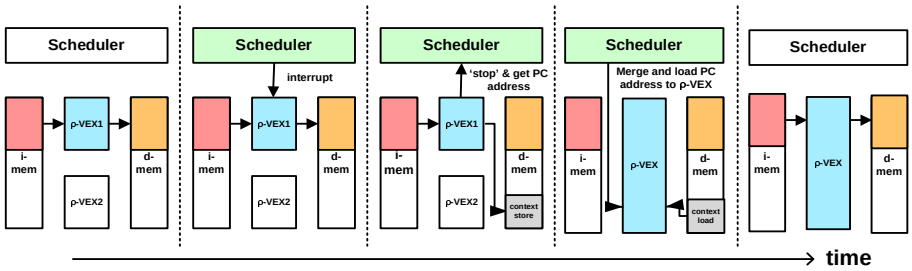
Table 4.3 presents the implementation results for the 2-4-8-issue processor with the task migration support. We utilized the Xilinx ISE release version 13.2 and the Virtex-6 *XC6VLX240T-1-FF1156* FPGA for the synthesis and implementation. The maximum frequency is 110 MHz. We utilized version 3 of the interrupts system (software instructions switching context method) presented in Section 3.3.2. This is the most standard version, requires minimal hardware changes, and has an interrupt response time of 76 cycles. The task migration from one core to another requires a total of 155 cycles (76 cycles for storing the first core's context, 1 cycle for accessing it's PC, 1 cycle for reconfiguring the issue-width, 76 cycles for restoring the stored context to the newly configured core, and 1 cycle for loading it's PC).



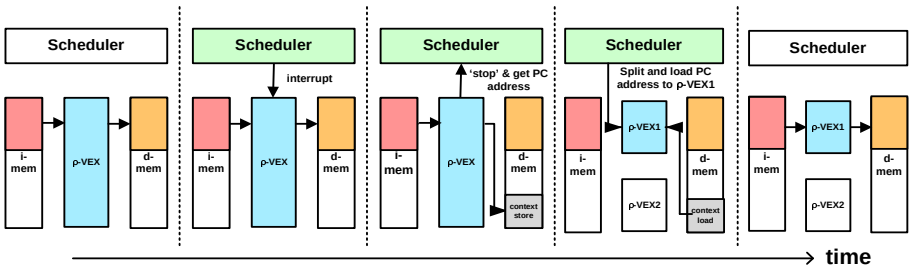
(a) From ρ-VEX1 to ρ-VEX2, where ρ-VEX2 is idle.



(b) From ρ-VEX1 to ρ-VEX2, where ρ-VEX2 is not idle.



(c) From ρ-VEX1 to ρ-VEX, where the ρ-VEX is formed by merging ρ-VEX1 and ρ-VEX2.



(d) From ρ-VEX to ρ-VEX1, where ρ-VEX1 and ρ-VEX2 are formed by splitting the ρ-VEX.

Figure 4.11: Mechanism for task migration in the 2-4-8-issue adaptable processor.

Table 4.3: Implementation results for the 2-4-8-issue adaptable multi-core processor with the task migration support for the Virtex-6 *XC6VLX240T-1-FF1156* FPGA.

2-4-8-issue processor	Slice registers	Slice LUTs	DSP48E1s	RAMB18s
Without task migration	3187	16790	16	128
With task migration	3754	17520	16	128

4.3.3 Related Work

Task migration is used in multi-core systems to balance workload and network congestion. An unbalanced workload can result in excessive power consumption and thermal hot-spots and unbalanced network congestion can result in missed deadlines. Different task or process migration mechanisms and algorithms are presented in [81] [82]. The authors in [83] discuss different policies for real-time task migration in embedded multi-core architectures. The impact of task migration on embedded soft real-time streaming multimedia applications is assessed in [84]. Here, a middleware infrastructure at operating system (OS) level supporting dynamic task allocation for non-uniform memory architectures (NUMA) is presented. A context-aware run-time adaptive task migration mechanism to reduce the task migration latency in multi-core architectures is presented in [85]. A task migration between two cores results in cache warm-up overheads on the target core, which can result in missed deadlines for tight real-time schedules. A micro-architectural support for migrating cache lines that enables real-time tasks to meet their deadlines in the presence of task migration is proposed in [86].

Policies for task migration to control the thermal characteristics in multi-core systems are presented in [87] [88]. Energy-efficient real-time task scheduling and migration in multiprocessor systems is discussed in [89] [90]. The authors in [91] discuss the impact of task migration in network-on-chip based MPSoCs for soft real-time systems. Techniques to selectively migrate the code/data to reduce communication energy in embedded MPSoCs are presented in [92]. The authors in [93] discuss a fault-and-migrate mechanism for asymmetric multi-core architectures which traps a fault when a core executes an unsupported instruction, migrates the faulting thread to a core that supports the instruction, and allows the operating system to migrate it back when load balancing is necessary.

4.4 Simultaneous Reconfiguration of Issue-width and Instruction Cache

Applications with higher ILP perform better when run on a larger issue-width processor. Figure 4.12 depicts the IPC for some applications from different benchmark suites (MiBench [2], PowerStone [3]) for 2-issue, 4-issue, and 8-issue VLIW processors with a single load/store unit. As depicted in the figure, the IPC increases with the issue-width for applications with more ILP. Specializing a cache for a processor may improve the performance or energy consumption for one benchmark, but may perform poorly across others [94]. Studies have shown that more than half of the chip die is reserved for the on-chip caches and that the energy consumption in cache systems accounts for more than 50% of the total energy consumption [3] [95] [96] [97] [98]. Table 4.4 presents the instruction cache (I-cache) parameters for some commercial/research VLIW processors. As can be observed, there is a wide variation across different cache parameters (associativity, cache size, and line size). Compared to having a fixed cache, reconfiguring the cache for a processor at run-time can reduce the execution time and/or power/energy consumption for different benchmarks [94] [96] [98] [99] [100] [101] [102]. Compared to reconfiguring only the cache, reconfiguring the “issue-width + I-cache" together can further improve the execution time, energy consumption, and/or EDP.

In this section, we present a setup to analyze the effect of simultaneous reconfiguration of issue-width and I-cache for the 2-4-8-issue reconfigurable processor [103]. Notice that if different “issue-width + I-cache" configurations have the same execution times, but reduced energy consumptions or vice versa, it may be beneficial to reconfigure the core issue-width, the cache, or both. The 2-4-8-issue processor can be configured to be 2-issue, 4-issue, or 8-issue at run-time. The unused issue-slots are clock-gated to reduce dynamic power consumption of the processor. We considered an instruction cache that can be reconfigured in terms of *associativity*, total *cache size*, and *line size*. We utilized the VEX simulator [1] to simulate different “issue-width + L1 I-cache" configurations. For energy calculation, we utilized the CACTI 6.5 [104] and the *Synopsis Design Compiler* (Synthesis-E-2010.12-SP1) and targeted 90 nm technology. We utilized the MiBench [2], PowerStone [3], and custom (16 small applications/kernels from different domains) benchmark suites. The results of the analysis in terms of performance, dynamic energy consumption, and EDP are presented in Chapter 6. In this section, we only discuss the characteristics of the simultaneously reconfigurable system.

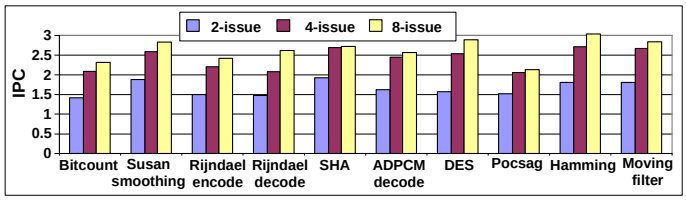


Figure 4.12: Instructions per cycle (IPC) for different applications [2] [3].

4.4.1 Related Work

The impact of cache parameters such as total size, line size, associativity, replacement policies, etc., on performance and energy consumption for different levels of caches (L1 and L2) has been widely reported. A reconfigurable cache memory with heterogeneous banks to reduce the cache size and hence the power consumption at run-time is presented in [109]. Reconfigurable aspects of the cache system for the TMS320C6211 processor are discussed in [110]. The 4-way unified L2 cache can be used as either mapped RAM or as 1, 2, 3, or 4 ways cache. Each way or bank is 16 Kbytes. A reconfigurable data cache design with a hardware-adaptive line size for miss rate and memory traffic reduction is presented in [99]. The paper does not discuss energy consumption.

Selective cache ways [96] provides the ability to disable a subset of the ways in a set-associative cache to reduce the energy consumption for little performance overhead. A mechanism for tuning cache ways and voltage scaling for embedded system platforms to reduce energy consumption is presented in [111]. Way predictive set-associative caches [101] [112] provide the ability to reduce energy consumption at the expense of longer average access time. The design

Table 4.4: Typical instruction cache parameters for some famous VLIW processors.

Processor	Issues	Assoc.	Size (Kbytes)	Line size (bytes)
TriMedia TM32A [105]	5	8	32	64
TriMedia TM3270 [105]	5	8	64	128
TMS320C6211 [49]	8	1	4	64
ST231 [46]	4	1	32	64
ST240 [106]	4	4	32	64
Transmeta TM5400 [51]	5	8	64	-
Fujitsu FR450 [107]	2	2	32	32
CoreVA [108]	4	1	16	64

presented in [102] dynamically divides the cache arrays into multiple partitions that can be used for different processor activities to increase the performance. A novel set and way management cache architecture for efficient run-time reconfiguration (Smart cache) is presented in [113], providing reconfigurability across cache size and associativity. A hybrid selective-sets-and-ways cache organization is proposed in [114] that always offers equal or better resizing granularity than both the selective-sets and selective-ways organizations. The impact of line size on energy consumption and performance for instruction and data caches is presented in [100]. Designs of configurable caches where all the three parameters (associativity, cache size, and line size) can be configured are presented in [98] [115]. It must be noted that all these papers present results for cache configurations with fixed issue-width processors.

The commonly used commercial VLIW processors such as the *TriMedia* series from NXP, *ST231* from STMicroelectronics, *TMS320C611* from Texas Instruments, *Crusoe* from Transmeta, and the *FRxxx* series from Fujitsu all utilize a fixed issue-width. Reconfiguring the issue-width at run-time improves the performance of applications with higher ILP. *Core fusion* [75], *TRIPS* [74] and *Voltron* **[72]** combine small cores and the on-chip memory to make larger issue-width cores at run-time to exploit the instruction, data, or thread-level parallelism. *Smart memories* [76] is a reconfigurable architecture capable of merging in-order RISC cores to form a VLIW machine. These studies focus only performance/speedup results for the available configurations of the system but do not discuss the energy consumption or EDP.

4.4.2 Characteristics of the Reconfigurable Processor

Figure 4.3(b) depicts the general view of the 2-4-8-issue adaptable VLIW processor. The processor can be configured to be 2-issue, 4-issue, or 8-issue with different number of MEM/LS units, but for this analysis, we kept the number of MEM units to be 1 for every type of the processor issue-width. This is done in order to keep the data cache same for every type of processor issue-width. The issue-width is changed in a single cycle after the reconfiguration bits are written to a *configuration register*. Additionally, these bits are also utilized to clock-gate the unused FUs and parts of the processor system to reduce the dynamic power consumption. For this analysis, the processor supports only single-tasking computation. Multitasking or multi-threading support is not available. When an application starts executing, it is allowed to finish completely and then a new application is started. Hence, we do not need any complex mechanisms for task pre-emption, and the design becomes very simple.

The reconfiguration is needed per application basis. The request to change issue-width remains pending until the currently running application finishes execution. The request to change issue-width for a new application can be communicated by decoding a custom instruction that can be placed at the end of the currently running code or at the start of the new application's code. The custom operation writes the required bits to the configuration register which triggers the process of reconfiguration.

4.4.3 Characteristics of the Reconfigurable Instruction Cache

Our instruction cache architecture is based on [115] and includes three parameters: *cache associativity*, *cache size*, and *line size*. The cache reconfiguration is done in a single cycle after the *cache configuration register* is written. Because the processor does not support multi-tasking, the cache reconfiguration is required only when application changes. There is no need for run-time methods/policies, no cache flushing, no reconfiguration overhead, and hence, the cache reconfiguration time is reduced. Information about the best configuration (issue-width + I-cache) can be stored in the program executable and written to the issue-width and cache configuration registers before the application starts execution. According to Table 4.4, there is a wide variation across the cache parameters, therefore, we utilized the following parameters for our reconfigurable cache.

- Cache associativity: 1/2/4/8 ways
- Cache size: 4/8/16/32 Kbytes
- Cache line size: 16/32/64 bytes

The total cache (in all parameters) is available to all types of the configured issue-width cores. Following are the reconfiguration methods for the considered cache parameters.

- **Cache Associativity; Way Concatenation:** For reconfiguration of cache associativity, the way concatenation technique is used [115]. The base cache includes 8 banks that can operate as 8 ways. By writing to the cache configuration register, the ways can be effectively concatenated, resulting in a 4-way, 2-way, or 1-way (direct-mapped) 32 Kbytes cache.
- **Cache Size; Way Shutdown:** For reconfiguration of cache size, the way shutdown technique is used. With way shutdown, the 32 Kbytes 8-way

cache can be reconfigured as a 16 Kbytes cache that can be either 4-way, 2-way or direct mapped, an 8 Kbytes cache that can be either 2-way or direct mapped, or a 4 Kbytes direct mapped cache.

- **Cache Line Size; Line Concatenation:** For reconfiguration of line size a base physical line size of 16 bytes is implemented, with larger line sizes implemented logically as multiple physical lines [115]. By writing to the cache configuration register, line size can be reconfigured as either 16, 32, or 64 bytes.

Based on the previous three mentioned methods, there are only 30 cache configurations possible that are practically implementable. The remaining cache configurations are not possible due to the hardware design limitations. Hence, our cache configuration space is 30.

4.4.4 Energy Estimation

The following equation is utilized to estimate the total dynamic energy consumption of the I-cache including both the hit and miss energies.

$$\begin{aligned} Cache\_Energy &= Accesses * Energy/access + Misses * \\ &\quad Energy/miss \\ &= Accesses * Energy/access + Misses * \\ &\quad Kmiss * Energy/access \\ &= (Accesses + Kmiss * Misses) * \\ &\quad Energy/access \end{aligned} \tag{4.1}$$

Kmiss is a factor representing a multiple of the cache hit energy consumption. According to [115] which takes into account the energy consumption from the complete instruction memory hierarchy including the external memory, the value of *Kmiss* ranges from 50 to 200. Here, we consider the *Kmiss* to be 50. For our analysis, there are 30 I-cache configurations and 3 issue-width configurations; hence, the total search space for each application is 90 “issue-width + I-cache" configurations. Each application is simulated 90 times utilizing the VEX toolchain [1] to generate *total memory accesses*, *cache hits*, *cache misses*, and *execution cycles* statistics. Using equation 4.1, we calculated the I-cache energy consumption for each application with 90 different configurations. The cache energy per access is obtained from

CACTI 6.5 [104]. For calculating the processor energy consumption, we utilized the *Synopsis Design Compiler* (Synthesis-E-2010.12-SP1) to get the average power consumption for 90 nm technology. We then calculated the processor energy consumption for all applications with the following equation.

$$Processor\_Energy = Power\_consumed * Cycle\_time * Execution\_cycles \quad (4.2)$$

The total energy consumption and EDP are calculated as follows.

$$Total\_Energy = Processor\_Energy + Cache\_Energy \quad (4.3)$$

$$EDP = Total\_Energy * Execution\_cycles \quad (4.4)$$

Execution_cycles, Total_Energy, and EDP for each benchmark application with 90 different "issue-width + I-cache" configurations are calculated and then analyzed in Section 6.5.

4.5 Summary

In this chapter, we presented the design and implementation of two run-time adjustable issue-slots multi-core processors. The processors have multiple (two for the 2-4-issue processor and four for the 2-4-8-issue processor) 2-issue cores, each of which can run independently. If not in use, each core can be taken to a lower power mode by gating off its source clock. Multiple 2-issue cores can be combined at run-time to form larger issue-width VLIW cores. Other than the issue-width, the type and number of different FUs, and the size of the multiported GR register file can also be configured at run-time. The processors can target a variety of applications having instruction and/or data level parallelism. Additionally, the chapter presented a run-time task migration scheme for the 2-4-8-issue processor. With the task migration, the cores can be utilized more efficiently. A task running on a core can be migrated to a larger or a smaller issue-width core to increase the performance or reduce the power consumption, respectively. Finally, we presented a system for the simultaneous reconfiguration of issue-width and instruction cache for the 2-4-8-issue processor, where along with the issue-width (2/4/8), the instruction cache can be reconfigured in terms of associativity, cache size, and line size.

Note.

The content of this chapter is partially based on the following papers:

S. Wong, **F. Anjam**, and M.F. Nadeem. Dynamically Reconfigurable Register File for a Softcore VLIW Processor. In *Design, Automation, and Test in Europe Conference (DATE)*, pp. 969–972, 2010.

F. Anjam, M. Nadeem, and S. Wong. A VLIW Softcore Processor with Dynamically Adjustable Issue-slots. In *International Conference on Field Programmable Technology (FPT)*, pp. 393–398, 2010.

F. Anjam, M. Nadeem, and S. Wong. Targeting Code Diversity with Run-time Adjustable Issue-slots in a Chip Multiprocessor. In *Design, Automation and Test in Europe Conference (DATE)*, pp. 1358–1363, 2011.

F. Anjam, Q. Kong, R.A.E. Seedorf, and S. Wong. A Run-time Task Migration Scheme for an Adjustable Issue-slots Multi-core Processor. In *International Symposium on Applied Reconfigurable Computing (ARC)*, pp. 102–113, 2012.

F. Anjam, Q. Kong, R.A.E. Seedorf, and S. Wong. On the Implementation of Traps for a Softcore VLIW Processor. In *HiPEAC Workshop on Reconfigurable Computing (WRC)*, 2012.

F. Anjam, L. Carro, S. Wong, G.L. Nazar, and M.B. Rutzig. Simultaneous Reconfiguration of Issue-width and Instruction Cache for a VLIW Processor. In *International Conference on Embedded Computer Systems: Architecture Modeling and Simulation (SAMOS)*, pp. 183–192, 2012.

5

Configurable Fault Tolerance

High reliability *and dependability of processing systems require the implementation of fault tolerance techniques. Fault tolerance can be achieved utilizing hardware, software, or hybrid approaches. In this chapter, we present configurable fault tolerance mechanisms for the ρ-VEX processor. Separate techniques are employed to protect different modules of the processor from single event upsets (SEU) errors. Parity checking is utilized to detect errors in the instruction and data memories and the GR register file, while triple modular redundancy (TMR) approach is employed for all the synchronous flip-flops (FFs). At design-time, a user can choose between the standard non fault-tolerant design, a fault-tolerant design where the fault tolerance is permanently enabled, and a fault-tolerant design where the fault tolerance can be enabled and disabled at run-time. These options enable a user to trade-off between hardware resources, performance, power consumption, and reliability. Following is the contribution of the chapter.*

- *A hardware-based configurable fault tolerance technique is presented for the ρ-VEX processor. The fault tolerance can be included/excluded in the processor at design-time and/or enabled/disabled at run-time.*

The remainder of the chapter is organized as follows. Section 5.1 presents the introduction and motivations. Section 5.2 discusses the related work. The base processor is briefly introduced in Section 5.3. The fault-tolerant design of the ρ-VEX processor is presented in Section 5.4. Experimental results are discussed in Section 5.5. Finally, the chapter is summarized in Section 5.6.

5.1 Introduction and Motivations

When the data path of a processor gets larger and complex, the probability of errors (such as radiation-induced soft errors) also increases. Because VLIW processors can provide high performance at low power, they are gaining widespread utilization not only in general-purpose embedded systems but also in safety-critical systems such as biomedical, space, military, communication, industrial, and automotive systems. Therefore, it is important to employ fault-tolerant techniques in order to guarantee high reliability and dependability of the safety-critical systems. Run-time detection plays an important role in dependable systems, where it is needed that the computed data is either correct or an error signal is generated whenever there is a possible error.

In this chapter, we present configurable fault-tolerant techniques [116] for the ρ-VEX processor. The processor is implemented in VHDL and the fault tolerance techniques are implemented at hardware level. The processor employs different fault tolerance techniques such as parity checking and TMR to increase the reliability and dependability of the system. The processor is implemented in a Xilinx Virtex-6 FPGA as well as synthesized to 90 nm ASIC technology. Apart from the general parameters such as the issue-width, number of FUs, etc., the fault tolerance is also configurable. At design-time, users can choose to implement a processor with no fault tolerance, a processor with the fault tolerance permanently enabled, or run-time reconfigurable. The permanently enabled and the run-time reconfigurable designs consume almost similar dynamic power. The advantage of the latter design is that the fault tolerance can be disabled at run-time, resulting in reduced dynamic power consumption. The fault tolerance can be enabled/disabled by executing an instruction on the processor. For applications which can tolerate some bit flips such as audio/video decoding, the fault tolerance can be disabled at run-time to reduce the dynamic power consumption. On the other hand, applications which are susceptible to even a single bit flip such as sending/receiving DTMF tones on a mobile device or doing some security related processing can enable the fault tolerance at run-time to temporally increase the reliability. The configurable processor provides a trade-off for hardware resources, performance, power consumption, and reliability.

Since one of the main purposes of this chapter is to evaluate power consumption for different fault-tolerant designs, therefore, we include results for ASIC implementation as well. In case of ASIC, the change in area and power consumption can be observed very clearly when a circuit is triplicated for the TMR scheme. Area and power consumption increase almost linearly with triplicat-

ing a circuit. When a circuit is triplicated in FPGA, the require area (Slices, LUTs, etc.) and power consumption may not increase linearly because some part of the circuit may also be accommodated in the already utilized area for the base circuit.

5.2 Related Work

Recently, fault tolerance for microprocessor systems is gaining increasing importance. Transient errors are considered as the main source of errors in processor systems. Different on-line detection and mitigation techniques are proposed to detect and correct transient error faults. These techniques are mainly based on redundancy approaches. Here, instructions are replicated, re-computed, and then results are compared for checking errors. Mainly, there are two approaches for redundancy; software-based and hardware-based.

A software-based redundancy approach utilizes a compiler to duplicate/triplicate instructions. This increases code size and power consumption and reduces performance [117]. The advantage is that no hardware modification is needed. Compiler-based software redundancy schemes with increased code size and performance degradation are presented in [118] [119]. Similar techniques for VLIW and superscalar processors are discussed in [120] [121] [122]. A software method to detect transient and common-mode faults in statically-scheduled VLIW processor in presented in [123].

A hardware-based redundancy approach requires changes to the architecture and additional hardware for managing replication, re-computation, and comparing results to detect errors. The advantage is that there is no need to change the code or the compiler, and that there is little or no performance degradation and no code size overhead. At the hardware level, one solution is to replicate the complete processor system and then implement a majority voter to select between the three results [124] [125]. In this case, there is no need to change the processor architecture, with the disadvantage that a fine-grain control over instruction-level checking is not possible. Another solution is to modify the architecture, implement additional FUs and other control hardware to perform the execution of replicated instructions [126] [127] [128]. A technique that utilizes additional FUs to detect and correct transient errors in combinational logic is presented in [129]. The author in [130] triplicates the sequential elements in the processor to detect and correct SEU errors. Recently, hybrid approaches (software and hardware) for error detection and correction were presented in [129] [131].

5.3 The Base ρ-VEX Processor

As discussed in Chapter 3, different parameters of the ρ-VEX processor such as the issue-width, the number and type of different FUs, supported instructions, memory-bandwidth, register file size etc., can be chosen at design time. The processor is a 5-stage pipelined processor consisting of *fetch*, *decode*, *execute0*, *execute1*/*memory*, and *writeback* stages. The base processor utilized for this chapter can be configured to be 2-issue, 4-issue, or 8-issue. Each type of core has a single load/store (*MEM*) unit and the same number of *ALUs* as the issue-width. The 2-issue, 4-issue, and 8-issue cores have 2, 2, and 4 *MUL* units. The processor has a 64×32-bit multiported GR register file and an 8×1-bit multiported BR register file. As discussed in Section 3.2, the GR register file is one of the most complex and resource consuming modules, therefore, it is implemented in three different mechanisms to evaluate resource utilization, performance, and power consumption characteristics. For ASIC, it is implemented with FFs and other combinational resources, while for FPGA, it is implemented with dual-port synchronous BRAMs, as well as with look-up tables (LUTs) and FFs.

5.4 The Fault-Tolerant ρ-VEX Processor

Single event upset (SEU) errors effect a memory cell or FF. It is a bit flip caused by a charged particle. The noise induced by some radiation when exceeds the threshold voltage, a bit flip may occur. Due to wire process shrinking, the threshold voltage is decreasing, and hence, electronic systems are becoming more susceptible to SEUs. When an SEU occurs in a memory (storage or configuration), it is called as *permanent error*. When it occurs in a flip-flop, it is referred to as a *transient error*. To recover from the permanent error, reconfiguration or re-loading of the configuration data to the configuration memory is required. For a memory used as a general storage (e.g., instruction memory), the permanent error could be checked and corrected by parity checking and some error correcting code (ECC). TMR technique is used widely to recover from a transient error. When TMR mitigation techniques are adopted, the same circuit is triplicated and a majority voter is implemented between the three computed results. Hence, a single fault occurring in one part of the TMR circuit is protected as the result is obtained from the other two circuits.

For this chapter, we consider SEU errors that occur due to a direct hit in a FF or a memory element used as a general storage (instruction and data memories,

and the GR register file). We do not consider the FPGA configuration memory, and assume that it is protected by other techniques. According to [132], the probability that SEU errors in combinational logic can propagate to a register on a clock is very low, therefore, we do not consider such permanent SEU errors in combinational logic. The ρ-VEX processor utilizes two types of sequential cells for its implementation: synchronous BRAMs for instruction/data memories and the GR register file (FPGA implementation), and FFs used for other storage such as general registers, pipeline registers, state machines, and status/control functions. We employ different SEU protection techniques for BRAMs and FFs. The hardwired BRAMs in the Xilinx and Altera FPGAs provide an extra bit per byte of data which can be used as a parity bit. Hence, for a 32-bit word, up to 4 parity bits are available and can be used without increasing the number of BRAMs. In case of an ASIC, additional area is required to implement parity bits in instruction and data memories. Following, we discuss different modules of the fault-tolerant ρ-VEX processor which utilize different error protection techniques.

5.4.1 Instruction Memory

For the ρ-VEX processor, each operation called syllable is encoded in a 32-bit word. Multiple syllables are combined to make a long instruction which is executed every clock cycle. The instruction width for a 2-issue, 4-issue, and 8-issue ρ-VEX processor is 64-bit, 128-bit, and 256-bit, respectively. Our design provides configurable number of parity bits (1, 2, or 4) per 32-bit instruction (syllable). Hence, for every 8 bits of instruction, a parity bit is available. The parity bits are statically calculated by *XOR* operations in the assembler tool and stored along with the instructions in the dedicated parity bits of the memory. Instructions are read and passed through the fetch stage to the decode stage. The parity bits are checked in the decode stage in parallel with instruction decoding to minimize the timing overhead. If a parity error is detected for an instruction, the fetch and decode stages are flushed, and the pipeline is halted. The correct instruction can then be copied from the higher level memory (Flash card, on-board memory, etc.) to the local instruction memory, and the pipeline can then be restarted.

5.4.2 Data Memory

The data width of the ρ-VEX processor is 32-bit whatever the issue-width may be. The data memory is implemented with BRAMs. Additional bits are

utilized as parity bits. Because the ISA has memory operations that can operate on words, half-words, and bytes, therefore, we utilized 1 parity bit per byte of the data. Initially, parity bits are generated statically in the assembler tool and placed along with data in the external memory. During initialization, the data and the parity bits are copied from the external memory to the local data memory. During a store operation, the parity bits are calculated and written to the data memory together with the new data. The parity bits are generated in the MEM unit which resides in the execute0 stage. During a load operation, a data word is read from the data memory along with the parity bits. The parity of the data word is checked in the writeback stage before writing the word to the GR register file. If there is a parity error, a data error trap is generated and the pipeline is halted. The simplest method to recover from this error is to reload the whole data memory for the program from the external memory and start the program from the beginning. Other complex error recovery methods such as roll back to the instruction which modified the data location may also be considered but implementing such methods are out of scope of this thesis.

5.4.3 GR Register File

As discussed in Chapter 3, the 2-issue, 4-issue, and 8-issue ρ-VEX processors require GR register files with 2W4R ports, 4W8R ports, and 8W16R ports, respectively. In Section 3.2, we presented different implementations for the register file in order to evaluate resource utilization, performance, and power consumption. Table 5.1 presents the details of these implementations. GR register file version 1 is a direct behavioral implementation and utilizes the FPGA's configurable LUTs and FFs. It implements large multiplexers to provide multiple read and write ports. For ASIC, it is implemented with FFs and other combinational resources. The hardware resource requirement for version 1 register files grows largely with issue-width, therefore, the version 3 is implemented with BRAMs as discussed in Chapter 3. Each BRAM is configured in simple dual port mode with 1W1R port. For multiple ports, the BRAMs are organized into multiple banks and data is duplicated across various BRAMs. Here, the number of banks is equal to the number of write ports, and the number of BRAMs per bank is equal to the number of read ports. The *direction table* is a small register table having the same number of ports as the original register file. For the 2W4R, 4W8R, and 8W16R ports register files, the width of the direction table is 1, 2, and 3 bit(s), respectively, and its depth is the same as that of the GR register file itself. The direction table is implemented with LUTs + FFs. The GR register file version 3 is implemented only for FPGAs.

Table 5.1: Implementation types for GR register files

Version	Implementation detail
1	Straight-forward behavioral implementation. Utilizes LUTs + FFs for FPGA, and FFs for ASIC.
2	Same design as version 3. Utilizes LUTRAMs + LUTs + FFs for FPGA, and FFs for ASIC.
3	Banking and replication with BRAMs. Utilizes BRAMs + LUTs + FFs for FPGA. Not implemented for ASIC.
4	Similar to version 3, but running the internal ports of the BRAMs at twice high the frequency of the external ports. Utilizes BRAMs + LUTs + FFs for FPGA. Not implemented for ASIC.

Figure 3.4 depicts the register file for a 4-issue ρ-VEX processor. Each write port is associated with a bank and all the BRAMs in a bank are simultaneously updated. Each BRAM is organized in a 32-bit wide aspect ratio and parity bits are design-time configurable (1, 2, or 4 for each 32-bit word). The parity bits are generated in the writeback stage and written together with the data. The register data is accessed in the decode stage but the parity check is done in the execute0 stage to avoid the timing overhead. If a parity error is detected on the read data on a register file port, the pipeline is flushed and the error correction procedure is started. We implemented a simple mechanism to correct the corrupted data. For each write port, the written data is already duplicated in multiple BRAMs each associated with a read port (4, 8, and 16 BRAMs for 2-issue, 4-issue, and 8-issue processors, respectively). When a parity error is detected in a data on a read port, the same data is read on another port from a different BRAM in the same bank. The parity for this data is also checked. If the parity is correct, it is assumed that this data is correct. This data is then written to all the BRAMs in the bank where the corrupted data was present in a BRAM. The pipeline is then restarted at the point of the failing instruction and normal execution resumes. Currently, we check only one neighbor BRAM for the correct data instead of all the BRAMs in a bank to simplify the design. If a data word cannot be corrected by the employed technique (e.g., if the same location in all the BRAMs in a bank is corrupted at the same time), an unrecoverable error trap is generated.

The register file version 2 design is similar to version 3. The only difference is that instead of using BRAMs, the required memory blocks are implemented with the distributed memory (LUTRAMs) + LUTs + FFs in FPGA. For ASIC, it is implemented with FFs and other combinational resources.

The design of the GR register file version 4 is similar to that of the version 3. The main difference is that the internal ports of the register file are clocked at twice the frequency of the external ports. This emulates a quad-port BRAM, and hence reduces the required number of BRAMs by one-fourth compared to the version 3 design. The fault detection and recovery techniques are similar to that of the version 3 design. The GR register file version 4 is implemented only for FPGAs.

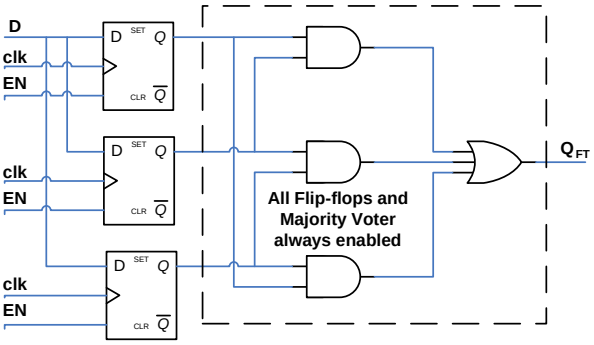
5.4.4 TMR Approach for all Flip-Flops

In the ρ-VEX processor, flip-flops are used for different purposes such as data holding registers, status registers, pipelines latches/registers, state machine registers, etc. The VEX ISA specifies a 1-bit 8-element multiported BR register file for a multi-issue VLIW processor. For 2-issue, 4-issue, and 8-issue ρ-VEX processors, the ISA requires BR register files with 2W2R ports, 4W4R ports, and 8W8R ports, respectively. The BR and link register (LR) files for the processors are implemented with FFs. For both FPGA and ASIC implementations, TMR approach is utilized to protect against the SEU errors in all the FFs used in the processors. Each FF is triplicated and a majority voter is implemented for it, and hence, an SEU error in a single FF can be tolerated. Because the FFs are continuously clocked, any SEU error can be removed within one clock cycle with the output of the voter providing the correct (glitch-free) value.

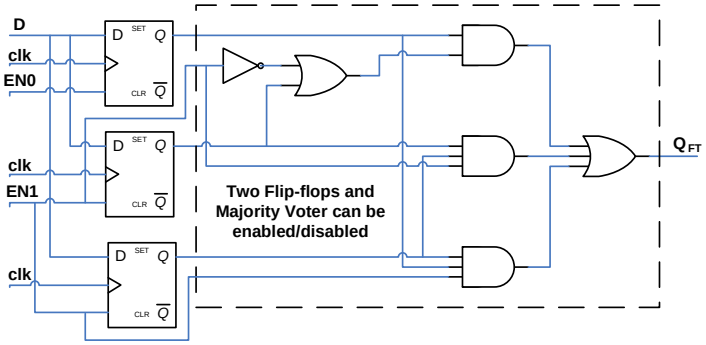
5.4.5 Working of the Configurable Fault-Tolerant System

We implemented fault tolerance techniques that can be customized at design-time and enabled/disabled at run-time. Designers can specify to include or not include the fault tolerance in a processor as well as specify whether the included fault tolerance is permanently enabled or can be enabled/disabled at run-time. Figure 5.1(a) depicts the TMR scheme and the majority voter for the permanently enabled fault-tolerant design. If an application requires that fault tolerance should always be enabled, this design has the advantage of requiring less hardware resources, consuming less dynamic power, and running at higher clock frequency compared to the case where the fault tolerance is run-time configurable (enable/disable).

On the other hand, there could be scenarios in which the application requires fault tolerance only at specific instances of time but not always. For example, certain specific portion of a code needs protection but not all or the device



(a) Permanently enabled TMR approach



(b) Run-time enabled/disabled TMR approach

Figure 5.1: Two approaches used for TMR.

has to be used in an increased radiations environment. In this case, the system should be able to turn off the fault tolerance circuit to avoid consuming the additional dynamic power due to the triplication of FFs. In our run-time reconfigurable design, the fault-tolerant circuit can be enabled and disabled at run-time. The reconfiguration can be controlled by decoding a custom instruction on the processor. This instruction can be placed at different points in the code where the fault-tolerant circuit needs to enabled or disabled. The reconfiguration process can also be initiated from higher level by the user. In this case, the processor can be interrupted and an ISR executed that enables or disables the fault-tolerant circuit. Figure 5.1(b) depicts the TMR scheme and the majority voter for the run-time reconfigurable fault-tolerant design. In this case, the additional two FFs and the majority voter can be enabled/disabled by controlling the *EN1* signal. This design slightly increases the hardware resources and critical path compared to the design in which the fault-tolerant circuit is always enabled. The advantage is that dynamic power consumption

can be reduced at run-time if an application does not require fault tolerance at some point in time. Detailed analysis is presented in Section 5.5.

5.4.6 Fault Coverage and Test Methodology

Fault coverage refers to the percentage of specific type of faults that can be detected with an employed fault tolerance technique. A specific technique may not cover all types of faults, therefore, different techniques are utilized to increase fault coverage of a system. We define a *detectable error* as an error that can be detected by the employed fault detection technique, i.e., the error is within the fault coverage range. A *correctable error* is an error which can be detected and corrected by the employed fault detection and correction techniques. *Non-correctable errors* are those errors which are either not detected or detected but cannot be corrected by the employed technique. Following we discuss the fault coverage of the different techniques that we utilized for error detection and protection.

In the previous sections, we discussed the fault tolerance techniques that are employed to protect different modules of the ρ-VEX processor. We utilized *even parity* scheme to detect errors in the instruction and data memories and the GR register files (only for version 3 and 4). The parity scheme is simple, fast, and requires less hardware for implementation (only XOR gates needed) compared other advanced error detection codes. Although the parity bits per 32-bit word are design-time configurable (1, 2, or 4 bits) in our case, the presented results in Section 5.5 are for 4 bits of parity per 32-bit word. Hence, 1 parity bit is available for every 8-bit of data/instruction. In this case, the error correction technique only needs to correct or recover only 8 bits instead of 32 bits. Whatever the number of data bits that are associated a parity bit, there is a common limitation to parity schemes. A parity bit is only guaranteed to detect an odd number of bit errors. If an even number of bits has errors, the parity bit records the correct number of ones, even though the data is corrupt. A parity bit can only detect all single bit errors and all multiple bit errors where the number of errors is odd. This makes the fault coverage of the parity scheme to be at best around 50%. Techniques which detect/correct multiple bit errors where the number of errors is not odd are out of scope of this thesis.

Apart from the instruction and data memories and the GR register files (only for version 3 and 4), all other modules of the processor are protected against SEU errors utilizing TMR scheme. Each FF is triplicated and a majority voter is implemented for it. In this manner, a bit flip occurring in a single FF of a TMR section can be tolerated as the final result is obtained from the other two

FFs. As all the FFs are continuously clocked, any SEU error can be removed within one clock cycle. Hence, for a single bit error per TMR section, the fault coverage of the TMR technique is 100%. If there are more than 1 errors per TMR section, this technique cannot detect the errors. Both of the permanently enabled and run-time enabled/disabled TMR circuits as presented in Section 5.4.5 have the same fault coverage. These circuits can protect only against SEU errors in a single FF per TMR section. Techniques protecting against SEU errors in multiple FFs per TMR section are out of scope of this thesis.

To test our designs, we utilized the simulation-based fault-injection method [133] which does not require any hardware setup. The method allows fast and easy implementation of the fault injection platform but limits the number of experiments due to its high computational requirements and long simulation time. With the VHDL description of the ρ-VEX processor and utilizing the *ModelSim simulation tool (version 64-bit SE 6.6e)*, we performed realistic fault emulation and detailed system monitoring.

We have written special non-synthesizable routines that generate faults in different regions of the processor at different clock edges, and then record the results. Bit errors are induced in the pipeline registers and other sequential elements of the processors. Injecting errors in FFs does not require stalling a processor and the execution can continue as normal. To inject errors in the GR register file (FPGA implementation) of a processor or the instruction or data memory requires stalling the processor. We injected 3000 1-bit and 2-bit errors in each of the 2-issue, 4-issue, and 8-issue ρ-VEX processors running *matrix multiplication* and *sorting* applications. These errors were injected in two different manners. In the first case, errors were continuously inserted in different modules of a processor after fixed number of clock cycles when an application started execution. In the second case, errors were injected in different modules of a processor randomly distributed over the duration of an application execution.

To test the TMR circuits, errors were injected in FFs. We observed that errors affecting a single FF of a TMR section were automatically corrected. These are called as correctable errors. Errors that effected more than one FFs of a TMR section went undetected, were not corrected, and hence, the results were wrong. These are called as non-correctable errors.

To test the parity scheme, errors were injected in the instruction and data memories and the GR register files (version 3 and version 4 only) of the processor. All1-bit errors in the instruction and data memories were detected and the processor was stopped to correct them, while the 2-bit errors went undetected.

For the GR register file version 3 and version 4 which are implemented using BRAMs and protected with parity bit, the behavior remains the same as that for the instruction and data memories. All1-bit errors were detected and the processor was stopped to correct them (correctable errors), while the 2-bit errors went undetected (non-correctable errors). All of the correctable errors in this case were corrected (see Section 5.4.3). The non-correctable errors which include errors in all of the BRAMs in a bank generated a trap halting the processor execution.

5.5 Implementation Results and Discussion

In this section, we evaluate the implementation results for the different fault-tolerant designs of the ρ-VEX processor. For FPGA implementation, we utilized the *Xilinx ISE (version 13.3)* and the Virtex-6 *XC6VLX240T-1FF1156* FPGA, whereas for ASIC implementation, we utilized the *Synopsis Design Compiler (version G-2012.06-SP2)* and targeted 90 nm technology. The GR and BR register files in all cases are 64×32-bit and 8×1-bit, respectively. The 2-issue, 4-issue, and 8-issue cores have 2, 4, and 8 *ALUs*, and 2, 2, and 4 *MULs*, respectively. Each type of core has a single load/store (*MEM*) unit. The parity bits are design-time configurable, i.e., 1, 2, or 4 bits per 32-bit of word. The results presented in this chapter are for 4 bits of parity per 32-bit word, i.e., 1 bit per byte of data. We represent the base non fault-tolerant by *D1* and the permanently enabled fault-tolerant design by *D2*. *D3* and *D4* (both having same area in terms of hardware resources) represent the processor design in which fault tolerance can be enabled/disabled at run-time. *D3* represents the fault tolerance enabled scenario, while *D4* represents the fault tolerance disabled scenario.

5.5.1 Hardware Resources/Area and Critical Path Delay

Figure 5.2 and Figure 5.3 depict the hardware resources/area and critical path delay results for the base and the fault-tolerant ρ-VEX processors with different types of register files and without instruction and data memories.

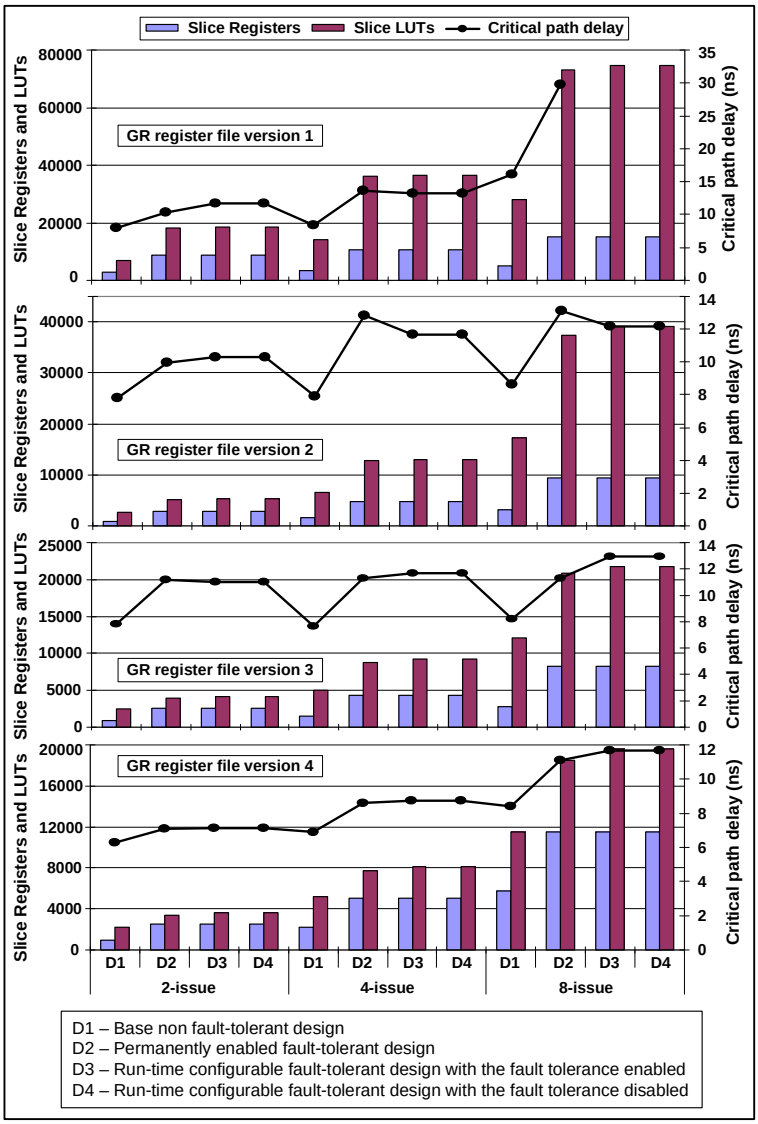


Figure 5.2: Implementation results for the ρ-VEX processors for the Xilinx Virtex-6 FPGA. In addition to the mentioned resources, the 2-issue, 4-issue, and 8-issue cores utilize 4, 4, and 8 DSP48E1s modules, 4, 16, and 64 RAMB36s (GR register file version 3), and 1, 4, and 32 RAMB36s (GR register file version 4), respectively.

As can be observed from Figure 5.2 and Figure 5.3, adding fault tolerance to a processor requires more hardware resources especially the FFs (which are triplicated due to TMR approach) and the additional logic gates for implementing

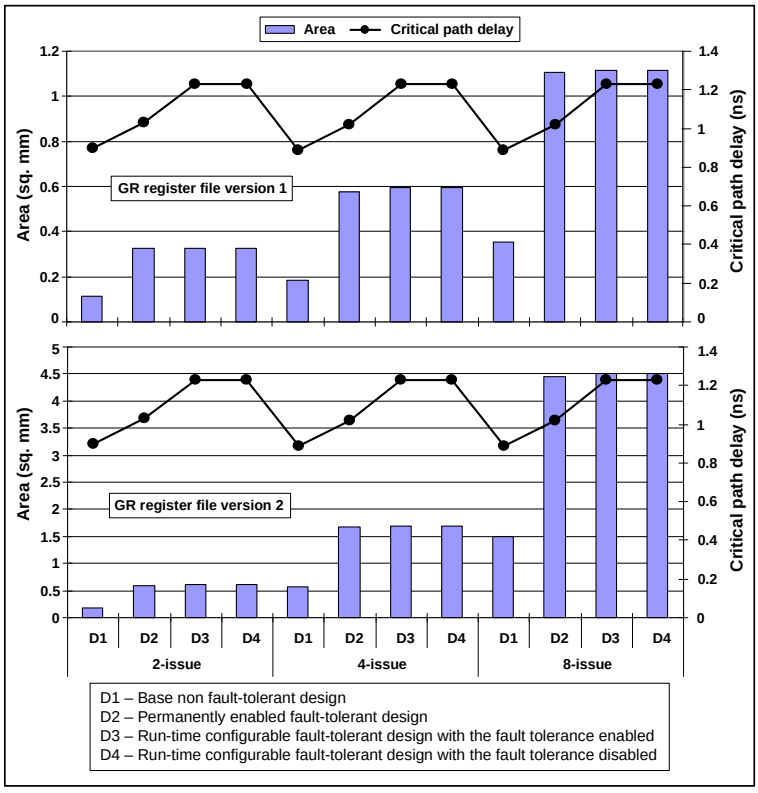


Figure 5.3: Synthesis results for the ρ-VEX processors for 90 nm technology.

majority voters. For the FPGA implementation, the number of BRAMs for the GR register file version 3 and version 4 and instruction and data memories remain the same because we utilize the available extra parity bits in the BRAMs. GR register file version 3 and version 4 are only available for FPGA implementation. For ASIC, the area required for implementing additional parity bits for instruction and data memories increases. In terms of bits increase, it is 1, 2, or 4 bits per 32-bit of word depending upon the desired number of parity bits. Designs *D3/D4* utilize slightly more hardware resources and run at less frequency compared to *D2*. The logic gates utilized for majority voters in *D3/D4* may be accommodated in the already utilized LUTs (FPGA implementation), therefore, the critical path delay remains almost the same as that for *D2*. For ASIC, the increase in critical path delay can be clearly observed when moving from *D1* to *D2* to *D3/D4* due to the additional logic gates in the path (majority voters). In the FPGA implementation, the *D3/D4* designs for 8-issue core with GR register file version 1 become very large and complex. The Xilinx tool

could only map the designs, while the router failed to route them even after running for a long time (more than 5 days). Therefore, the critical path delays could not be calculated for these designs.

5.5.2 Dynamic Power Consumption

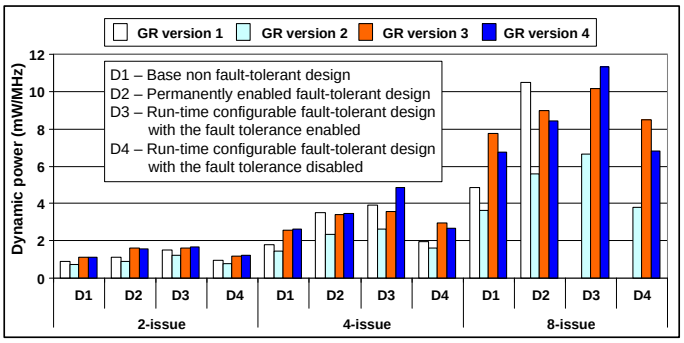
We utilized the Xilinx *XPower Analyzer* tool and the *Synopsis Design Compiler* to measure the dynamic power consumption per MHz for the *XC6VLX240T-1FF1156* FPGA and the 90 nm technology, respectively, as presented in Figure 5.4. Dynamic power is calculated utilizing the equation:

$$Dynamic\ \ Power = ACV^2f \tag{5.1}$$

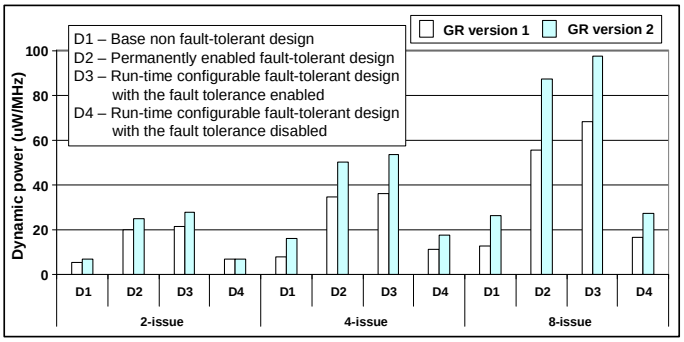
In this equation, A is the switching activity, C is the total capacitive load, V is the voltage, and f is the frequency. Increasing the frequency f increases the dynamic power consumption. The capacitive load C is calculated by the FPGA and the ASIC tools from the implemented designs. The voltage V is constant for the specific FPGA and the specific ASIC gate technology that we are using. The switching activity A can be adjusted by the designer. Because A is a linear term in equation 5.1, therefore, any value of A will result in a linear change in power consumption. Since our purpose is to show the relative power consumption of the different designs of our processor, we can choose any possible value for A. The absolute value of power consumption may change but the trends will remain the same when A is changed. For all our dynamic power estimation in this thesis, we assume A to be 0.1 (i.e., 10% switching activity).

As can be observed from Figure 5.4, implementing fault tolerance in the processors increases the dynamic power consumption due to increased hardware resources. Designs *D2* and *D3* (fault tolerance enabled) consume almost similar dynamic power, while *D4* (fault tolerance disabled) consumes considerably less power compared to *D2* designs. For the FPGA implementation, the power consumption results are not available for the 8-issue *D3*/*D4* designs with GR register file version 1, as the designs could not be routed by the Xilinx tools.

Figure 5.5 depicts the percentage dynamic power reduction for the *D4* designs compared to *D2* designs. In case of the FPGA implementation, the *D4* designs with GR register file version 1 consume 12.73% and 44.32% less dynamic power compared to the 2-issue and 4-issue *D2* designs, respectively. For the GR register file version 2, the 2-issue, 4-issue, and 8-issue *D4* designs consume 13.33%, 30.77%, and 31.54% less dynamic power compared to the



(a) Xilinx Virtex-6 FPGA

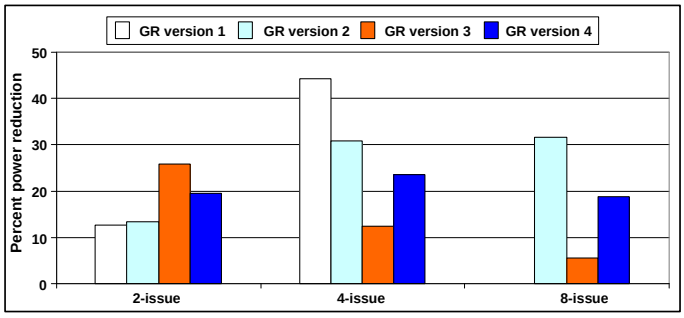


(b) 90 nm ASIC technology

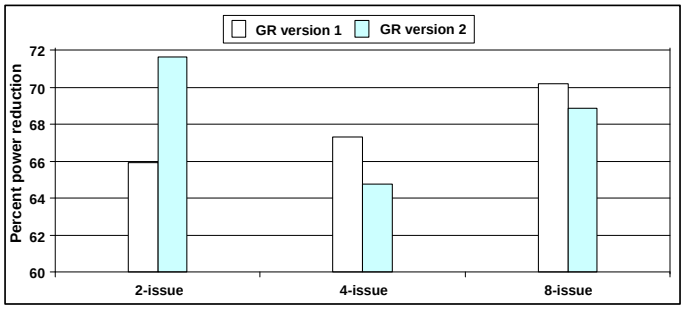
Figure 5.4: Dynamic power consumption per MHz for the ρ-VEX processors.

D2 designs, respectively. Looking at the GR register file version 3, we can observe that the 2-issue, 4-issue, and 8-issue *D4* designs are 25.93%, 12.43%, and 5.56% more power efficient compared to the *D2* designs, respectively. Similarly, for the GR register file version 4, we can observe that the 2-issue, 4-issue, and 8-issue *D4* designs consume 19.48%, 23.56%, and 18.81% less dynamic power compared to the *D2* designs, respectively. For the larger issue-width cores, the GR register file version 3 requires increased number of BRAMs due to the additional number of ports. In FPGAs, BRAMs contribute more to dynamic power compared to FFs, therefore, for the 8-issue processors with GR register file version 3, the dynamic power consumption does not reduce considerably when moving from *D2* to *D4*. This is not visible in the ASIC results, as the GR register files are implemented using FFs, not BRAMs.

For the ASIC implementation, the *D4* designs with GR register file version 1 consume 65.92%, 67.30%, and 70.22% less dynamic power compared to



(a) Xilinx Virtex-6 FPGA



(b) 90 nm ASIC technology

Figure 5.5: Percent dynamic power reduction for the *D4* designs compared to *D2*.

the 2-issue, 4-issue, and 8-issue *D2* designs, respectively. Considering the GR register file version 2, we can observe that the 2-issue, 4-issue, and 8-issue *D4* designs are 71.67%, 64.78%, and 68.87% more power efficient compared to the *D2* designs, respectively. This is considerable power saving, and if fault tolerance is not required at some point in time, it can be turned off to reduce the dynamic power consumption.

5.6 Summary

In this chapter, we presented hardware-based configurable fault-tolerant designs for the ρ-VEX VLIW processor. The designs can detect and correct SEU errors. Parity checking is utilized to detect errors in the instruction and data memories, and the general register files (FPGA implementation). For all other sequential elements, the TMR approach with majority voting is implemented. Different designs for fault tolerance scheme such as permanently enabled at

design-time or with run-time options for enabling and disabling, were presented. These options enable a user to trade-off between hardware resources, performance, power consumption, and reliability.

Note.

The content of this chapter is partially based on the following paper:

F. Anjam and S. Wong. Configurable Fault-Tolerance for a Configurable VLIW Processor. In *International Symposium on Applied Reconfigurable Computing (ARC)*, pp. 167–178, 2013.

6

Results and Analysis

*A **VLIW** processor that can be adapted/reconfigured at design-time as well as at run-time can target applications with diverse requirement of area, performance, and power/energy consumption. In Chapter 3, we presented a design-time configurable VLIW processor that can be adapted in different parameters before it is implemented in hardware. The parameters include the processor's issue-width, the type and number of different FUs and their latencies, type and size of multiported register files, size and width of instruction and data memories, type of interrupt system, and type of default custom operations. Hence, for each particular application, an optimized processor can be generated targeting area, performance, and power consumption characteristics. If the behavior of an application is not known beforehand, the application has different phases with distinct requirements, or a number of different applications need to be executed, a fixed processor may not perform well for all the phases/applications. In Chapter 4, we presented a run-time reconfigurable processor that can change its organization after its realization in hardware. The run-time parameters include the processor's issue-width, the type and number of different FUs, the size of the multiported register file, and size and width of instruction and data memories. The previous chapters provided the implementation (area/hardware) results for the design-time and run-time reconfigurable processors. The current chapter is dedicated to the performance and power/energy consumption analysis of these designs. Following are the contributions of this chapter:*

- *Performance and power consumption results are presented for different issue-width processors.*
- *The effectiveness of run-time task migration among different cores in the 2-4-8-issue processor is evaluated.*

- *The impact of simultaneous reconfiguration of issue-width and instruction cache on performance, energy consumption, and EDP is analyzed.*
- *The effect of increasing the read/write ports (load/store units) on the hardware resources and capacity of data memory/cache is studied.*

The remainder of the chapter is organized as follows. Performance and power consumption results for the 2-4-issue and 2-4-8-issue processors are presented in Section 6.1 and Section 6.2, respectively. Section 6.3 presents the power consumption results for the 2-issue, 4-issue, and 8-issue stand-alone ρ-VEX processors with different types of register files presented in Chapter 3. Section 6.4 discusses the task migration support for the 2-4-8-issue processor and presents the performance and power consumption results. An analysis for the simultaneous reconfiguration of issue-width and instruction cache is presented in Section 6.5. Section 6.6 presents an analysis for the size and required hardware resources for multiport data memory/cache or multiple load/store (LS) units. Finally, Section 6.7 summarizes the chapter.

6.1 2-4-issue Processor

The 2-4-issue processor has two 2-issue cores, which can be utilized independently or combined together to form a 4-issue core. We consider two application scenarios for the processor. The first scenario exploits instruction level parallelism while the second data level parallelism.

Application Scenario 1 This scenario corresponds to applications or kernels with large ILP such as a matrix multiplication program or a discrete Fourier transform (DFT) kernel. Generally, these kernels are part of some larger applications like MPEG video, etc., and these kernels are repeated many times while the application is running. Therefore, running such applications/kernels on a larger issue-width core can provide more performance as compared to a smaller issue-width core. Hence, in our case we can combine the two 2-issue cores to form one 4-issue core and exploit the available ILP. We executed a 100-by-100 matrix multiplication program and a DFT kernel on a single 2-issue core and the combined 4-issue core. Figure 6.1 depicts the speedup for these applications/kernels normalized to the 4-issue core. In this figure, *Issue_2_1* means that the application is running on one of the two 2-issue cores, and *Issue_4* means that the application is running on the combined 4-issue

core. It can be observed from the figure, that running these applications/kernels on a larger issue-width core can improve the performance of these applications/kernels. On the other hand, running these applications on a single 2-issue core can benefit from the lower power consumption as the other 2-issue core can be taken to a lower power mode by gating of its source clock.

Application Scenario 2 In this scenario, the application is such that its data set can be easily divided and run on more than one cores with the data divided among the cores. This scenario corresponds to applications with large data level parallelism such as the advanced encryption standard (AES) encryption/decryption. The AES algorithm takes an input data of 128 bits and a key of 128, 196 or 256 bits and produces an encrypted output data of 128 bits. For decryption the same key is utilized as used in the encryption process. We utilized a 128 bit key version of the AES algorithm. We encrypted and decrypted a text of 1024 bytes. For the single 4-issue core, the C program for the encryption and decryption are compiled and assembled with the input data of 1024 bytes. For the two 2-issue cores, the input data is split into two sets each of 512 bytes. Each core is provided its own data set and the same program for encryption/decryption runs on it. Figure 6.1 depicts the speedup normalized to the combined 4-issue core. In this figure, *Issue_2_2* means the application is running on both of the two 2-issue cores with the data divided among the cores, and *Issue_4* means that the application is running on the combined 4-issue core. It can be observed from the figure that the *Issue_2_2* system completed the execution of the application in almost half time compared to the single *Issue_4* system.

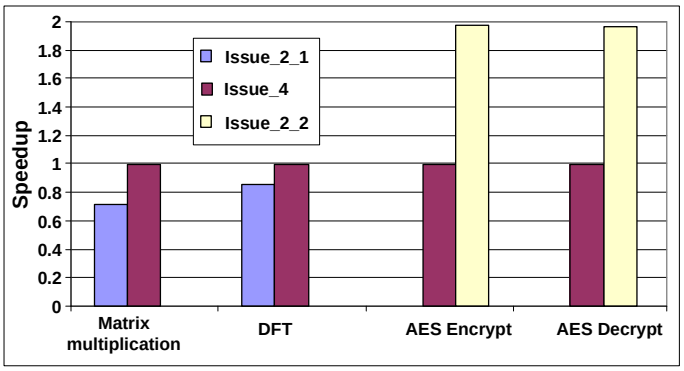
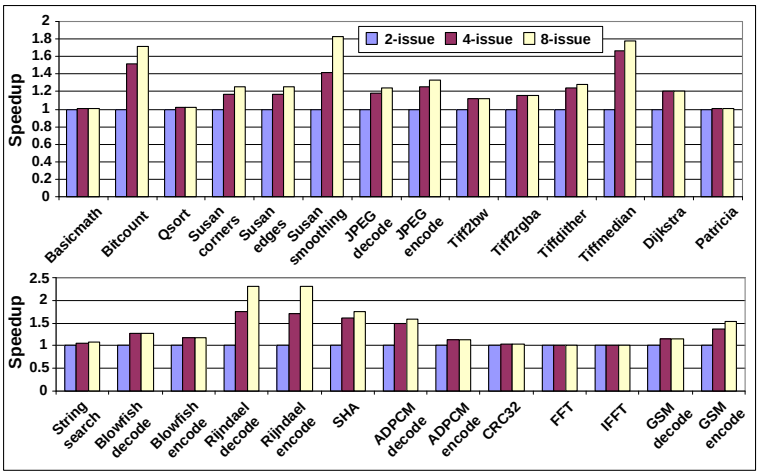


Figure 6.1: Speedup for the 2-4-issue processor normalized to 4-issue core.

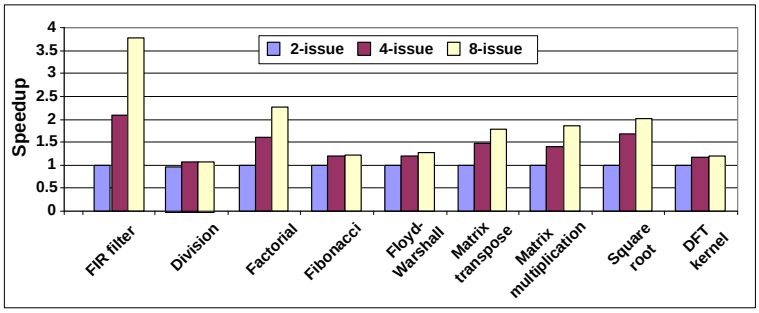
6.2 2-4-8-issue Processor

The 2-4-8-issue processor has four 2-issue cores, which can be utilized independently or combined together to form a variety of configurations. We utilized the *MiBench* benchmark suite [2] and a *custom* benchmark suite. The MiBench is suite of different embedded applications divided into six categories, which includes Automotive and Industrial Control, Consumer Devices, Office Automation, Networking, Security, and Telecommunications. The custom benchmark suite is a collection of different applications/kernels consisting of the following 9 embedded applications: finite impulse response (FIR) filter, integer division, factorial, Fibonacci series, Floyd-Warshall graph, matrix transpose, matrix multiplication, integer square root, and a DFT kernel. We consider two application scenarios for the 2-4-8-issue processor. The first scenario exploits instruction level parallelism while the second data level parallelism. If an application can be split into multiple independent threads that can be run on multiple cores, the performance can be improved. If an application cannot be split into multiple independent threads, it can be run on a larger issue-width core to exploit ILP.

Application Scenario 1 In this case, the available ILP can be exploited by executing the application/kernel as a whole on larger issue-width cores instead of dividing it into multiple threads. Generally, these kernels are part of some larger applications like H.264/MPEG audio/video, etc., and these kernels are executed multiple times while the application is running. Therefore, running such applications/kernels on a larger issue-width core can provide more performance compared to a smaller issue-width core. By combining multiple 2-issue cores to form a larger issue-width core (4-issue or 8-issue), we can exploit the available ILP in a better manner. We executed the MiBench and our custom benchmark suites with three different configurations of the 2-4-8-issue processor, i.e., 2-issue, 4-issue, and 8-issue cores. Figure 6.2 depicts the speedup for the three types of the processor cores normalized to that for a 2-issue core for the two benchmark suites. Here, the 2-issue, 4-issue, and 8-issue cores utilize local data memories with 1, 2, and 4 load/store units, respectively. It can be observed from the figure, that running these applications/kernels on a larger issue-width core can improve the performance of these applications/kernels. On the other hand, running these applications on smaller issue-width cores can benefit from lower power consumption as the other 2-issue cores can be taken to a lower power mode by turning them off.



(a) MiBench benchmark suite



(b) Custom benchmark suite

Figure 6.2: Speedup for the 2-4-8-issue processor normalized to 2-issue core.

Application Scenario 2 n this case, the application is such that its data set can be easily divided and run on multiple independent cores. This scenario corresponds to applications with large data level parallelism such as the Rijndael encryption/decryption algorithm. We utilized a 128 bits key version of this algorithm. We encrypted and decrypted back a text of 2048 bytes using the Rijndael encryption/decryption algorithms. Initially, we run the application as a whole with 2048 bytes on a single 8-issue core. We then run the same application on two 4-issue cores with the data divided among the two cores. Each core encrypts/decrypts its own 1024 bytes of data. In the third experiment, we run the same applications on four 2-issue cores providing 512 bytes of data to each core. The individual encrypted/decrypted data is then combined into a single result. Figure 6.3 depicts the execution cycles for the three types of

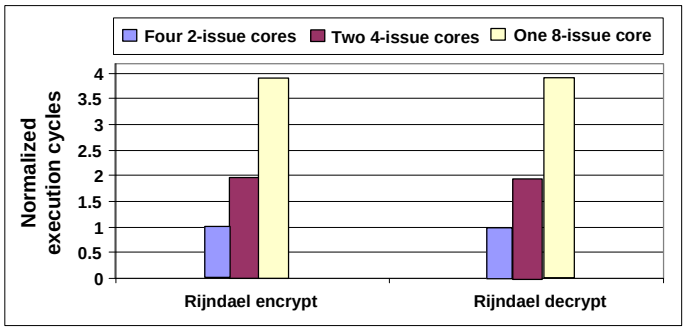


Figure 6.3: Execution cycles normalized to the four 2-issue cores for the Rijndael encryption/decryption algorithms.

the processor system normalized to that for the four 2-issue cores. It can be observed from the figure, that applications with larger data level parallelism execute faster when run on multiple smaller issue-width cores with the input data distributed among them compared to running the application on one larger issue-width core with all the input data. A matrix multiplication program can also be executed over multiple independent cores with the data is distributed over them. For example, one of the matrices is provided to every core, while the rows of the other matrix are distributed over all the cores. Each core performs it own part of the calculation, which is then combined and re-arranged into the final result.

6.2.1 Dynamic Power Consumption

In the 2-4-8-issue processor, the clock input for each 2-issue core is driven by a separate controlled buffer [134]. The clock buffer for a core is controlled by the *run* signal of that core. If the *run* signal for a core is at logic low, the clock to that core is gated off. We used the Xilinx *XPower Analyzer* tool, the ISE release version 13.2, and the Virtex-6 *XC6VLX240T-1FF1156* FPGA for the power consumption analysis. We utilized the typical operating conditions with 10% switching activity. According to equation 5.1, the switching activity A is a linear term in the equation for dynamic power estimation, therefore, any value of A will result in a linear change in power consumption. Since our purpose is to show the relative power consumption of the different versions of our processor, we can choose any possible value for A. The absolute value of power consumption may change but the trends will remain the same when A is changed. For all our dynamic power estimation in this chapter, we assume

A to be 0.1 (i.e., 10% switching activity). The frequency *f* is fixed at 1 MHz. Increasing the frequency increases the dynamic power consumption for any number of the active 2-issue cores. When more cores are turned *on*, the net capacitive load increases, and hence, the dynamic power consumption increases. The capacitive load *C* is calculated by the Xilinx *XPower Analyzer* tool from the placed and routed design of our processor. The voltage *V* is constant for the specific FPGA that we are using. Figure 6.4 depicts the dynamic power consumption per MHz for the 2-4-8-issue processor. It can be observed from the figure that turning off *one*, *two*, or *three* 2-issue cores reduces the dynamic power consumption of the whole system by 34%, 60%, or 82%, respectively. Hence, if any of the 2-issue cores is not active, it can be turned *off* and the system can be taken to a lower power mode.

6.3 Power Consumption for Stand-alone ρ-VEX Processors

In Section 3.2, we presented different types of register file implementations for the ρ-VEX processor. Depending upon the choice and/or the available resources in the FPGA, a ρ-VEX processor can be implemented utilizing any of these register files. When the number of ports on a register file increases, its area and resource requirement increases, and hence, the register file starts increasing the critical path delay of the processor. Different implementations of the register files utilize different types of FPGA resources resulting in different critical path length of the processor. In this section, we present the dynamic power consumption per MHz of the 2-issue, 4-issue, and 8-issue stand-alone ρ-VEX processors with the different types of register files presented in Sec-

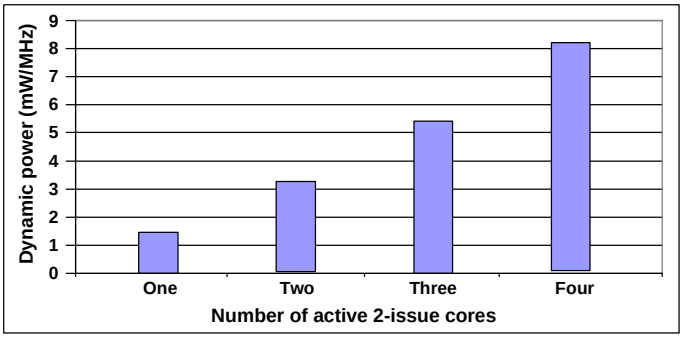


Figure 6.4: Dynamic power consumption for the 2-4-8-issue processor.

tion 3.2. We used the Xilinx *XPower Analyzer* tool, the ISE release version 13.2, and the Virtex-6 *XC6VLX240T-1FF1156* FPGA for the power consumption analysis. We utilized the typical operating conditions with 10% switching activity. Figure 6.5 depicts the dynamic power consumption per MHz for the stand-alone ρ-VEX processor with different issue-widths and different types of register files. As can be observed from the figure, compared to the register file version 1, processors utilizing the register file version 2 consume less dynamic power. The reason is that the register file version 2 is mapped to LUTRAMs resulting in a compact design while version 1 could not be mapped to LUTRAMs, and hence utilizes more LUTs. Processors with register file version 3 utilize more BRAMs (by order of $4\times$) and more signal paths for routing compared to the version 4, therefore they consume more power than the processors with register file version 4. On the other hand, the register file version 4 runs at double frequency compared to the version 3, therefore the power consumption for the processors utilizing register file version 4 also increase.

6.4 Run-time Task Migration Support

As mentioned in Section 4.3, task migration from one core to another requires a total of 155 cycles. Out of these 155 cycles, 76 cycles are required for storing the context of the first core, 1 cycle for accessing the program counter (PC) of the first core, 1 cycle for reconfiguring the issue-width, 76 cycles for restoring the context to the newly configured core, and 1 cycle for loading PC of that core. This means that switching a running application from one type of core

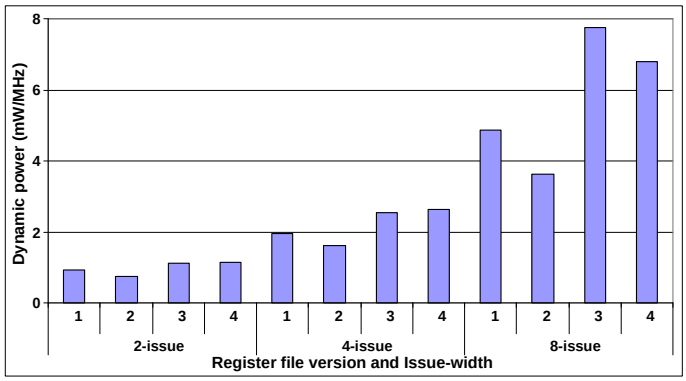


Figure 6.5: Dynamic power consumption for the stand-alone ρ-VEX processor with different issue-widths and different types of register files.

to another core requires 155 extra cycles, but then the execution time or power consumption for the remaining part of the application can be reduced. The issue-width or the organization of the 2-4-8-issue processor can be changed by writing dedicated bits to the configuration register of the processor. The configuration register can be accessed by decoding a custom instruction on the processor. This instruction can be placed at the specific points in the executable code, where an issue-width change is required. In a more global scenario, the configuration register can be implemented in the global space accessible to other dedicated hardware/software controllers. In this case, there is no need for designing the custom operation for the processor. The reconfiguration process can be initiated by some external agents/controllers based on certain run-time metrics such as hardware utilization, power/energy considerations, arrival of other tasks, cache related statistics, etc.

To show the effectiveness of our run-time task migration capable hardware, we utilized the *generic binaries* scheme [80] to generate the binary code for our variable issue-width processor. In the generic binaries scheme, an application is compiled such that the same binary code can be executed correctly on different issue-width VLIW processors with some performance degradation. Here, an application is compiled for an 8-issue core. Then the assembly code is parsed and the operations are re-arranged according to a specific format such that the same instruction can be executed by a 4-issue or a 2-issue core in multiple clock cycles. In this case, there is a performance degradation of 9% to 30% for the 4-issue and 2-issue cores when executing the generic binaries compared to the binaries compiled for specific issue-widths. Utilizing the generic binaries, the processor issue-width can be reconfigured at any point during execution without needing to introduce checkpoints. This avoids the use of complex algorithms and hardware to ensure the application is restarted at the same point in a different version of the code. The advantage is that the same binaries can be utilized when switching the processor issue-width and there is no need for loading/accessing multiple binaries. This reduces the required storage space for instructions and data, their loading time, and the power consumption related to loading a new code. More detail about generic binaries can be found in [80].

We considered the following benchmark applications/kernels: Sobel filter, FIR filter, data encryption standard (DES), secure hash algorithm (SHA), Huffman compression, and Rijndael encode. Generally, these applications/kernels are part of some large applications such as H.264, and are repeated continuously or at least many times. Figure 6.6 depicts the overall execution cycles normalized to a 2-issue core with 1 LS unit for different benchmarks when the

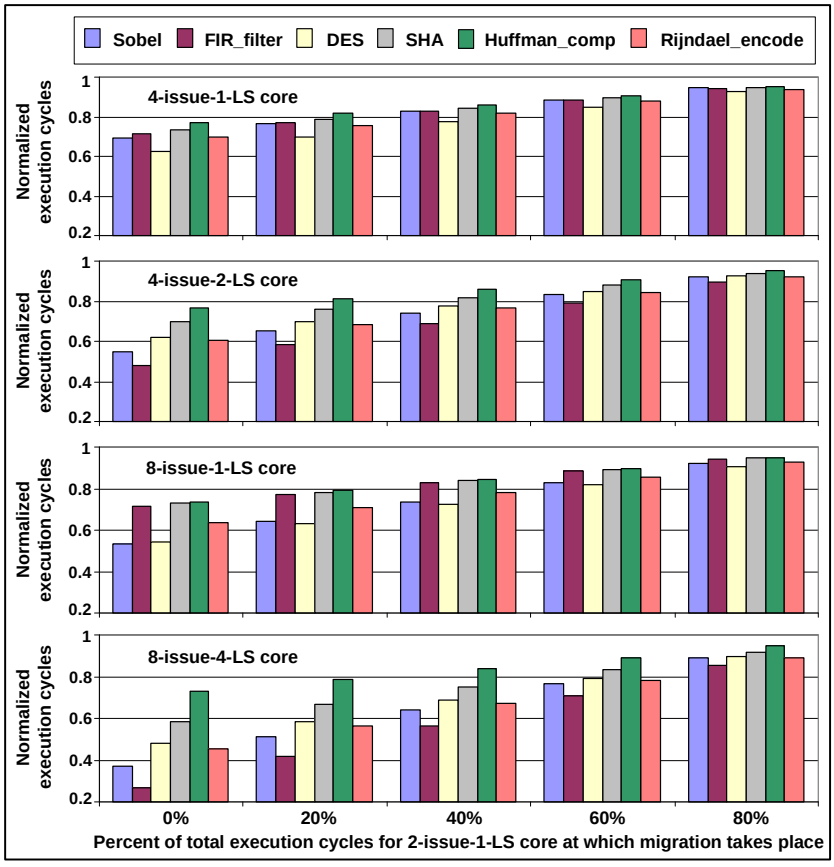


Figure 6.6: Execution cycles normalized to a 2-issue core with 1 load/store (LS) unit.

applications are migrated from this core to a larger issue-width core at different percentage of the total execution cycles for the *2-issue-1-LS* core. The maximum performance is at 0% of execution time, i.e., when the application has just started on the *2-issue-1-LS* core. Hence, reduction in execution cycles is more when the migration is done in the beginning of an application execution. As depicted in Figure 6.6, the compiler is able to extract more ILP for the different applications. In our 2-4-8-issue processor, each of the four 2-issue cores has 1 LS unit. When multiple 2-issue cores are combined, the resulting larger issue-width core can also utilize the additional LS units to increase the data input (provided the data memory has multiple ports) and hence, can further reduce the execution cycles for different applications.

6.4.1 Dynamic Power Consumption

We calculated the dynamic power consumption for the 2-4-8-issue processor with the interrupt system and task migration support. When a code running on a larger issue-width core is shifted to a smaller issue-width core (e.g., from an 8-issue to a 2-issue), the unused 2-issue cores can be clock gated to reduce the dynamic power consumption of the system. We used the Xilinx *XPower Analyzer* tool, ISE release version 13.2, and the Virtex-6 *XC6VLX240T-1FF1156* FPGA for the power consumption analysis. Instead of measuring the dynamic power consumption for a particular application, we utilized 10% switching activity to measure the dynamic power consumption at typical operation conditions. Figure 6.7 depicts the dynamic power consumption per MHz for the 2-4-8-issue processor with task migration support. It can be observed from the figure that turning off *one*, *two*, or *three* 2-issue cores reduces the dynamic power consumption of the whole system by 33%, 59%, or 81%, respectively.

6.5 Simultaneous Reconfiguration of Issue-width and Instruction Cache

As stated earlier, increasing the issue-width of a VLIW processor increases the performance for applications with inherent ILP. Studies have shown that more than half of the chip die is reserved for the on-chip caches and that the energy consumption in cache systems accounts for more than 50% of the total energy consumption. Instruction cache (I-cache) reconfiguration plays an important role in the performance, energy consumption, and/or energy-delay

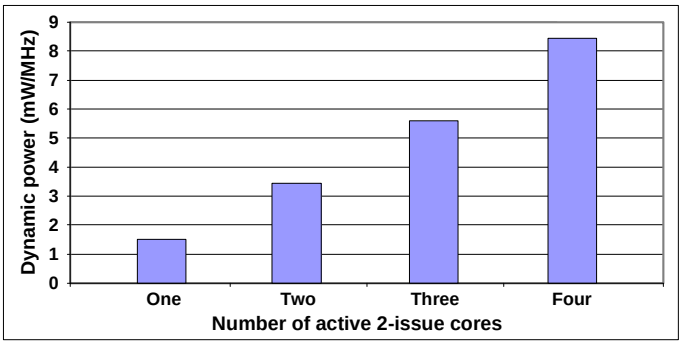


Figure 6.7: Dynamic power consumption for the 2-4-8-issue processor with task migration support.

product (EDP) for different applications. Instead of reconfiguring only the I-cache, reconfiguring both the “issue-width + I-cache" can further improve the performance, energy consumption, and/or the EDP. In this section, we study the effect of I-cache reconfiguration on the performance, dynamic energy consumption, and the EDP for a reconfigurable issue-width VLIW processor. We define EDP as the product of energy consumed and the total execution cycles per application. When issue-width is changed, a different schedule is followed by the compiler and a different request for instructions and data is generated. In this section, we analyze how this request can be better fulfilled by tuning the available I-cache.

6.5.1 Experimental Setup and Benchmark Applications

We utilized the VEX toolchain [1] which includes a parametrized C compiler and a simulator. The compiler reads a machine configuration file and then compiles and schedules the code according to the machine specifications. As mentioned in Section 4.4, there are 30 I-cache configurations (cache associativity: 1/2/4/8 ways, cache size: 4/8/16/32 Kbytes, cache line size: 16/32/64 bytes) and 3 issue-width configurations (2-issue, 4-issue, 8-issue; all with single load/store unit); hence the total search space for each application is 90 “issue-width + I-cache" configurations. The simulator generates a log file containing different information such as total memory accesses, total misses, execution cycles, stall cycles, function profiles etc. For energy calculation of ASIC implementation, we utilized *CACTI 6.5* [104] and *Synopsis Design Compiler* (Synthesis-E-2010.12-SP1) and targeted 90 nm technology. We calculated the energy consumption for each configuration as mentioned in Section 4.4.4. We utilized the *MiBench* [2], *PowerStone* [3], and *custom* benchmark suites for the analysis. The custom benchmark suite includes the following 16 small applications/kernels from different domains: discrete cosine transform (DCT), discrete Fourier transform (DFT), finite impulse response filters (FIR), Floyd-Warshall graph, Hamming distance, Huffman compression and decompression, inverse discrete cosine transform (IDCT), matrix multiply, moving filter, run length encoding (RLE), different sorting applications such as bubblesort, quicksork, radixsort, and shellsort. In Section 6.2, we also utilized applications from the same benchmark suites but excluding the effect of caches. For the study in the current section, we include the I-cache results as well.

6.5.2 Results and Analysis

As stated earlier, when the issue-width is changed, a different schedule is followed by the compiler and a different request for instructions and data is generated. Configuring the I-cache for a fixed issue-width affects the memory accesses and miss/hit rates. The miss/hit rate directly impacts an application's performance and energy consumption as well as EDP. Similarly, configuring the issue-width for a fixed I-cache also impacts memory accesses. As discussed in Section 6.5.1, we consider a large number of applications and I-cache and issue-width configurations. Due to limited space, we cannot discuss all of the results. We present some of the interesting and motivating results showing the importance of reconfiguring both the issue-width and I-cache together.

Figure 6.8 depicts an analysis for three applications; Basicmath, ADPCM decode (D-adpcm), and Rijndael encode (E-rijndael) for the three configurations of our processor issue-width with varying the I-cache configurations. Here, *1W8KB16B* means a cache with 1 way associativity, 8 Kbytes total size, and 16 bytes line size. This is the base cache. We vary the cache in all its three parameters, i.e., doubling the size, the line size, and the associativity. The first graph in Figure 6.8 depicts the execution cycles normalized to "2-issue core + 1W8KB16B I-cache" configuration. Focusing at the Basicmath application, we can observe that there is no effect of changing the issue-width; hence, for all issue-widths, the execution cycles remain the same at different I-cache configurations. We can observe that for any issue-width configuration, varying the I-cache configuration does vary the performance as well as the energy consumption, but this change remains same across all the issue-widths. When either of the execution cycles or energy consumption changes, the EDP changes accordingly. Focusing at the D-adpcm application, we can observe that varying the I-cache configurations has no effect on the performance for different issue-width configurations. The performance increases only with the issue-width reconfiguration. On the other hand, the energy consumption varies with the issue-width and hence the EDP. Considering the Rijndael encode application, we can observe that both the issue-width and the I-cache configurations impact the performance and energy consumption. The "8-issue + 1W8KB32B I-cache" results in the highest performance, the least energy consumption, and hence, the least EDP. This shows that both the issue-width and I-cache reconfiguration together can bring the most optimized result.

In the previous example, we considered a small variation in cache configurations. In the next example, we considered a wider variation in the cache parameters for the three types of the issue-widths. Figure 6.9 depicts the im-

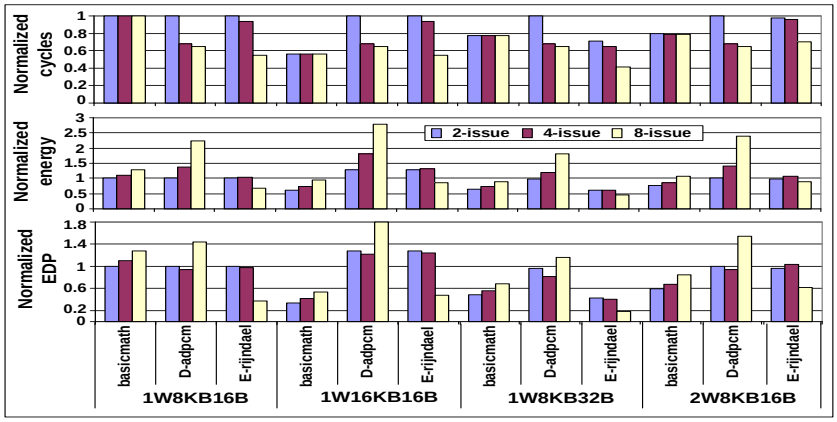


Figure 6.8: Impact of simultaneous reconfiguration of issue-width and I-cache; execution cycles, energy, and EDP normalized to 2-issue and 1W8KB16B I-cache.

pact of I-cache configuration for the different issue-widths for Patricia and Pocsag applications. We consider an I-cache of *1W32KB16B* that is varied in different dimensions. The first, second, and third cache sets are 1W(4-8-16-32)KB16B (varying cache size), 1W32KB(16-32-64)B (varying line size) and (1-2-4)W32KB16B (varying the associativity), respectively. The base caches of the three sets are: 1W4KB16B, 1W32KB16, and 1W32KB16B. The execution cycles, energy, and EDP for each issue-width configuration are normalized to that of the "own issue-width + the base I-cache in each set" configuration.

Considering the Patricia application, when the cache is varied for each type of the issue-width in the first cache set, the execution cycles, energy consumption, and EDP are improved compared to that of the same issue-width with 1W4KB16B I-cache. When the cache is varied in the second and third sets, there is a small variation in the performance, but there is a big variation in energy consumption. It must be noted that the performance does not change with the issue-width; rather it only changes with varying the cache. In case of the Pocsag application, performance and energy consumption only change with varying the I-cache in the first and second cache sets for each issue-width. There is no effect for the third cache set, meaning that the associativity has almost no effect on the execution cycles, energy consumption and EDP for any issue-width for this application. This example shows that both the issue-width and I-cache reconfiguration are important to achieve optimal results in terms of performance or energy consumption.

Different I-cache configurations results in different execution cycles and en-

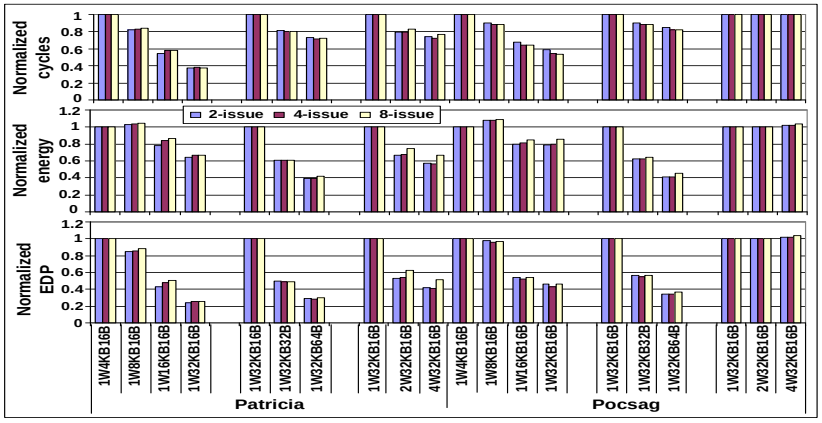
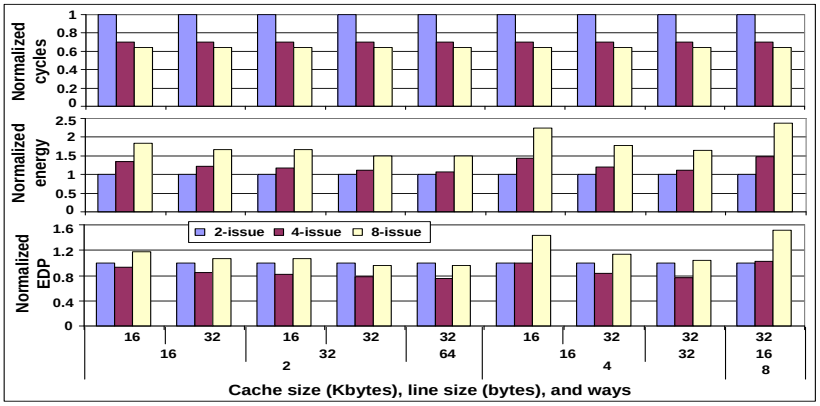


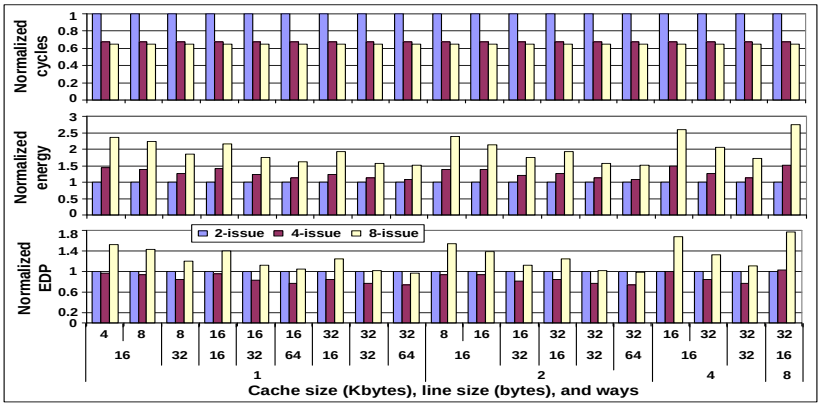
Figure 6.9: Impact of simultaneous reconfiguration of issue-width and I-cache; execution cycles, energy, and EDP for 2-issue, 4-issue, and 8-issue cores with varying I-cache normalized to own issue-width with the base I-cache in each set.

ergy consumption. It is possible that for a particular application, there are different cache configurations resulting in same execution cycles but different energy consumption. In the following, we present few such cases to show the importance of the simultaneous reconfiguration of I-cache and issue-width.

Figure 6.10 depicts different I-cache configurations for the Rijndael encode and ADPCM decode applications, for which the execution cycles remain the same while the energy consumption and EDP vary. The execution cycles, energy consumption and EDP are normalized to that of the2-issue core. Here, the execution cycles decrease with increasing the issue-width, but remains constant for all the considered caches. Considering Figure 6.10(a) for the Rijndael encode application, the 2-issue core consumes less energy at every point compared to the 4-issue and 8-issue cores. As the execution cycles for the 8-issue core are less than that for the 2-issue core, the EDP for the 8-issue is lower than that for the 2-issue core at some points. The 4-issue core behaves somewhere in between the 2-issue and 8-issue cores. For the ADPCM decode application in Figure 6.10(b), the 2-issue core consumes less energy at every point compared to the 4-issue and 8-issue cores. As the execution cycles for the 8-issue core are less than that for the 2-issue core, the EDP for the 8-issue core is equal to or less than that for the 2-issue core at some points. The 4-issue core consumes more energy and requires much less execution cycles compared to the 2-issue core, therefore its EDP is lower than that for the 2-issue core. Similarly, the 4-issue core consumes less energy compared to the 8-issue core, while there is a small difference in the execution cycles, therefore, its EDP is



(a) Rijndael encode

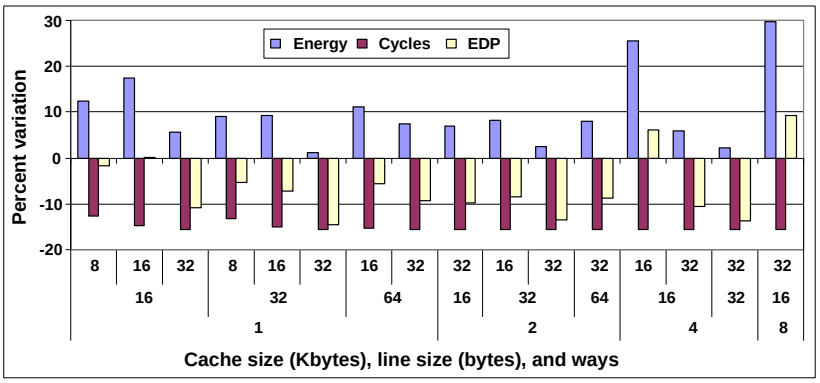


(b) ADPCM decode

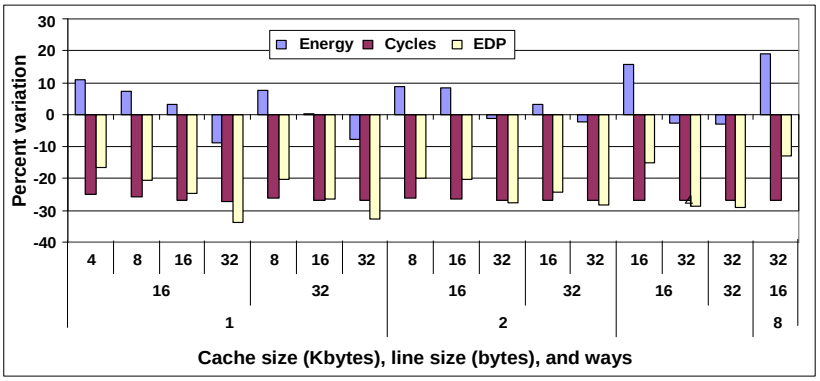
Figure 6.10: I-cache configurations for which execution cycles remain the same but energy consumption and EDP vary.

also lower than the 8-issue core. Hence, if low power is required, the 2-issue core can be selected, and if high performance is required, the 8-issue core can be selected. For lower EDP, the 4-issue core can be selected.

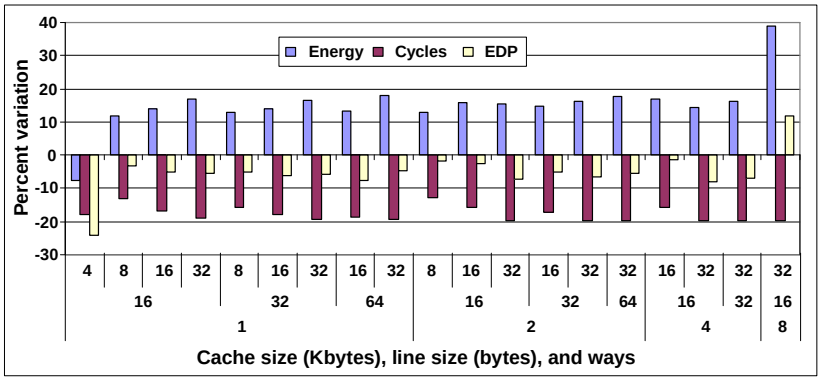
We can also optimize the configuration process by considering the percentage variations in energy consumption, execution cycles, and EDP for an application when the issue-width is varied from 2-issue to 4-issue and 8-issue with different I-caches. By plotting these configurations, we can easily spot the optimal points. Figure 6.11 depicts the percentage variations in energy consumption, execution cycles, and EDP for the Dijkstra, Tiffmedian, and GSM encode applications when the issue-width is changed from 2-issue to 4-issue



(a) Dijkstra



(b) Tiffmedian



(c) GSM encode

Figure 6.11: Percentage variation in energy, execution cycles, and EDP for 4-issue core compared to 2-issue core with different I-caches.

with different I-caches. Considering Figure 6.11(a) for the Dijkstra application, when the issue-width is increased from 2-issue to 4-issue, there is a 15% reduction in execution cycles for almost each cache configuration. The energy consumption varies from 1% to 30% and the EDP from -15% to 9%. For the Tiffmedian application in Figure 6.11(b), when the issue-width is increased from 2-issue to 4-issue, there is a 27% reduction in execution cycles for almost each cache configuration. The energy consumption varies from -9% to 19% and the EDP from -34% to -13%. Similarly, for the GSM encode application in Figure 6.11(c), when the issue-width is increased from 2-issue to 4-issue, there is a 20% reduction in execution cycles for almost each cache configuration. The energy consumption varies from -8% to 39% and the EDP from -24% to 12%.

Figure 6.12 depicts the percentage variations in energy consumption, execution cycles, and EDP for the Rijndael encode application when the issue-width is changed from 2-issue to 4-issue and 8-issue with different I-caches. Here, the continuous lines are drawn only for clarity purpose; otherwise, the values are only at discrete points. When the issue-width is changed from 2-issue to 4-issue, the execution cycles vary from -30% to -2%, the energy consumption changes from 2% to 34%, and the EDP from -26% to 7%. When the issue-width is increased from 2-issue to 8-issue, the execution cycles vary from -45% to 0.4%, the energy consumption changes from -33% to 144%, and the EDP from -63% to 145%.

All of the previous mentioned example show that the simultaneous reconfig-

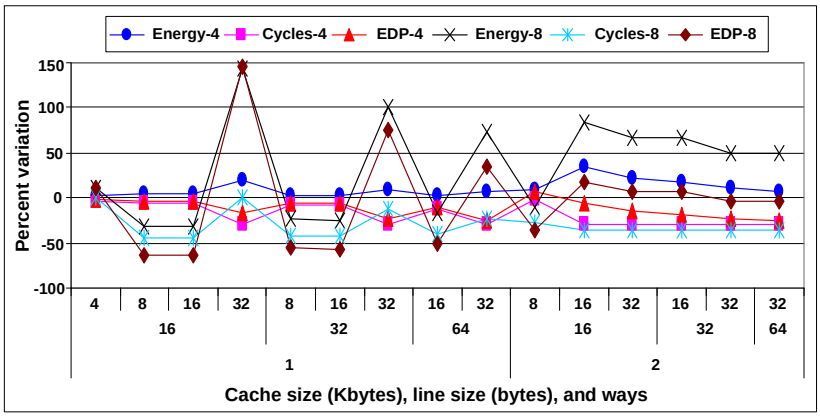
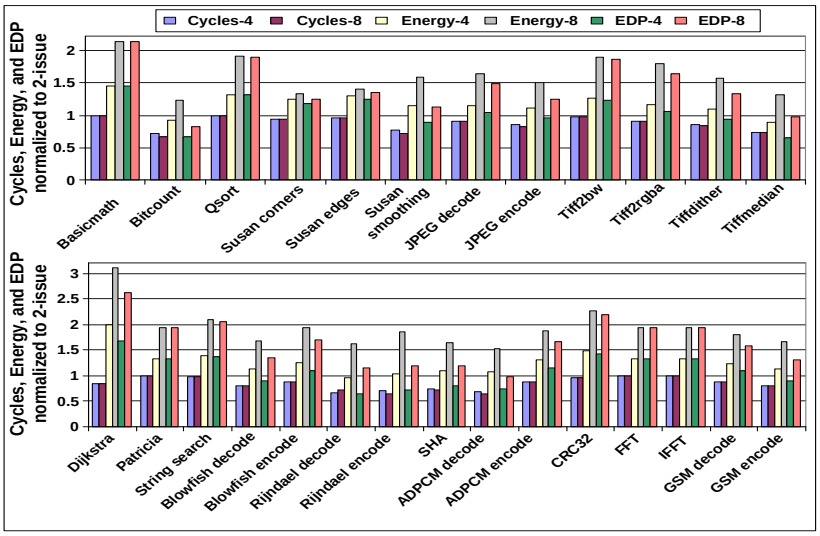


Figure 6.12: Percentage variation in energy, cycles, and EDP for 4-issue and 8-issue cores compared to 2-issue core with different I-caches for the Rijndael encode.

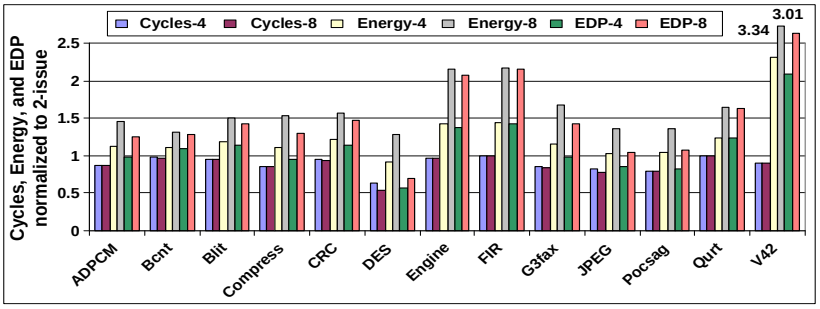
uration of issue-width and I-cache increases the search space for finding the optimal energy consumption, execution cycles, and EDP compared to reconfiguring either the issue-width or the I-cache alone. If energy is the main concern, the issue-width + I-cache resulting in lower energy consumption can be selected. If performance is the prime concern, the issue-width + I-cache with the lower execution cycles can be selected.

Because of the limited space, we cannot show similar results for all of the considered benchmark applications. In the following, we show and discuss results for the complete set of benchmark applications by considering their "*best I-caches*". The best I-cache with a particular issue-width for an application could be the one resulting in the highest performance, the least energy consumption or the least EDP. For the following discussion, we consider the best I-cache for each issue-width and each application as the one which results in minimum energy with reasonable performance (not less than 20% of the maximum performance). Our assumption is based on the fact that mostly the main purpose of cache reconfiguration is considered as the energy reduction.

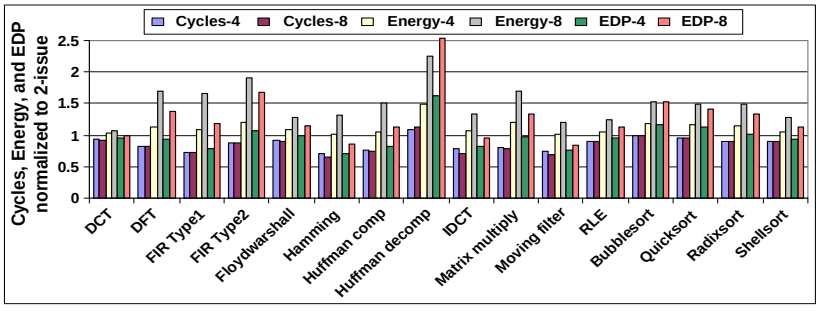
Instead of comparing to a fixed "issue-width + I-cache", we compare the results for the 2-issue, 4-issue, and 8-issue cores with their best I-caches (as explained in the previous paragraph) for all of the benchmark applications. In this manner, we can optimize the performance, dynamic energy consumption, and the EDP for each application. Figure 6.13 depicts the execution cycles, dynamic energy consumption, and the EDP for the 4-issue and 8-issue cores with their best I-caches normalized to that of the 2-issue core with its best I-cache for the MiBench, PowerStone, and the custom benchmark suites. In general, we observed that switching from 8-issue core to 4-issue or 2-issue core reduces energy consumption. The main reason is that the 8-issue core reads a longer instruction (256 bits) per access from the cache while the 4-issue and 2-issue cores read shorter instructions (128 bits and 64 bits, respectively) per cache access. Additionally, the 8-issue core utilizes more functional units compared to 4-issue and 2-issue cores. On the other side, switching from 2-issue core to 4-issue or 8-issue core increases the performance as more operations can be executed in parallel. In the following, we briefly discuss these results (performance, energy consumption, and EDP) for the different issue-widths. The purpose of this discussion is to evaluate effectiveness of the simultaneous reconfiguration of the I-cache and the issue-width for the different benchmark applications. The best I-cache for a particular issue-width and a particular application can only be selected when the cache hardware is able to reconfigure as is the issue-width.



(a) MiBench benchmarks



(b) PowerStone benchmarks



(c) Custom benchmarks

Figure 6.13: Execution cycles, energy consumption, and EDP for the 4-issue and 8-issue cores normalized to 2-issue core (all with their best I-caches).

From Figure 6.13, we can observe that there are some applications/kernels such as Bitcount, Tiffmedian, ADPCM decode, DES, DCT, Hamming distance, IDCT, Moving filter, where the EDP for the 8-issue core with its best I-cache is less than or equal to that of the 2-issue core with its best I-cache. For these applications switching from 2-issue to 8-issue core increases the performance more than the energy consumption and hence reduces the EDP. The largest reduction in the EDP is for the DES application, which is about 30%. There are some applications such as Susan smoothing, Rijndael decode, Rijndael encode, SHA, JPEG, Pocsag, FIR Type1, Floyd-Warshall, Huffman compression, RLE, shellsort, where, for a small increase in EDP, one can get more performance when the issue-width is changed from 2-issue to 8-issue.

Similarly, considering the 4-issue and 2-issue cores with their best I-caches, we can observe that there are many applications, where the EDP for the 4-issue core is less than or equal to that of the 2-issue core. These applications are; 11 in MiBench: Bitcount, Susan smoothing, JPEG encode, Tiffdither, Tiffmedian, Blowfish decode, Rijndael encode and decode, SHA, ADPCM decode, and GSM encode, 6 in PowerStone: ADPCM, Compress, DES, G3fax, JPEG, and Pocsag, and 12 in custom benchmark suite: DCT, DFT, FIR type1, Floyd-Warshall, Hamming, Huffman compression, IDCT, matrix multiply, moving filter, RLE, radixsort, and shellsort. This means that for these applications, switching from 2-issue to 4-issue core increases the performance more than increasing the energy consumption and hence, reduces the EDP. Compared to “2-issue + the best I-cache", “4-issue + best I-cache" reduces the EDP for Tiffmedian, Rijndael decode, and DES by about 30%, 36%, and 41%, respectively. Additionally, there are some applications, where, with a small increase in EDP, one can get more performance when the issue-width is changed from the 2-issue to the 4-issue.

Considering the energy consumption with the best I-caches for every issue-width, there is no case in the considered benchmarks, where the 8-issue core consumes less energy than the 2-issue or 4-issue core. The main reason is that the 8-issue core consumes more power compared to the smaller issue-widths. There are some applications such as Bitcount, Tiffmedian, Rijndael decode, DES, Hamming distance, where the 4-issue core consumes less energy than that of the 2-issue core, both with their best/optimal I-caches. Tiffmedian and DES consume 11% and 6% less energy, respectively, on a 4-issue core compared to a 2-issue core both with their best I-caches. There are many applications such as Susan smoothing, JPEG encode, Tiffdither, Blowfish decode, Rijndael decode, SHA, ADPCM decode, GSM encode, Compress, G3fax, JPEG, Pocsag, DCT, DFT, FIR Type1, Floyd-Warshall, Huffman compression, IDCT,

matrix multiply, moving filter, RLE, shellsort, where, by switching from a 2-issue to a 4-issue core (both with their best I-caches) results in a large performance gain with a small energy increase.

Considering the execution cycles with the best I-caches for every issue-width, all the considered benchmark applications perform better with the 8-issue and 4-issue cores compared to the 2-issue core. Switching from 2-issue to 8-issue core (both with their best caches) reduces the execution cycles for Hamming distance, ADPCM decode, and DES by about 36%, 40%, and 46%, respectively. The largest reduction in execution cycles when switching from a 2-issue to 4-issue core (both with their best caches) is for the Rijndael decode application which is about 37%.

6.6 Multiport Data Memory/Cache Analysis

Multiple load/store (LS) units can increase the performance for some data intensive applications. Multiple LS units mean multiple read/write ports on the data memory/cache. In this section, we evaluate the cost of multiple LS units or multiple read/write ports on the data memory/cache. In FPGAs, large amount embedded synchronous memory is available in the form of BRAMs. The data memory (local or cache) is implemented with BRAMs. In the Xilinx Virtex-6 *XC6VLX240T* FPGA, there are 832 BRAMs. Each BRAM provides 18 Kbits or 2 Kbytes of data storage. Each BRAM provides one read and one write (1R1W) port.

6.6.1 Local Data Memory

Consider we need *M* Kbytes of data memory to be implemented using BRAMs. When the data memory is local, *N = ceil(M/2)* BRAMs can store and provide the data to a single LS unit. When the LS units increase, multiple of *N* BRAMs are needed in order to provide the same *M* Kbytes of memory. BRAMs are arranged in different banks where each bank is associated with a write port. Multiple BRAMs are utilized inside a bank where each BRAM is associated with a read port. Figure 6.14 depicts a 2R2W ports data memory configuration implemented using BRAMs. The *Direction Table* is a multiport memory implemented with the FPGA's configurable LUTs/slices. The depth of the *Direction Table* is the same as that of the data memory and its width depends on the number of write ports (log_2 of the number of write ports). The hardware utilization and the latency of the *Direction Table* and the asso-

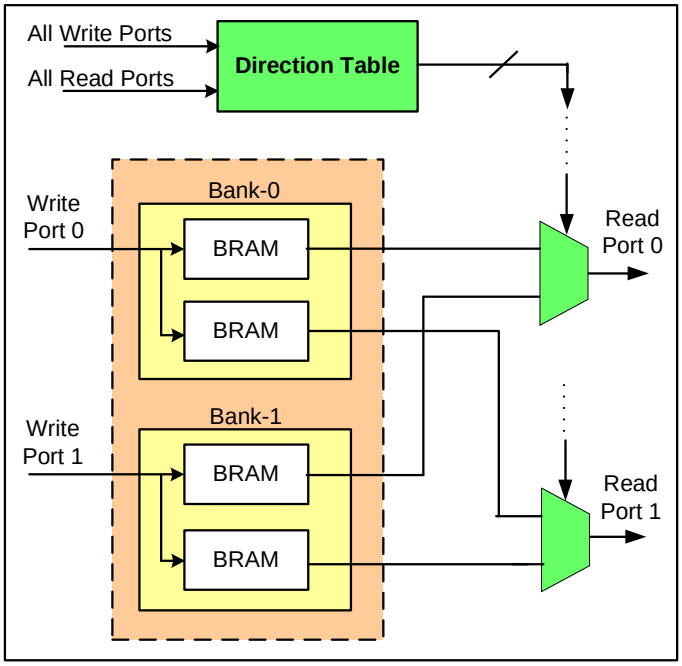


Figure 6.14: 2R2W ports data memory configuration implemented with BRAMs.

ciated multiplexers grow with the size of the data memory. Table 6.1 presents the number of BRAMs required to provide *M* Kbytes of data memory with multiple read/write ports or LS units. As can be observed from the table, the resource requirement or area increases exponentially with increasing the LS units. Compared to a single LS unit, keeping the area/resources same for the data memory, providing two, three, and four read/write ports reduces the data storage of the memory by 1/4, 1/9, and 1/16, respectively.

Table 6.1: Number of BRAMs required for M Kbytes of data memory.

Load/store units	Read/Write ports	Total BRAMs
1	1R1W	N = ceil(M/2)
2	2R2W	2×2×N
3	3R3W	3×3×N
4	4R4W	4×4×N

6.6.2 Data Cache

In FPGAs, cache memory is also implemented using BRAMs. Providing multiple LS units or multiple read/write ports complicates the cache controller design as well as increases the cache memory area. We performed an analysis for the area requirement (total number of BRAMs) of the data cache with multiple read/write ports. We did not include the hardware resources required to implement the cache controllers. We only present results regarding the cache memory. The data cache memory has mainly two components: *data store* and *tag store*. Both are implemented with BRAMs. The size of the data store is determined by the cache size, and the size of the tag store is determined by the line size as well as the cache size. The cache associativity also affects the size of the tag store. Figure 6.15 depicts the total number of BRAMs required to implement a 1-way data cache memory (data store + tag store) with multiple read/write ports and varying cache parameters. The cache size varies from `4`, `8`, `16`, to `32` Kbytes, and line size varies from `16`, `32`, to `64` bytes. It can be observed from the figure that keeping the cache parameters the same, the number of BRAMs increases exponentially with increasing the number of read/write ports. This consideration is important when designing a VLIW processor with multiple LS units. Although, increasing the LS units may improve the performance of some applications, but the designer has to keep in mind the related hardware cost. For example, the designer has to consider whether he/she needs more memory size (in Kbytes) with less number of read/write ports or less memory size with more read/write ports given the same number of BRAMs.

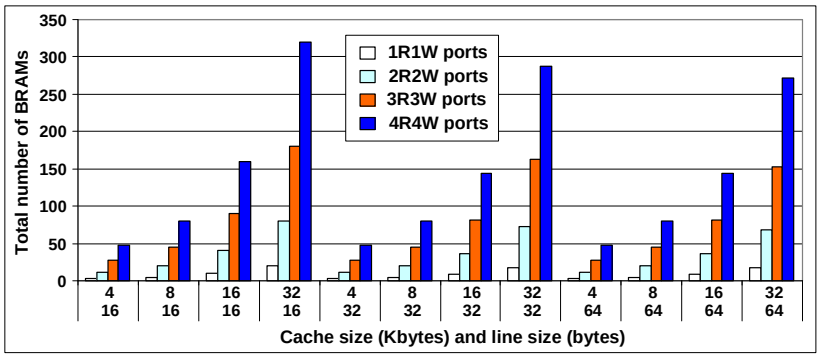


Figure 6.15: Number of BRAMs (Xilinx RAMB18s) required to implement 1-way data cache memory (data store + tag store) with multiple read/write ports.

6.7 Summary

In this chapter, we evaluated our reconfigurable processor designs presented in the previous chapters in terms of performance and power/energy consumption. The 2-4-issue and 2-4-8-issue adaptable processors are utilized in different configurations (2-issue, 4-issue, and 8-issue). For performance analysis, we utilized different application benchmark suites (MiBench, PowerStone, and a custom-made benchmark suite). For power consumption analysis, we utilized the Xilinx XPower Analyzer tool for FPGAs. We discussed the effectiveness of the run-time task migration among different cores for the 2-4-8-issue processor. With the task migration scheme, performance can be improved or power consumption can be reduced at run-time. Additionally, we presented an analysis (performance, dynamic energy consumption, and EDP) on the simultaneous reconfiguration of the issue-width (2, 4, and 8) and instruction cache (associativity, cache size, and line size) for the 2-4-8-issue processor. Finally, we analyzed the effect of increasing the number of read/write ports (LS units) on the data memory/cache in terms of total storage capacity and hardware area.

Note.

The content of this chapter is partially based on the following papers:

F. Anjam, M. Nadeem, and S. Wong. A VLIW Softcore Processor with Dynamically Adjustable Issue-slots. In *International Conference on Field Programmable Technology (FPT)*, pp. 393–398, 2010.

F. Anjam, M. Nadeem, and S. Wong. Targeting Code Diversity with Run-time Adjustable Issue-slots in a Chip Multiprocessor. In *Design, Automation and Test in Europe Conference (DATE)*, pp. 1358–1363, 2011.

F. Anjam, Q. Kong, R.A.E. Seedorf, and S. Wong. On the Implementation of Traps for a Softcore VLIW Processor. In *HiPEAC Workshop on Reconfigurable Computing (WRC)*, 2012.

F. Anjam, Q. Kong, R.A.E. Seedorf, and S. Wong. A Run-time Task Migration Scheme for an Adjustable Issue-slots Multi-core Processor. In *International Symposium on Applied Reconfigurable Computing (ARC)*, pp. 102–113, 2012.

F. Anjam, L. Carro, S. Wong, G.L. Nazar, and M.B. Rutzig. Simultaneous Reconfiguration of Issue-width and Instruction Cache for a VLIW Processor. In *International Conference on Embedded Computer Systems: Architecture Modeling and Simulation (SAMOS)*, pp. 183–192, 2012.

7 Conclusions

We have argued in this dissertation that the combination of programmability with reconfigurability by implementing a reconfigurable programmable VLIW processor in an FPGA will bring several advantages such as improved performance, reduced power/energy consumption, design flexibility, and rapid application development. Because FPGA development requires the knowledge of HDLs, to this end, we presented an open-source customizable design of a VLIW processor. A complete development toolchain including a parametrized compiler and a simulator is publicly available. Applications can be developed in a high-level language, such as C, while at the same time, the processor organization can be adapted to the specific requirements of different applications both at design-time as well as at run-time. In this dissertation, we presented different optimization techniques for the proposed VLIW processor and evaluated its effectiveness.

In this final chapter, we summarize the main conclusions and present the major contributions of the thesis, and list some possible future research directions. The remainder of the chapter is organized as follows. Section 7.1 summarizes the main conclusions of this dissertation. Section 7.2 lists the major contributions described in this dissertation. Finally, Section 7.3 highlights several possible future research directions.

7.1 Summary

In Chapter 1, we have highlighted the importance of programmability and reconfigurability. Programmability refers to reordering the existing instructions to perform different tasks. The instruction set is fixed and different programs make use of the instructions to execute different tasks on the processor. Programmability means how flexible a processing element is to adapt to a new

application. Generally, programmable processors cannot change their organizations after fabrication, and have lower performance and higher power consumption compared to a dedicated ASIC. On the other hand, reconfigurability refers to the ability to change the functionality of instructions themselves, i.e., the instruction set can be redefined. A (re)configurable processor can adapt its instruction set as well as its hardware organization. For example, the issue-width can be changed when required by an application for improved performance or reduced power consumption.

In order to take advantage of both, we have proposed to combine programmability with reconfigurability by implementing a programmable VLIW processor in a reconfigurable hardware such as FPGA. A VLIW processor has simple hardware design, consumes low power, and can provide high performance. Different parameters of the processor such as issue-width, the number and type of execution units, register file size, degree of pipelining, cache parameters, fault tolerance, peripherals implementation, etc., can be made configurable and selectable at design-time. Hence, an optimized processor in terms of performance, area, power/energy consumption, and reliability can be quickly implemented for each application. Additionally, the processor is made run-time reconfigurable, where, after the implementation in hardware, certain parameters of the processor can be adapted in order to target performance vs. power consumption trade-offs.

After discussing the advantages and disadvantages of VLIW and superscalar processors in Chapter 1, we have argued in Chapter 2 to focus on a VLIW processor rather than a superscalar processor. Both processors utilize multiple parallel execution units to exploit ILP. For a VLIW processor, a compiler extracts the ILP, where as for a superscalar processor, a run-time hardware determines the number of operations to be issued in parallel. This makes the design of a VLIW processor simpler and more power efficient compared to a superscalar processor at the expense of a complex compiler. We presented different motivational scenarios why we chose to start with the available ISA and toolchain. We discussed the VEX ISA based on which we have developed our adaptable VLIW processor. The VEX toolchain is used for architectural exploration and code generation. In the end, we surveyed the state-of-the-art in configurable softcore processors.

In Chapter 3, we presented an open-source design-time customizable softcore VLIW processor called ρ-VEX. We presented a methodology to implement and utilize the processor. Applications written in C language can be profiled and simulated with the VEX toolchain to determine the suitable parameters

for the processor. The parameters include the processor's issue-width, the type and number of different execution units and their latencies, the type and size of register files and the number of read/write ports, size of instruction and data memories, type of interrupt and exception systems, selection of default custom operations, datapath sharing, etc. These parameters are placed in two configuration files which are input to our synthesizable VHDL code during the processor implementation. Hence, without knowing the HDLs, the designer can generate a desired/optimized ρ-VEX processor. The same parameters are provided to the C compiler to generate the VEX assembly code for the application. This code is passed through a custom assembler to generate binaries for the application. Instruction and data memories can be initialized from the binaries. Using this methodology, trade-off between performance, hardware resource utilization, and power consumption can be made for different applications, and hence, optimized implementations can be generated. The following has been achieved in relation to the open questions posed in Section 1.3:

- By implementing a VLIW processor in an FPGA, we have combined programmability with reconfigurability. To this end, we have proposed a softcore VLIW processor that can be customized in different parameters before implemented in hardware. Applications can be profiled to determine the suitable processor organization for it, which can then be implemented in hardware. Hence, the processor can be tuned to match the particular requirements of each application.

- The synthesizable VHDL design for the VLIW processor has been made parametrized, and hence, optimized solutions can be generated without using any C-to-VHDL tools. Applications can be developed in C, while taking advantages of the reconfigurability provided by an FPGA.

- An optimized instruction encoding scheme has been proposed to increase the available opcode space. A methodology to extend the instruction set of the processor has been presented. Different sub-word custom operations have been implemented that could be added to the processor at design-time.

In Chapter 4, we extended the design-time configurable processor presented in Chapter 3 to make it run-time reconfigurable. The processors have multiple (two for the 2-4-issue processor and four for the 2-4-8-issue processor) 2-issue cores, each of which can run independently. If not in use, each core can be taken to a lower power mode by gating off its source clock. Multiple 2-issue cores can be combined at run-time to form larger issue-width VLIW cores and

a variety of other multi-core configurations. The run-time reconfigurable parameters include the issue-width, the number and type of execution units, and the register file size. The processors can target a variety of applications having instruction, data, and task level parallelism. Based on the interrupt system presented in Section 3.3, we developed a run-time task migration scheme for the 2-4-8-issue processor. With this scheme, cores can be utilized more efficiently. A task running on a core can be migrated to a larger or a smaller issue-width core for performance improvement or power reduction, respectively. Additionally, we discussed the simultaneous reconfiguration of issue-width and instruction cache for the 2-4-8-issue processor to target performance, dynamic energy consumption, and EDP. The following has been achieved in relation to the open questions posed in Section 1.3:

- In order to target performance vs. power consumption trade-offs at run-time, dynamically reconfigurable multi-core processors comprising of multiple 2-issue ρ-VEX cores have been proposed and implemented. The smaller cores could be utilized independently to exploit DLP/TLP or could be combined at run-time for making larger issue-width cores to exploit ILP. The cores could only be combined or split when they are idle, i.e., not executing any application.
- A mechanism for run-time task migration among different cores of a multi-core processor has been proposed to improve the performance or reduce the power consumption of the processor at run-time. With the task migration scheme, cores could be combined or split even when they are not idle, and hence, could be utilized more efficiently.
- A system with reconfigurable issue-width and instruction cache has been proposed in order to analyze the effect of simultaneous reconfiguration of issue-width and instruction cache on the performance, dynamic energy consumption, and EDP for different applications.

When the datapath of a processor gets larger and complex, the probability of errors (such as radiation-induced soft errors) also increases. Therefore, it is becomes necessary to employ fault-tolerant techniques in order to guarantee high reliability and dependability of the safety-critical systems. Run-time detection plays an important role in dependable systems, where it is needed that the computed data is either correct or an error signal is generated whenever there is a possible error. In Chapter 5, we presented hardware-based configurable fault tolerance mechanisms for our configurable processors. Separate techniques are employed to protect different modules of the processor from single event

upset errors. Parity checking is utilized to detect errors in the instruction and data memories and the general register file, while triple modular redundancy approach is employed for all the synchronous flip-flops. At design-time, a user can specify to include or exclude the fault tolerance in the processor designs. Additionally, the user can choose to implement a design in which fault tolerance is always enabled or run-time reconfigurable. In the later case, the fault tolerance can be enabled and disabled at run-time to optimize power consumption whenever fault tolerance is not needed. These options enable a user to trade-off between hardware resources, performance, power consumption, and reliability. The following has been achieved in relation to the open questions posed in Section 1.3:

- Hardware-based configurable fault tolerance techniques have been proposed and implemented for the ρ-VEX processor to mitigate single event upset errors. The fault tolerance can be included/excluded in the processor at design-time and/or enabled/disabled at run-time.

In Chapter 6, we evaluated our reconfigurable processor designs presented in the previous chapters in terms of performance and power/energy consumption. We utilized the 2-4-issue and 2-4-8-issue processors in different configurations (2-issue, 4-issue, and 8-issue) and used different application benchmark suites (MiBench, PowerStone, and a custom-made benchmark suite). We evaluated the effectiveness of the run-time task migration among different cores for the 2-4-8-issue processor. Additionally, we analyzed the effect of simultaneous reconfiguration of issue-width and instruction cache on the performance, dynamic energy consumption, and EDP for different benchmark applications. Finally, we evaluated how increasing the number of read/write ports on data memory/cache affects its capacity and the required hardware resources.

7.2 Main Contributions

In this section, we highlight the main contributions of our research that is described in this dissertation:

- In order to merge programmability with reconfigurability, we proposed a programmable VLIW processor implemented in a reconfigurable hardware, such as FPGA. The processor can be adapted to the specific requirements (static and dynamic) of different applications.

- Different optimizations have been presented for the proposed processor. These include the different type of the multiported register file, different implementation styles for the interrupt system, a datapath sharing mechanism, the hardware multiplier, etc.
- An optimized instruction encoding scheme has been proposed in order to increase the available opcode space. A methodology to extend the instruction set of the processor has been presented. A set of different sub-word custom operations have been implemented that could be included at design-time.
- Dynamically reconfigurable multi-core processors comprising of multiple smaller cores have been proposed to target performance and power consumption characteristics at run-time. The processor can be used to exploit ILP, DLP, and TLP.
- A scheme for run-time task migration among different cores of the multi-core processor has been proposed for performance improvement or power reduction at run-time. With this scheme, cores can be combined or split even when they are not idle, and hence, can be utilized more efficiently.
- Hardware-based configurable fault tolerance techniques have been presented to mitigate SEU errors in the proposed processors. The fault tolerance in the processor can be included/excluded at design-time. The included fault tolerance can be made permanently enabled or enabled/disabled at run-time.
- The impact of simultaneous reconfiguration of issue-width and instruction cache on performance, energy consumption, and EDP have been evaluated. The results showed that instead of reconfiguring either the issue-width or the instruction cache alone, reconfiguring both together has more potential to improve the performance, energy consumption, and/or EDP.

7.3 Future Research Directions

In this dissertation, we have proposed an adaptable processor that can be tuned to the requirements of different applications both at design-time as well as at run-time. The proposed approach combines the benefits of programmability

and reconfigurability. Following are some possible future research directions in which the introduced approach could progress:

- Currently, the ρ-VEX processor has 5 stages. In order to increase the clock frequency, the number of processor stages can be increased. For example, the decode stage can be split into a decode stage and operands read stage. The execution stages can also be split over multiple stages. Increasing the number of processor stages complicates the design of the forwarding network. The situation becomes worse with increasing the issue-width of the processor. The advantage in our case is that we can simulate the application with different latencies for the execution units. Hence, based on the required criteria (performance, hardware area, power consumption, etc.), we can select to implement a partial forwarding network in order to balance its complexity and performance making the design highly customizable.

- The ρ-VEX processor implements the complete VEX instruction set. Because the instruction set is very rich, implementing all of the defined operations increases the hardware area as well as power consumption. In order to generate optimize application-specific processors, the inclusion of the hardware for the required-only operations can be made design-time selectable. The application can be profiled to determine the used operations in the program. This information can then be used to select only the required operations when implementing the processor.

- In FPGAs, the standard clock gating techniques are not efficient and cannot avoid the power consumed in the clock networks which accounts for a considerable amount of the total consumed power. Partial reconfiguration based structural clock-gating technique [135] can be implemented for the ρ-VEX processor. The technique is based on the dynamic partial reconfiguration of the configuration memory frames related to the clock routing resources in FPGA. A small hardware controller can be implemented to perform the reconfiguration process which can be controlled by decoding a special instruction on the processor.

- The 2-4-8-issue processor consists of multiple 2-issue cores which can be used in different configurations. Run-time algorithms can be implemented to schedule different tasks on the processor. The algorithms can use compile-time and run-time information (such as performance, power consumption, etc.) to properly configure the available cores depending upon the tasks in the task queue. Hence, the cores can be efficiently

utilized and performance and power consumption can be optimized at run-time. The algorithms can be implemented in a special hardware or in software executing on a core.

- In this dissertation, we have analyzed the effect of simultaneous reconfiguration of issue-width and instruction cache on the performance, energy consumption, and EDP. A similar analysis can be performed for the data cache. Both the instruction and data caches can be included in the analysis to extend the scope. Because increasing the number of load/store units can increase the performance, therefore, data caches with multiple read/write ports can be considered in the analysis. Run-time algorithms can be implemented to perform the reconfiguration of the caches and the issue-width when required depending upon different parameters gathered at run-time or compile-time.

- In a different project, the AMBA AHB/APB bus protocol has been implemented for the ρ-VEX processor. This setup can be extended to implement a ρ-VEX based complete system-on-chip (SoC). With the bus implementation, it becomes simple to integrate different peripheral components, such as caches, interrupt system, UARTs, timers, and other I/O components. Memory management unit (MMU), direct memory access (DMA) unit, and other advanced components can be implemented in order to run an operating system on the processor. This could lead to the development of a multi-core based high-performance SoC.

- Support for adding custom operations at run-time by means of partial reconfiguration could be investigated. With partial reconfiguration, hardware resources can be shared among different functionalities/implementations. Hardware accelerators or even large co-processors can be defined and connected to the processor at run-time to off-load compute-intensive tasks.

- Hybrid (software and hardware) approaches for fault detection and recovery can be explored. The 2-4-8-issue processor can be configured to run a code in duplicate (two 4-issue cores) or triplicate (three 2-issue cores). With a slight modification in the micro-architecture, the results can be compared for error detection. With the task migration scheme presented in Section 4.3 and the generic binaries [80], the running code can be migrated from a faulty core to a non-faulty core.

Bibliography

[1] Hewlett-Packard Laboratories. VEX Toolchain. http://www.hpl.hp.com/downloads/vex/.

[2] M.R. Guthaus, J.S. Ringenberg, D. Ernst, T.M. Austin, T. Mudge, and R.B. Brown. MiBench: A Free, Commercially Representative Embedded Benchmark Suite. In *International Workshop on Workload Characterization (WWC)*, pages 3–14, 2001.

[3] A. Malik, B. Moyer, and D. Cermak. A Low Power Unified Cache Architecture Providing Power and Performance Flexibility. In *International Symposium on Low Power Electronics and Design (ISLPED)*, pages 241–243, 2000.

[4] J. Fisher, P. Faraboschi, and C. Young. Embedded Computing: A VLIW Approach to Architecture, Compilers and Tools. *Morgan Kaufmann*, 2005, ISBN: 1558607668.

[5] B.V. Iyer, "Length Adaptive Processors: A Solution for the Energy/Performance Dilemma in Embedded Systems. *Ph.D. Thesis*, North Carolina State University, 2009.

[6] Celoxica Inc. Handel-C Language Overview, http://www.celoxica.com/.

[7] Celoxica Inc. DK Design Suite 5. http://www.celoxica.com/.

[8] S. Vassiliadis, S. Wong, G.N. Gaydadjiev, K.L.M. Bertels, G.K. Kuzmanov, and E. M. Panainte. The MOLEN Ploymorphic Processor. *IEEE Transactions on Computers*, 53(11):1363–1375, 2004.

[9] Trimaran: An Infrastructure for Research in Backend Compilation and Architecture Exploration. http://www.trimaran.org/.

[10] P. Faraboschi, G. Brown, J.A. Fisher, G. Desoli, and F. Homewood. Lx: A Technology Platform for Customizable VLIW Embedded Processing. In *International Symposium on Computer Architecture (ISCA)*, pages 203–213, 2000.

[11] P.G. Lowney, S.M. Freudenberger, T.J. Karzes, W.D. Lichtenstein, R.P. Nix, J.S. O'Dondell, and J.C. Ruttenberg. The Multiflow Trace Scheduling Compiler. *Journal of Supercomputing*, 7:51–142, 1993.

[12] J. Fisher. Trace Scheduling: A Technique for Global Microcode Compaction. *IEEE Transactions on Computers*, C-30(7):478–490, 1981.

[13] S. Wong and F. Anjam. The Delft Reconfigurable VLIW Processor. In *International Conference on Advanced Computing and Communications (ADCOM)*, pages 242–251, 2009.

[14] S. Wong, T.V. As, and G. Brown. ρ-VEX: A Reconfigurable and Extensible Softcore VLIW Processor. In *International Conference on Field Programmable Technologies (FPT)*, pages 369–372, 2008.

[15] R.A.E. Seedorf, F. Anjam, A.A.C. Brandon, and S. Wong. Design of a Pipelined and Parameterized VLIW Processor: ρ-VEX v.2.0. In *HiPEAC Workshop on Reconfigurable Computing (WRC)*, 2012.

[16] Xilinx Inc. MicroBlaze Processor Reference Guide. http://www.xilinx.com/.

[17] IBM CoreConnect. http://www-3.ibm.com/chips/products/coreconnect/.

[18] Altera Inc. Nios-II Processor Reference Handbook. http://www.altera.com/.

[19] Lattice Semiconductore Inc. LatticeMico32 Processor. http://www.latticesemi.com/mico32/.

[20] WISHBONE System-on-Chip (SoC) Interconnection Architecture for Portable IP Cores. http://www.opencores.org/.

[21] Aeroflex Gaisler Inc. LEON Synthesizable Processors. http://www.gaisler.com/cms/.

[22] ARM Ltd. AMBA Open Specifications. http://www.arm.com/products/system-ip/amba/amba-open-specifications.php/.

[23] D. Lampret. OpenRISC 1200 IP Core Specification, 2001. http://www.opencores.org/.

[24] C. Iseli and E. Sanchez. Spyder: A Reconfigurable VLIW Processor using FPGAs. In *FPGAs for Custom Computing Machines (FCCM)*, pages 17–24, 1993.

[25] C. Iseli and E. Sanchez. Spyder: A SURE (SUperscalar and REconfigurable) Processor. *Journal of Supercomputing*, 9(3):231–52, 1995.

[26] V. Brost, F. Yang, and M. Paindavoine. A Modular VLIW Processor. In *International Symposium on Circuits and Systems (ISCAS)*, pages 3968–3971, 2007.

[27] Texas Instruments. TMS320C6201 Fixed-point Digital Signal Processor. SPRS051H, 2004.

[28] A. Lodi, M. Toma, F. Campi, A. Cappelli, and R. Canegallo. A VLIW Processor with Reconfigurable Instruction Set for Embedded Applications. *IEEE Journal on Solid-State Circuits*, 38(11):1876–1886, 2003.

[29] A.K. Jones, R. Hoare, D. Kusic, J. Fazekas, and J. Foster. An FPGA-based VLIW Processor with Custom Hardware Execution. In *International Symposium on Field Programmable Gate Arrays (FPGA)*, pages 107–117, 2005.

[30] C. Grabbe, M. Bednara, J.V.Z. Gathen, J. Shokrollahi, and J. Teich. A High Performance VLIW Processor for Finite Field Arithmetic. In *International Parallel and Distributed Processing Symposium (IPDPS)*, pages 189.2, 2003.

[31] M. Koester, W. Luk, and G. Brown. A Hardware Compilation Flow For Instance-Specific VLIW Cores. In *International Conference on Field Programmable Logic and Applications (FPL)*, pages 619–622, 2008.

[32] M.A.R. Saghir, M. El-Majzoub, and P. Akl. Customizing the Datapath and ISA of Soft VLIW Processors. In *High Performance Embedded Architectures and Compilers (HiPEAC)*, LNCS 4367, pages 276-290, 2007.

[33] D.A. Patterson and J.L. Hennessey. Computer Organization and Design: The Hardware/Software Interface. *Morgan Kaufmann*, Third Edition, 2005, ISBN: 1558606041.

[34] M. Schlansker and B. Rau. EPIC: Explicitly Parallel Instruction Computing. *IEEE Computer*, 33(2):37–45, 2000.

[35] W. Chu, R. Dimond, S. Perrott, S. Seng, and W. Luk. Customisable EPIC Processor: Architecture and Tools. In *Design, Automation, and Test in Europe Conference (DATE)*, pages 236–241, 2004.

[36] R. Seshasayanan and S.K. Srivatsa. A Novel Architecture for VLIW Processor. *Academic Open Internet Journal*, vol. 21, 2007, ISSN: 1311-4360.

[37] B. Mei, S. Vernalde, D. Verkest, H.D. Man, and R. Lauwereins. ADRES: An Architecture with Tightly Coupled VLIW Processor and Coarse-Grained Reconfigurable Matrix. In *Field Programmable Logic and Applications (FPL)*, LNCS 2778, pages 61–70, 2003.

[38] B. Mei, S. Vernalde, D. Verkest, H.D. Man, and R. Lauwereins. DRESC: A Retargetable Compiler for Coarse-grained Reconfigurable Architectures. In *International Conference on Field Programmable Technology (FPT)*, pages 166–173, 2002.

[39] A. Wolfe and J.P. Shen. A Variable Instruction Stream Extension to the VLIW Architecture. In *Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, pages 2–14, 1991.

[40] N. Clark, H. Zhong, K. Fan, S. Mahlke, K. Flautner, and K.V. Nieuwenhove. OptimoDE: Programmable Accelerator Engines through Retargetable Customization. In *Hot Chips*, 2004.

[41] Tensilica Inc. Xtensa LX4 Customizable DPU. http://http://www.tensilica.com/.

[42] R. Koenig, T. Stripf, J. Heisswolf, and J. Becker. A Scalable Microarchitecture Design that Enables Dynamic Code Execution for Variable-Issue Clustered Processors. In *International Parallel and Distributed Processing Symposium (IPDPS-RAW)*, pages 150–157, 2011.

[43] T. Stripf, R. Koenig, P. Rieder, and J. Becker. A Compiler Back-End for Reconfigurable, Mixed-ISA Processors with Clustered Register Files. In *International Parallel and Distributed Processing Symposium (IPDPS-RAW)*, pages 462–469, 2012.

[44] H. Corporaal. Microprocessor Architectures: From VLIW to TTA. *John Wiley & Sons*, 1997, ISBN: 047197157X.

[45] H. Corporaal and P. van der Arend. MOVE32INT, a Sea of Gates realization of a high performance Transport Triggered Architecture. *Microprocessing and Microprogramming*, 38(1-5):53–60, 1993.

[46] ST200 VLIW Series - ST231 Core and Instruction Set Architecture Manual, 2004. http://www.st.com/.

[47] S. Rathnam and G. Slavenburg. Processing the New World of Interactive Media. The Trimedia VLIW CPU Architecture. *IEEE Signal Processing Magazine*, 15(2):108–117, 1998.

[48] A. Suga and K. Matsunami. Introducing the FR500 Embedded Microprocessor. *IEEE Micro*, 20(4):21–27, 2000.

[49] Texas Instruments. TMS320C6211, Fixed-point Digital Dignal Processor, SPRS073, 1998.

[50] C. Basoglu, R.J. Gove, K. Kojima, and J. O'Donnell. A Single-chip Processor for Media Applications: The MAP1000. *International Journal of Imaging Systems and Technology*, 10:96–106, 1999.

[51] L. Greppert and T.S. Perry. Transmeta's Magic Show. *IEEE Spectrum*, 37(5):26–33, 2000.

[52] Altera Inc. Advanced Synthesis Cookbook: A Design Guide for Stratix II, Stratix III, and Stratix IV Devices. 2009. http://www.altera.com.

[53] C.E. LaForest and J.G. Steffan. Efficient Multi-ported Memories for FPGAs. In *International Symposium on Field Programmable Gate Arrays (FPGA)*, pages 41–50, 2010.

[54] Xilinx Inc. Quad-Port Memories in Virtex Devices. *Application Note XAPP228*, 2002. http://www.xilinx.com.

[55] M.A.R. Saghir and R. Naous. A Configurable Multi-ported Register File Architecture for Soft Processor Cores. In *International Symposium on Applied Reconfigurable Computing (ARC)*, LNCS 4419, pages 14–25, 2007.

[56] Steven Muchnick. Advanced Compiler Design and Implementation. *Morgan Kaufmann*, 1997, ISBN: 1558603204.

[57] J.L. Hennessy and D.A. Petterson. Computer Architecture: A Quantitative Approach. *Morgan Kaufmann*, Third Edition, 2002, ISBN: 1558605967.

[58] F. Anjam, S. Wong, and M.F. Nadeem. A Multiported Register File with Register Renaming for Configurable Softcore VLIW Processors. In *International Conference on Field Programmable Technology (FPT)*, pages 403–408, 2010.

[59] F. Anjam, Q. Kong, R.A.E. Seedorf, and S. Wong. On the Implementation of Traps for a Softcore VLIW Processor. In *HiPEAC Workshop on Reconfigurable Computing (WRC)*, 2012.

[60] F. Anjam, Q. Kong, R.A.E. Seedorf, and S. Wong. A Run-time Task Migration Scheme for an Adjustable Issue-slots Multi-core Processor. In *International Symposium on Applied Reconfigurable Computing (ARC)*, pages 102–113, 2012.

[61] F. Anjam, S. Wong, and M.F. Nadeem. A shared Reconfigurable VLIW Multiprocessor System. In *International Parallel and Distributed Processing Symposium (IPDPS-RAW)*, pages 1–8, 2010.

[62] Altera Inc. Tutorial: Creating Multiprocessor Nios II Systems. 2007. http://www.altera.com.

[63] A. Hung, W. Bishop, and A. Kennings. Symmetric Multiprocessing on Programmable Chips Made Easy. In *Design, Automation, and Test in Europe Conference (DATE)*, pages 240–245, 2005.

[64] G.G. Mplemenos and I. Papaefstathiou. MPLEM: An 80-processor FPGA Based Multiprocessor System. In *International Symposium on Field-Programmable Custom Computing Machines (FCCM)*, pages 273–274, 2008.

[65] S. Xu and H.P. Smith. A Multi-MicroBlaze Based SOC System: From SystemC Modeling to FPGA Prototyping. In *International Symposium on Rapid System Prototyping (RSP)*, pages 121–127, 2008.

[66] K. Ravindran, N. Satish, Y. Jin, and K. Keutzer. An FPGA-Based Soft Multiprocessor System for IPv4 Packet Forwarding. In *International Conference on Field Programmable Logic and Applications (FPL)*, pages 487–492, 2005.

[67] M. Hubner, K. Paulsson, and J. Becker. Parallel and Flexible Multiprocessor System-On-Chip for Adaptive Automotive Applications based on Xilinx MicroBlaze Soft-Cores. In *International Parallel and Distributed Processing Symposium (IPDPS)*, workshop 3, vol. 4, pages 149a, 2005.

[68] P. Huerta, J. Castillo, J.I. Martinez, and V. Lopez. A MicroBlaze Based Multiprocessor SoC. *WSEAS Transactions on Circuits and Systems*, 4(5):423–430, 2005.

[69] F. Anjam, S. Wong, and M.F. Nadeem. A VLIW Softcore Processor with Dynamically Adjustable Issue-slots. In *International Conference on Field Programmable Technology (FPT)*, pages 393–398, 2010.

[70] F. Anjam, M. Nadeem, and S. Wong. Targeting Code Diversity with Run-time Adjustable Issue-slots in a Chip Multiprocessor. In *Design, Automation, and Test in Europe Conference (DATE)*, pages 1358–1363, 2011.

[71] Xilinx Inc. Early Access Partial Reconfiguration User Guide. *User Guide UG208*, 2006. http://www.xilinx.com.

[72] H. Zhong, S. A. Lieberman, and S. A. Mahlke. Extending Multicore Architectures to Exploit Hybrid Parallelism in Single-thread Applications. In *International Symposium on High Performance Computer Architecture (HPCA)*, pages 25–36, 2007.

[73] M.B. Taylor et al. Evaluation of the Raw Microprocessor: An Exposed-wire-delay Architecture for ILP and Streams. In *International Symposium on Computer Architecture (ISCA)*, pages 2–13, 2004.

[74] K. Sankaralingam, R. Nagarajan, H. Liu, C. Kim, J. Huh, D. Burger, S. W. Keckler, and C. R. Moore. Exploiting ILP, TLP, and DLP with the Polymorphous TRIPS Architecture. In *International Symposium on Computer Architecture (ISCA)*, pages 422–433, 2003.

[75] E. Ipek, M. Kirman, N. Kirman, and J.F. Martinez. Core Fusion: Accommodating Software Diversity in Chip Multiprocessors. *ACM SIGARCH Computer Architecture News*, 35(2):186–197, 2007.

[76] K. Mai, T. Paaske, N. Jayasena, R. Ho, W. J. Dally, and M. Horowitz. Smart Memories: A Modular Reconfigurable Architecture. In *International Symposium on Computer Architecture (ISCA)*, pages 161–171, 2000.

[77] M. Fillo et al. The M-Machine Multicomputer. In *International Symposium on Microarchitecture (Micro)*, pages 146–156, 1995.

[78] W.F. Lee. VLIW Microprocessor Hardware Design For ASICs and FPGA. *McGraw-Hill*, 2007, ISBN: 0071497021.

[79] S. Wong, F. Anjam, and M.F. Nadeem. Dynamically Reconfigurable Register File for a Softcore VLIW Processor. In *Design, Automation, and Test in Europe Conference (DATE)*, pages 969–972, 2010.

[80] A. Brandon and S. Wong. Support for Dynamic Issue Width in VLIW Processors using Generic Binaries. In *Design, Automation, and Test in Europe Conference (DATE)*, pages 827–832, 2013.

[81] J.M. Smith. A Survey of Process Migration Mechanisms. *ACM SIGOPS Operating Systems Review*, 22(3):29–40, 1988.

[82] M. Richmond and M. Hitchens. A New Process Migration Algorithm. *ACM SIGOPS Operating Systems Review*, 31(1):31–42, 1997.

[83] K.M. Katre, H. Ramaprasad, A. Sarkar, and F. Mueller. Policies for Migration of Real-Time Tasks in Embedded Multi-Core Systems. In *Real-Time Systems Symposium (RTSS)*, pages 17–20, 2009.

[84] A. Acquaviva, A. Alimonda, S. Carta, and M. Pittau. Assessing Task Migration Impact on Embedded Soft Real-Time Streaming Multimedia Applications. *EURASIP Journal on Embedded Systems*, 9:1–15, 2008.

[85] J. Jahn, M.A.A. Faruque, and J. Henkel. CARAT: Context-Aware Runtime Adaptive Task Migration for Multi Core Architectures. In *Design, Automation, and Test in Europe Conference (DATE)*, pages 1–6, 2011.

[86] A. Sarkar, F. Mueller, H. Ramaprasad, and S. Mohan. Push-Assisted Migration of Real-Time Tasks in Multi-Core Processors. In emphConference on Languages, Compilers, and Tools for Embedded Systems (LCTES), pages 80–89, 2009.

[87] D. Cuesta, J.L. Ayala, J.I. Hidalgo, D. Atienza, A. Acquaviva, and E. Macii. Adaptive Task Migration Policies for Thermal Control in MPSoCs. In *International Symposium on VLSI (ISVLSI)*, pages 110–115, 2010.

[88] Y. Ge, P. Malani, and Q. Qiu. Distributed Task Migration for Thermal Management in Many-Core Systems. In *Design Automation Conference (DAC)*, pages 579–584, 2010.

[89] L. Zheng. A Task Migration Constrained Energy-Efficient Scheduling Algorithm for Multiprocessor Real-time Systems. In *International Conference on Wireless Communications, Networking and Mobile Computing (WiCom)*, pages 3055–3058, 2007.

[90] E. Seo, J. Jeong, S. Park, and J. Lee. Energy Efficient Scheduling of Real-Time Tasks on Multicore Processors. emphTransactions on Parallel and Distributed Systems, 19(11):1540–1552, 2008.

[91] E.W. Briao, D. Barcelos, F. Wronski, and F.R. Wagner. Impact of Task Migration in NoC-based MPSoCs for Soft Real-time Applications. In *International Conference on VLSI-SoC*, pages 296-299, 2007.

[92] O. Ozturk, M. Kandemir, S.W. Son, and M. Karakoy. Selective Code/Data Migration for Reducing Communication Energy in Embedded MpSoC Architectures. In *Great Lakes Symposium on VLSI (GLSVLSI)*, pages 386–391, 2006.

[93] T. Li, P. Brett, B. Hohlt, R. Knauerhase, S. McElderry, and S. Hahn. Operating System Support for Shared-ISA Asymmetric Multi-core Architectures. In *Workshop on the Interaction between Operating Systems and Computer Architecture (WIOSCA)*, pages 19–26, 2008.

[94] R. Balasubramonian, D.H. Albonesi, A. Buyuktosunoglu, and S. Dwarkadas. Memory Hierarchy Reconfiguration for Energy and Performance in General-purpose Processor Architectures. In *International Symposium on Microarchitecture (Micro)*, pages 245–257, 2000.

[95] J. Yang, R. Gupta, and J.F. Martinez. Energy Efficient Frequent Value Data Cache Design. In *International Symposium on Microarchitecture (Micro)*, pages 197–207, 2002.

[96] D.H. Albonesi. Selective Cache Ways: On Demand Cache Resource Allocation. In *International Symposium on Microarchitecture (Micro)*, pages 248–259, 1999.

[97] S. Segars. Low Power Design Techniques for Microprocessors. In *International Solid-State Circuits Conference (ISSCC) - Tutorial*, 2001.

[98] C. Zhang, F. Vahid, and W. Najjar. A Highly Configurable Cache Architecture for Embedded Systems. In *International Symposium on Computer Architecture (ISCA)*, pages 136–146, 2003.

[99] A.V. Veidenbaum, W. Tang, R. Gupta, A. Nicolau, and X. Ji. Adapting Cache Line Size to Application Behavior. In *International Conference on Supercomputing (SC)*, pages 145–154, 1999.

[100] C. Zhang, F. Vahid, and W. Najjar. Energy Benefits of a Configurable Line Size Cache for Embedded Systems. In *International Symposium on VLSI (ISVLSI)*, pages 87–91, 2003.

[101] K. Inoue, T. Ishihara, and K. Murakami. Way-Predictive Set-Associative Cache for High Performance and Low Energy Consumption. In *International Symposium On Low Power Electronics and Design (ISLPED)*, pages 273–275, 1999.

[102] P. Ranganathan, S. Adve, and N.P. Jouppi. Reconfigurable Caches and Their Application to Media Processing. In *International Symposium on Computer Architecture (ISCA)*, pages 214–224, 2000.

[103] F. Anjam, L. Carro, S. Wong, G.L. Nazar, and M.B. Rutzig. Simultaneous Reconfiguration of Issue-width and Instruction Cache for a VLIW Processor. In *International Conference on Embedded Computer Systems: Architecture Modeling and Simulation (SAMOS)*, pages 183–192, 2012.

[104] CACTI: An Integrated Cache and Memory Access Time, Cycle Time, Area, Leakage, and Dynamic Power Model. http://www.hpl.hp.com/research/cacti/.

[105] NXP Semiconductor. TriMedia Processor Series. http://www.nxp.com/.

[106] ST240 Core and Instruction Set Architecture Manual. http://www.st.com/.

[107] Fujitsu Ltd. FR450 Series VLIW Embedded Microprocessor. http://www.fujitsu.com/.

[108] T. Jungeblut, R. Dreesen, M. Porrmann, U. Ruckert, and U. Hachmann. Design Space Exploration for Resource Efficient VLIW-Processors. In *University Booth of the Design, Automation, and Test in Europe Conference (DATE)*, 2008.

[109] D. Benitez, J.C. Moure, D. Rexachs, and E. Luque. A Reconfigurable Cache Memory with Heterogeneous Banks. In *Design, Automation, and Test in Europe Conference (DATE)*, pages 825–830, 2010.

[110] Texas Instruments. TMS320C6211 Cache Analysis. *Application Report SPRA472*, 1998.

[111] T. Givargis and F. Vahid. Tuning of Cache Ways and Voltage for Low-Energy Embedded System Platforms. *Journal of Design Automation for Embedded Systems*, 7(1–2):35–51, 2002.

[112] M. Powell, A. Agarwal, T.N. Vijaykumar, B. Falsafi, and K. Roy. Reducing Set-associative Cache Energy via Way-prediction and Selective Direct-mapping. In *International Symposium on Microarchitecture (Micro)*, pages 54–65, 2001.

[113] K.T. Sundararajan, T.M. Jones, and N. Topham. A Reconfigurable Cache Architecture for Energy Efficiency. In *International Conference on Computing Frontiers (CF)*, no. 9, pages 1–2, 2011.

[114] S.H. Yang, M.D. Powell, B. Falsafi, and T.N. Vijaykumar. Exploiting Choice in Resizable Cache Design to Optimize Deep-Submicron Processor Energy-Delay. In *International Conference on High-Performance Computer Architecture (HPCA)*, pages 151–161, 2002.

[115] C. Zhang, F. Vahid, and R. Lysecky. A Self-Tuning Cache Architecture for Embedded Systems. *ACM Transactions on Embedded Computing Systems*, 3(2):407–425, 2004.

[116] F. Anjam and S. Wong. Configurable Fault-Tolerance for a Configurable VLIW Processor. In *International Symposium on Applied Reconfigurable Computing (ARC)*, pages 167–178, 2013.

[117] J.B. Nickle and A.K. Soman. REESE: A Method of Soft Error Detection in Microprocessors. In *International Conference on Dependable Systems and Networks (DSN)*, pages 401–410, 2001.

[118] C. Bolchini. A Software Methodology for Detecting Hardware Faults in VLIW Data Paths. *IEEE Transactions on Reliability*, 52(4):458–468, 2003.

[119] J.S. Hu, F. Li, V. Degalahal, M. Kandemir, N. Vijaykrishnan, and M.J. Irwin. Compiler-Directed Instruction Duplication for Soft Error Detection. In *Design, Automation, and Test in Europe Conference (DATE)*, pages 1056–1057, 2005.

[120] J.G. Holm and P. Banerjee. Low Cost Concurrent Error Detection in a VLIW Architecture using Replicated Instructions. In *International Conference on Parallel Processing (ICPP)*, pages 192–195, 1992.

[121] D.M. Blough and A. Nicolau. Fault Tolerance in Super-scalar and VLIW Processors. In *IPDPS Workshop on Fault-Tolerant Parallel and Distributed Systems*, pages 193–200, 1992.

[122] N. Oh, P.P. Shirvani, and E.J. McCluskey. Error Detection by Duplicated Instructions in Super-scalar Processors. *IEEE Transactions on Reliability*, 51(1):63–75, 2002.

[123] L. Sterpone, D. Sabena, S. Campagna, and M.S. Reorda. Fault Injection Analysis of Transient Faults in Clustered VLIW Processors. In *International Symposium on Design and Diagnostics of Electronic Circuits and Systems (DDECS)*, pages 207–212, 2011.

[124] Y. Ichinomiya, S. Tanoue, T. Ishida, M. Amagasaki, M. Kuga, and T. Sueyoshi. Memory Sharing Approach for TMR Softcore Processor. In *International Conference on Applied Reconfigurable Computing (ARC)*, pages 268–274, 2009.

[125] V. Vasudevan, P. Waldeck, H. Mehta, and N. Bergmann. Implementation of Triple Modular Redundant FPGA based Safety Critical System for Reliable Software Execution. In *SCS Australian Workshop on Safety-Related Programmable Systems*, pages 113–119, 2006.

[126] M. Franklin. A Study of Time Redundant Fault Tolerance Techniques for Superscalar Processors. In *International Workshop on Defect and Fault Tolerance in VLSI Systems (DFT)*, pages 207–215, 1995.

[127] F. Rashid, K.K. Saluja and P.A. Ramanathan. Fault Tolerance Through Re-execution in Multiscalar Architecture. In *International Conference on Dependable Systems and Networks (DSN)*, pages 482–491, 2000.

[128] T. Sato and I. Arita. Evaluating Low-cost Fault-tolerance Mechanism for Microprocessors on Multimedia Applications. In *Pacific Rim International Symposium on Dependable Computing (PRDC)*, pages 225–232, 2001.

[129] Y.Y. Chen and K.L. Leo. Reliable Data Path Design of VLIW Processor Cores with Comprehensive Error-coverage Assessment. *Microprocessors and Microsystems*, 34:49–61, 2010.

[130] J. Gaisler. A Portable and Fault-Tolerant Microprocessor Based on the SPARC V8 Architecture. In *International Conference on Dependable Systems and Networks (DSN)*, pages 409–415, 2002.

[131] M. Scholzel and S. Mulleri. Combining Hardware- and Software-Based Self-Repair Methods for Statistically Scheduled Data Paths. In *International Symposium on Defect and Fault Tolerance in VLSI Systems (DFT)*, pages 90–98, 2010.

[132] P. Liden, P. Dahlgren, R. Johansson, and J. Karlsson. On Latching Probability of Particles Induced Transients in Combinational Networks. In

International Symposium on Fault-Tolerant Computing (FTCS), pages 340–349, 1994.

[133] E. Touloupis, J.A. Flint, V.A. Chouliaras, and D.D. Ward. Study of the Effects of SEU-Induced Faults on a Pipeline-Protected Microprocessor. *IEEE Transactions on Computers*, 56(12):1585–1596, 2007.

[134] Xilinx Inc. Virtex-6 FPGA Clocking Resources. *User Guide UG362*, 2010. http://www.xilinx.com.

[135] L. Sterpone, L. Carro, D. Matos, S. Wong, and F. Anjam. A New Reconfigurable Clock-gating Technique for Low Power SRAM-based FPGAs. In *Design, Automation, and Test in Europe Conference (DATE)*, pages 1–6, 2011.

List of Publications

International Journals

1. **F. Anjam**, Q. Kong, R.A.E. Seedorf, and S. Wong. A Task Migration Scheme for a Run-time Adjustable Issue-slots Multi-core Processor. Submitted to *Elsevier Journal of Systems Architecture (JSA)*.

2. S. Wong, **F. Anjam**, A.A.C. Brandon, R.A.E. Seedorf, L. Carro, D. Matos, R. Giorgi, A. Scionti, S. Kavvadias, G. Keramidas, C. Scordino, S.A. McKee, B. Goel, and F. Papariello. ERA: An Integrated Dynamically Adaptive Platform. Submitted to *IEEE Micro (Special Issue on Reconfigurable Computing)*.

International Conferences/Workshops

1. G. Keramidas, S. Wong, **F. Anjam**, A.A.C. Brandon, R.A.E. Seedorf, C. Scordino, L. Carro, D. Matos, R. Giorgi, S. Kavvadias, S.A. McKee, B. Goel, and V. Spiliopoulos. Embedded Reconfigurable Computing: the ERA Approach. To appear in *Int. Conference on Industrial Informatics (INDIN)*, Bochum, Germany, July 2013.

2. **F. Anjam** and S. Wong. Configurable Fault-tolerance for a Configurable VLIW Processor. In *Int. Symposium on Applied Reconfigurable Computing (ARC)*, pages 167–178, Los Angeles, USA, March 2013.

3. **F. Anjam**, L. Carro, S. Wong, G.L. Nazar, and M.B. Rutzig. Simultaneous Reconfiguration of Issue-width and Instruction Cache for a VLIW Processor. In *Int. Conference on Embedded Computer Systems: Architecture Modeling and Simulation (SAMOS)*, pages 183–191, Samos, Greece, July 2012.

4. **F. Anjam,** Q. Kong, R.A.E. Seedorf, and S. Wong. A Run-time Task Migration Scheme for an Adjustable Issue-slots Multi-core Processor. In *Int. Symposium on Applied Reconfigurable Computing (ARC)*, pages 102–113, Hong Kong, March 2012.

5. **F. Anjam,** Q Kong, R.A.E. Seedorf, and S. Wong. On the Implementation of Traps for a Softcore VLIW Processor. In *HiPEAC Workshop on Reconfigurable Computing (WRC)*, Paris, France, January 2012.

6. R.A.E. Seedorf, **F. Anjam**, A.A.C. Brandon, and S. Wong. Design of a Pipelined and Parameterized VLIW Processor: ρ-VEX v.2.0. In *HiPEAC Workshop on Reconfigurable Computing (WRC)*, Paris, France, January 2012.

7. S. Wong, A.A.C. Brandon, **F. Anjam**, R.A.E. Seedorf, R. Giorgi, N. Puzovic, S. McKee, L. Carro, and G. Keramidas. Early Results from ERA – Embedded Reconfigurable Architectures. In *Int. Conference on Industrial Informatics (INDIN)*, pages 816–822, Lisbon, Portugal, July 2011.

8. **F. Anjam**, M. Nadeem, and S. Wong. Targeting Code Diversity with Run-time Adjustable Issue-slots in a Chip Multiprocessor. In *Design, Automation and Test in Europe Conference (DATE)*, pages 1358–1363, Grenoble, France, March 2011.

9. **F. Anjam**, M. Nadeem, and S. Wong. A VLIW Softcore Processor with Dynamically Adjustable Issue-slots. In *Int. Conference on Field Programmable Technology (FPT)*, pages 393–398, Beijing, China, December 2010.

10. **F. Anjam**, S. Wong, and M.F. Nadeem. A Multiported Register File with Register Renaming for Configurable Softcore VLIW Processors. In *Int. Conference on Field Programmable Technology (FPT)*, pages 403–408, Beijing, China, December 2010.

11. S. Wong, **F. Anjam**, and M.F. Nadeem. Dynamically Reconfigurable Register File for a Softcore VLIW Processor. In *Design, Automation and Test in Europe Conference (DATE)*, pages 969–972, Dresden, Germany, March 2010.

12. S. Wong and **F. Anjam**. The Delft Reconfigurable VLIW Processor. In *Int. Conference on Advanced Computing and Communications (ADCOM)*, pages 244–251, Bangalore, India, December 2009.

Other Publications (International Conferences/Workshops)

1. P.C. Santos, G.L. Nazar, **F. Anjam**, S. Wong, D. Matos, L. Carro. A Fully Dynamic Reconfigurable NoC-based MPSoC: The Advantages of Total Reconfiguration. In *HiPEAC Workshop on Reconfigurable Computing (WRC)*, Berlin, Germany, January 2013.

2. P.C. Santos, G.L. Nazar, **F. Anjam**, S. Wong, D. Matos, L. Carro. A Fully Dynamic Reconfigurable NoC-based MPSoC: The Advantages of a Multi-Level Reconfiguration. In *HiPEAC Workshop on Design Tools and Architectures for Multi-Core Embedded Computing Platforms (DITAM)*, Berlin, Germany, January 2013.

3. P.C. Santos, G.L. Nazar, **F. Anjam**, S. Wong, and L. Carro. Adapting Communication for Adaptable Processors: A Multi-Axis Reconfiguration Approach. In *Int. Conference on ReConFigurable Computing and FPGAs (ReConFig)*, pages 1–6, Cancun, Mexico, December 2012.

4. L. Sterpone, D. Matos, L. Carro, S. Wong, and **F. Anjam**. A New Reconfigurable Clock-gating Technique for Low Power SRAM-based FPGAs. In *Design, Automation and Test in Europe Conference (DATE)*, pages 1–6, Grenoble, France, March 2011.

5. M.F. Nadeem, **F. Anjam**, S. A. Ostadzadeh, M. Ahmadi, and S. Wong. Towards the Utilization of Reconfigurable Processors in Grid Networks. In *Workshop on Circuits, Systems and Signal Processing (ProRISC)*, Veldhoven, The Netherlands, November 2010.

6. M. Nadeem, S. Wong, G. Kuzmanov, M. Shabbir, **F. Anjam**, and M.F. Nadeem. Low-power, High-throughput Deblocking Filter for H.264/AVC. In *Int. Symposium on System-on-Chip (SoC)*, pages 93–98, Tampere, Finland, September 2010.

Curriculum Vitae

Fakhar Anjam was born on February 05, 1978 in Karak, Pakistan. He did his B.Sc. in Electrical and Electronic Engineering from N.W.F.P University of Engineering and Technology, Peshawar, Pakistan in 2002. He graduated with M.Sc. in Information Technology (IT) from Pakistan Institute of Engineering and Applied Sciences (PIEAS), Islamabad, Pakistan in 2004. From 2004 to 2008, he worked as a system and hardware developer in a research and development (R&D) organization in Pakistan. He was responsible for R&D of real-time embedded systems mainly related to various communication, industrial monitoring and control applications. In 2008, he joined the Computer Engineering Lab, in EEMCS faculty at Delft University of Technology (TU Delft) for pursuing his PhD. His research project was funded by the Higher Education Commission (HEC) Pakistan and TU Delft. His research interests include computer architecture, reconfigurable computing, softcore processors, VLIW processors, and FPGA design and development. During his stay at TU Delft, he has worked for the EU FP7 ERA (Embedded Reconfigurable Architectures) project, co-supervised several student projects and reviewed various international scientific journals and conference papers. He has presented several scientific papers in international conferences. Two of his submitted journal papers are currently under review. The research conducted by him is presented in this thesis. Fakhar enjoys walking, biking, long driving, exploring new places, swimming, playing cricket and of course watching TV.

ISBN 978-94-6186-191-7