Software-oriented approach to hardware-software co-simulation for an FPGA-based RISC extensible processor

Tham, Kwang Sheun

2007

Tham, K. S. (2007). Software-oriented approach to hardware-software co-simulation for an FPGA-based RISC extensible processor. Master's thesis, Nanyang Technological University, Singapore.

<https://hdl.handle.net/10356/35739>

<https://doi.org/10.32657/10356/35739>

94-8035
156
2X
8/8/10 R.

Software-oriented Approach to Hardware-Software Co-Simulation for an FPGA-based RISC Extensible Processor

Tham Kwang Sheun

School of Computer Engineering

A thesis submitted to the Nanyang Technological University
in fulfilment of the requirement for the degree of
Master of Engineering

2007

ABSTRACT

Innovative embedded applications like video and internet streaming in the multimedia domain have greatly increased the computational workload of embedded systems in consumer products. To meet such demands, reconfigurable platforms are seen as a compromise to bridge the gap between hardware-based and software-based implementations and can deliver better flexibility while maintaining performance for different types of applications. Such reconfigurable systems present a difficult problem for current modeling platforms as a tightly-coupled co-design/simulation effort for both hardware and software must be integrated in the framework.

Recent advancements in mixed hardware-software modeling platforms have shown great potential in providing the use of a common software language for describing the hardware and software of the entire system. It enables a convenient platform for exploring the reconfigurable characteristics of the microarchitecture and at the same time the software generation requirements can also be examined. System/hardware designers and application developers can work and verify concurrently on the same design source and this ensures hardware and software considerations are complimented throughout the design flow. The use of this software-oriented design methodology will improve the productivity of embedded system design and must be investigated in detail with respect to enabling hardware-software co-simulation for reconfigurable systems.

ACKNOWLEDGEMENT

I would like to express my deepest appreciation to my supervisor Assoc. Prof. Douglas Maskell for giving me the opportunity to undertake this challenging research and also for his encouragement and guidance in providing valuable ideas to this project.

I would also like to thank Mr. Timothy Oliver and Mr. Chen Xiaoyong for their constructive suggestions and assistances rendered to me during the course of this work.

Lastly, my appreciation goes out to the staff members and fellow students in the Centre for High Performance Embedded Systems (CHiPES).

TABLE OF CONTENTS

ABSTRACT	<i>i</i>
ACKNOWLEDGEMENT	<i>ii</i>
TABLE OF CONTENTS	<i>iii</i>
LIST OF FIGURES	<i>v</i>
LIST OF TABLES	<i>vi</i>
ABBREVIATIONS	<i>vii</i>
CHAPTER 1 INTRODUCTION	1
1.1. Background	1
1.2. Objectives	3
1.3. Major contributions	4
1.4. Outline of the thesis	5
CHAPTER 2 LITERATURE REVIEW	6
2.1. Reconfigurable Architectures	7
2.1.1. FPGA for Custom Computing Machines (FCCM)	7
2.1.2. Coarse-grained Reconfigurable Architectures	9
2.1.3. Software Configurable Architectures	11
2.1.4. Polymorphous Computing Architectures (PCA)	12
2.2. Dynamic Run-time Systems	13
2.2.1. Resource Model	13
2.2.2. Hardware-Software Compilation	14
2.2.3. Scheduling Methods	19
2.3. FPGA Place-and-route tools	20
2.3.1. Limitation of conventional place-and-route tools	21
2.3.2. JBits and Xilinx Partial Reconfiguration Toolkit (XPART)	22
2.3.3. Versatile place-and-route (VPR)	24
2.4. Simulation Environments	24
2.5. Hardware-Software Co-simulation Platforms	26
2.6. Summary	29
CHAPTER 3 IMPLEMENTING FPGA-BASED RECONFIGURABLE CO-PROCESSORS IN SYSTEMC	31
3.1. System Architecture Model	31
3.2. Loosely-coupled Extension Model	33
3.2.1. SystemC RISC CPU and datapath interface to Co-processor	34
3.2.2. Co-processor model	40
3.3. Tightly-coupled Extension Model	47
3.4. Combination of tightly-coupled and loosely-coupled extension model	51

CHAPTER 4 DESIGN OF A CO-SIMULATION AND COMPILATION FRAMEWORK	54
4.1. Co-simulation Flow	54
4.2. Compilation Flow	58
4.2.1. Software Compilation	59
4.2.2. Hardware Compilation	61
CHAPTER 5 APPLICATION EXAMPLES AND EXPERIMENTAL RESULTS	64
5.1. Application Examples	64
5.2. Experimental Results	70
5.2.1. Mapping Structures	70
5.2.2. Configuration Patterns	71
5.2.3. Performance Analysis	72
CHAPTER 6 CONCLUSION AND RECOMMENDATIONS	78
6.1. Conclusion	78
6.2. Recommendations	80
REFERENCES	81
APPENDIX	
A1. Supported MicroBlaze instruction set	A1
A2. Supported Nios-II instruction set	A3
A3. FSL	A5
A4. Bit-serial PE	A6
A5. Custom instruction logic	A7

LIST OF FIGURES

2-1	A Run-time System Overview.	6
2-2	Resource models; (a) 2D model (b) 2D fixed block partition I 1D variable block partition (d) 1D fixed block partition	14
3-1	FPGA-based architectural model	31
3-2	Loosely-coupled extension model	32
3-3	Tightly-coupled extension model	33
3-4	Examples of the pipeline architecture	34
3-5	Block diagram of the FSL interface to the internal CPU datapath	35
3-6	Top-level interface description	36
3-7	Excerpt code listing of the decode unit	37
3-8	Excerpt code listing of the FSL unit	38
3-9	(a) Bit-serial PE block diagram (b) MAC-unit block diagram	40
3-10	PE interconnection pattern	41
3-11	Structural hierarchy of the PE array	42
3-12	Excerpt code listing of the bit-serial PE array interconnection	43
3-13	FSL resource allocation: (a) MAC-unit array (b) Bit-serial PE array	44
3-14	Block diagram of the extension control unit	45
3-15	(a) Custom logic-ALU connection (b) Extended multi-cycle custom instruction block diagram	47
3-16	A code listing of custom instruction interface	49
3-17	Block diagram of the MicroBlaze extension with custom functions	51
3-18	Block diagram of the ISE with MAC-unit array	52
4-1	Co-simulation and compilation framework	55
4-2	Example of a C code for 4x4 matrix multiplication	59
4-3	Pseudo assembly code: (a) MicroBlaze extension using FSL functions (b) Nios-II ISE using custom instructions	60
4-4	RISC CPU (3-stage) instruction schedule for hardware execution	61
5-1	A 1-D DCT mapping	65
5-2	Interleaving intermediate computed values	66
5-3	Alternative 4x4 matrix multiplication mapping	67
5-4	A basic FIR filter procedure	68
5-5	A M-tap FIR mapping	69
5-6	A 4x4 and 8x8 matrix multiplication configuration patterns	72
5-7	Compute time comparison of the various kernel tasks	74
5-8	Speedup comparison between the various models	74
5-9	Comparison between MicroBlaze with custom functions, MAC-unit and software-only implementation	76
5-10	Comparison between ISE, ISE with MAC-unit and software-only implementation	76

LIST OF TABLES

3-1	Custom instruction mode of operations for ISE with MAC-unit array	53
4-1	Characteristics of the various components in the extension models	56
5-1	Comparison between number of PEs and cycles for computation, I/O and reconfiguration	70
5-2	A comparison between bit-serial PE array and MAC-unit array for MicroBlaze extension model	73

ABBREVIATIONS

ADL	Architectural Description Language
ALU	Arithmetic Logic Unit
API	Application Programmable Interface
ASIC	Application Specific Integrated Circuits
CAD	Computer Aided Design
CFG	Control Flow Graph
CIL	Custom Instruction Logic
CLB	Configurable Logic Block
CPU	Central Processing Unit
CTR	Compile-time Reconfiguration
DCT	Discrete Cosine Transform
DSP	Digital Signal Processing
EDA	Electronic Design Automation
FCCM	FPGA Custom Computing Machines
FIFO	First-in-First-out
FIR	Finite Impulse Response
FPGA	Field Programmable Gate Array
FSL	Fast Simplex Link
HDL	Hardware Description Language
ILP	Instruction Level Parallelism
IP	Intellectual Property
ISA	Instruction Set Architecture
ISE	Instruction Set Extension
ISS	Instruction Set Simulator
JPEG	Joint Photographic Experts Group
LUT	Lookup Table
MAC	Multiple-Accumulate
MIMD	Multiple Instruction Multiple Data
MM	Matrix Multiplication
MPEG	Motion Picture Experts Group
OS	Operating System
OSCI	Open SystemC Initiative
PCA	Polymorphous Computing Architecture
PE	Processing Element
RISC	Reduced Instruction Set Computer
RTL	Register Transfer Level
RTOS	Real-Time Operating System
RTR	Run-time Reconfiguration
SIMD	Single Instruction Multiple Data
SoC	System-on-Chip
VLIW	Very Long Instruction Word

CHAPTER 1

INTRODUCTION

1.1 BACKGROUND

Solutions to some computational problems are almost always driven by the justifications of implementation cost and demanded speed. From a commercial standpoint, finding a balance between the two can be a crucial factor. This trade-off decision will move from the extreme end of Application Specific Integrated Circuits (ASICs) to computing systems that offer wider availability and comparable performance such as Field Programmable Gate Arrays (FPGA), FPGA for custom computing machines (FCCM), to programmable processors. Adaptability seems to be a trend in embedded system design driven by the emergence of portable devices. As these devices become more innovative in terms of their application, there is a need for systems to achieve flexibility (i.e. adaptation to a changing environment) with little or no compromise to performance. This leads to a reusable design with a shorter design cycle as these products have a limited lifespan and hence, FCCM may become an attractive option.

Hybrid embedded systems like FCCM that consist of a co-processor coupled together with an embedded Central Processing Unit (CPU) have shown great potential in accelerating a wide variety of applications especially in the multimedia domain. This type of system can be viewed as an extension to the general purpose CPU and presents a more appealing option as it is self contained and can be self-reconfigurable without any external host control. Xilinx Virtex Pro and Altera Excalibur devices are examples of systems combining a state-of-the-art FPGA part with a standard processor (e.g. PowerPC, ARM, etc.). The general purpose processor can also be a soft-core CPU (e.g. MicroBlaze, Nios-II, etc.) which provides the flexibility to explore different types of coupling interface between the CPU and the co-processor. Moreover, these FPGAs can support dynamic partial reconfiguration that permits online (run-time) configuration. It can be seen that the synthesis of a hybrid embedded system is divided into two parts; a software

component that controls the co-processor execution and the hardware component that represents the microarchitecture of the co-processor core. Dynamic reconfiguration can be achieved through the partial reconfiguration capability of the device or inherent in the microarchitecture of the co-processor. To facilitate design space exploration efficiently, it is imperative to adopt a tightly-coupled co-design/simulation effort for the hardware and software components in these hybrid systems.

In recent years, commercial Electronic Design Automation (EDA) vendors have invested heavily on hardware design tools to simplify the complexity of designing hardware and also in the process have improved the design quality. However, as the partition of the hardware and software components occurs at an early stage, refinements to obtain the best solution are still being performed separately and naturally results in integration issues where the system specifications or real-time constraints are not satisfied. Such advancements in hardware design and modeling have also increased the complexity in the software generation. The current trend in embedded system design is beginning to emphasize on simplifying the software generation process while maintaining the design quality, leading to recent advancements in mixed hardware-software modeling platforms. These mixed hardware-software modeling platforms are driven from a software-oriented design methodology and provide a common language (e.g. C/C++) for designing and modeling the hardware and software. It allows system designers, hardware designers and application developers to work concurrently on the same design source before partitioning is performed. This enables early verification of the interaction between the hardware and software components in the entire system, allowing better system trade-off decisions to be made and more importantly reducing system integration issues. The use of such modeling platforms as an integrated hardware-software co-design/simulation environment for hybrid embedded systems must be explored in detail and consequently the synthesis approach for hardware realization that can be employed in the framework must also be investigated.

1.2 OBJECTIVES

The main objectives of this project are summarized as follows:

- i) To develop a system-level simulation framework using a mixed hardware-software modeling platform to encapsulate the hardware and software aspects of a FPGA-based reconfigurable co-processor.
- ii) To facilitate design space exploration of various hardware and software models in a reconfigurable microarchitecture using our framework.
- iii) To conduct experiments on some commonly used application tasks and analyze the simulation results in terms of the resource utilization, reconfiguration patterns and the performance for different co-processor models
- iv) To demonstrate an economical approach combining commercial back-end tools with our framework for performing hardware synthesis and implementation on the target devices.

It is expected that our study, by way of simulation will advocate the use of software-based modeling platforms to enable system-level hardware-software co-simulation in embedded system design predominately for FPGA-based reconfigurable devices.

1.3 MAJOR CONTRIBUTIONS

We have developed a system-level simulation framework using a software-based modeling platform to design and model a reconfigurable architecture that can adapt and accelerate commonly used application tasks. This integrated framework facilitates exploration of the microarchitecture and allows refinements to the design in consideration of the hardware and software characteristics during the initial design phase. It also analyzes the interface performance and the reconfiguration demands so that the host requirements can then be suggested. This work has been published in *LNCS vol. 3740*.

Based on the framework above, it is extended to model an FPGA-based hybrid embedded system that consists of a general purpose CPU integrated with a reconfigurable co-processor. We identify two types of soft-core CPU interfaces to create tightly-coupled and loosely-coupled extension models. The framework will provide an integrated hardware-software co-simulation effort where various hardware-accelerated solutions and software-only solution can be explored efficiently. We have also identified an economical approach where the models can be synthesized and implemented using commercial back-end tools. This work has been published in *FPL '06*.

Published

K.S. Tham and D.L. Maskell, “**Software-oriented System-level simulation for Design Space Exploration of Reconfigurable Architectures**”. *Lecture Notes in Computer Science vol. 3740, Springer-Verlag, pp. 391-404*, October 2005.

K.S. Tham and D.L. Maskell, “**Software-oriented approach to Hardware-Software Co-simulation for FPGA-based RISC extensible processor**”. *16th International Conference on Field Programmable Logic and Applications (FPL)*, Madrid, August 2006.

1.4 OUTLINE OF THE THESIS

The research work conducted for this project is described in this thesis and has been organized into 6 chapters. Chapter 1 presents an overview of the research work. It also highlights the objectives of the research and indicates the major contributions of the thesis.

Chapter 2 reviews the current research in reconfigurable architectures for embedded systems and the dynamic reconfiguration characteristics inherent in these microarchitectures. It focuses on the key properties that enable a dynamic run-time system, the various description languages that are used to design these microarchitectures, current FPGA place-and-route tools and co-simulation environments for modeling the hardware and software components of a reconfigurable embedded system.

Chapter 3 outlines the design of loosely-coupled and tightly-coupled reconfigurable co-processor models using the SystemC framework. The individual components used in the co-processor models and their implementation on the respective FPGA targets are also described in this chapter.

Chapter 4 introduces the design of the simulation and compilation framework. It defines each of the components in the simulation framework that enables hardware-software co-simulation. This chapter also highlights an economical approach using commercial back-end tools for synthesis and implementation purposes.

Chapter 5 describes the application examples that are used to conduct the experiments on the simulation framework. It also presents the experimental results and discusses the related findings.

Chapter 6 concludes the work done in this project and proposes recommendations for future work.

CHAPTER 2

LITERATURE REVIEW

Reconfigurable computing systems have been widely seen as a compromise to bridge the gap between hardware-based and software-based implementations. These systems have shown great potential in accelerating computational intensive tasks and when implemented on FPGA can provide significant speedup over general-purpose processors as well as increased flexibility compared to their ASIC counterparts.

Figure 2-1 A Run-time System Overview.

Figure 2-1 shows a run-time system that represents an overview of a typical reconfigurable system. The system can consist of one or more of these software modules with their relationship between one another in the design tool chain and implementation stage. At the top of the design tool chain, application kernels (i.e. from software profiling) are partitioned into hardware and software specifications. The partitioning algorithm is dependent on the underlying microarchitecture and may target one or more levels of parallelism. Hardware and software optimization is performed independently in the respective compilation stages and may use multiple sources that are developed over

several platforms. This limits the framework (i.e. design tool chain) to refine the hardware and software design concurrently. Different reconfigurable systems may adopt various approaches in their framework to achieve an appropriate hardware-software co-design strategy. Some may focus on simplifying the hardware compilation and synthesis process so that the system can be reconfigured at run-time. Others may use a software-based approach directed by compiler optimization to exploit different levels of parallelism and there are even frameworks which adopt mixed hardware and software optimization. It can be observed that the relationships between the design tool chain and implementation are closely dependent on one another. An approach adopted in the tool chain will have to be supported by the low-level modules.

In this chapter we will survey the different types of reconfigurable architectures for embedded applications and the dynamic reconfiguration characteristics that are inherent in these architectures. We also present various description languages that are used to design these micro-architectures, FPGA place-and-route tools and co-simulation environments for modeling the hardware and software components of a reconfigurable embedded system.

2.1 RECONFIGURABLE ARCHITECTURES

2.1.1 FPGA for Custom Computing Machines (FCCM)

FCCM is a class of reconfigurable system that is normally associated with the combination of a CPU and a standard FPGA. In general, the reconfigurable hardware logic functions like a co-processor to accelerate computational intensive tasks and can be tightly coupled to the internal datapath of the CPU or loosely coupled to the CPU through a memory-mapped interface or a peripheral bus. The implementation in [Dyer] uses the partial reconfiguration capability of the FPGA device to achieve dynamic reconfiguration. This implies that an active area can be reconfigured while ensuring the rest of the circuits in the device will continue to function correctly. The co-processor forms a dynamic core that can be ‘swapped’ with another type of computational core within a designated area on the device. Although this fabric provides the flexibility to adapt to different types of tasks, it only supports a temporal partition where a single task can exist and execute at a time. The configuration overhead can also be significant

especially when ‘swapping’ large cores, but this is better than changing the entire device configuration as only a partial area needs to be reconfigured.

Other types of systems have a greater demand for dynamic reconfiguration. The hardware resources in [Wie] are constantly ‘tuned’ to the system behaviour. Multiple hardware patterns co-exist in the same space and can be temporal as well. However, this system requires an external host and communication interface to partition and reconfigure the hardware. For embedded portable devices, it presents a less attractive option than a self-contained System-on-Chip (SoC) implementation that can meet the demand for portability.

The use of a soft-core processor in [Dyer] can also provide the flexibility to explore the different types of coupling interface between the CPU and the co-processor that cannot be achieved with hard-core CPUs (e.g. PowerPC). This coupling interface can be a conventional ‘glue’ logic type, a memory-mapped interface, a peripheral bus type or tightly-coupled to the internal datapath of a CPU. MicroBlaze [*uBlaze*] is an example of a commercially viable soft-core processor that provides a fast dedicated communication channel (i.e. Fast Simplex Link – FSL) to connect to a co-processor core. In [Rosinger], this FIFO-based FSL interface is used to integrate custom IP modules with the CPU and is ideal for streaming applications. A large number of FSL channels ensure that computational intensive tasks can be easily parallelized in the hardware and controlled by the software executed on the MicroBlaze processor. Soft-core processor-based platforms can provide fast prototyping with system implementation being relatively easy because of good support from commercial FPGA vendor tools (e.g Xilinx EDK and ISE) and Real-time Operating Systems (RTOS) (e.g. MicroC/OS II).

FPGA with a hard-core processor also presents an alternative solution. In [Wolinski], the fabric consists of a mesh network with reconfigurable cells. Each cell has a local controller to decode and execute instructions stored in the memory. The fabric receives signals from the embedded host to start or stop execution either in a Single Instruction Multiple Data (SIMD), Multiple Instruction Multiple Data (MIMD) or systolic method. Since custom instructions cannot be built with existing hard-core processors, the host has to broadcast instructions over a bus to enable execution or store them in the local program memory of each cell where they can then be executed concurrently.

We can see that the hardware compilation and software compilation of these systems requires a tightly coupled co-design effort. The architectural framework must partition a system into hardware and software components where the design for a computational problem must be automatically synthesized to enable partial reconfiguration of the target device. The software component will control the execution and reconfiguration of the co-processor that can be self-contained in a single chip or as an external host.

2.1.2 Coarse-grained Reconfigurable Architectures

Coarse-grained designs exploit parallelism at a higher-level than the fine-grained cell structure of the underlying device technology (e.g. FPGA). This also applies to the dynamic reconfiguration support. These coarse-grained microarchitectures normally consist of a static pattern to exploit a certain execution model (e.g. SIMD, systolic, etc). To extend their usage, a 'stored-program' concept is adopted where the functionality of the hardware resources and the interconnection between the resources can be programmed using software control. This software control requires memory storage to hold the current context (i.e. functionality and routing) of the hardware resources while allowing new configuration data to be loaded. We can see that the dynamic reconfiguration support is different from systems that exploit partial reconfiguration of an FPGA device. This approach has a simpler reconfiguration mechanism since it can be programmed through software and is generally faster than changing the device configuration. However, the flexibility is restricted by the degree of programmability (i.e. functionality and routing of the resources) that is pre-designed into the microarchitecture during compile-time and relies heavily on software compiler techniques to discover parallelism for a computational problem in hardware.

Garp [Hauser] presents one of the earlier works on combining a general-purpose CPU with a reconfigurable co-processor. The co-processor is composed of an array of entities that resembles the Configurable Logic Block (CLB) of the Xilinx 4000 series FPGA device. Although the reconfigurable part in this architecture is fairly similar to those used in FCCM, the configuration pattern of the co-processor is still programmable by software and does not involve synthesizing a hardware design for partial reconfiguration on a FPGA device. Dynamic reconfiguration can be achieved by storing the recently used

configurations in caches where the main processor can rapidly load and execute between several configurations without the need to access the memory constantly. However, the Garp architecture requires a considerable amount of design effort to align and fit a computational problem into an array of fine-grained entities.

MorphoSys [Singh] has a reconfigurable array of cells arranged in a 2-dimension mesh structure. The operation code for the Arithmetic Logic Unit (ALU) and multiplexer are provided through a context register that is broadcast to all the cells. The fabric can be setup so that each row or column of cells can perform a single operation on a set of data (SIMD). The context register gives a convenient way of providing configuration to the cells compared to program memory. However, the amount of memory that can be dedicated to these registers is limited. The embedded host would also need to provide a wide data bus (128-bit) for only rows or columns and not individual cells. The downside of this system is that a fine-grained application would not map well into such structure. A combination of fine and coarse-grained blocks will further extend the flexibility.

Matrix [Mirsky] has a similar arrangement of cells as MorphoSys. The architecture consists of an array of identical 8-bit datapath elements connected together in a configurable network. The configurable network can be reconfigured to support 8, 16, 24 or 32-bit wide datapaths and can handle multiple instruction streams either in SIMD, Multiple-SIMD or Very Long Instruction Word (VLIW) manner.

The Imagine Stream architecture [Kapasi] has a series of ALU clusters that are coupled to the microcontroller datapath. The data stream is transferred to these clusters through a stream register file synchronized by the controller. Each cluster has a number of functional units connected to a local register file, scratch-pad memory, a communication unit to the neighbouring cluster and a crossbar switch to route data between the functional units. These clusters can support a sequence of VLIW instructions and if the instructions are similar in all clusters, SIMD control can be achieved. This architecture has found usefulness in applications that have producer-consumer locality of streams and especially in accelerating graphics algorithms (e.g. polygon rendering).

2.1.3 Software Configurable Architectures

Software configurable architectures also employ a tightly coupled programmable logic in the internal datapath of a general-purpose processor and use custom instructions to control the hardware fabric. These architectures can provide great design flexibility to add new features for emerging trends in embedded applications. Depending on the type of implementation, platforms that adopt this architecture can offer a range from performance-based to low-cost solutions.

The Stretch S5 [Arnold] family of software configurable processors provides a complete SoC solution for embedded applications. The S5 architecture has an Instruction Set Extension (ISE) fabric integrated into the internal datapath of a Reduced Instruction Set Computer (RISC) processor. Data can be fed into the ISE fabric through a 32-entry register file (128-bit wide) that is coupled with a 128-bit wide access to the memory. Similar to the Stream architecture, the ISE fabric contains a plane of arithmetic units that are interlinked by a programmable routing fabric. The arithmetic units can be configured to support longer word-sized operations using custom instructions. These custom instructions can either be specified by the application developer or generated by the compiler. To extend the functionality, the S5 architecture also provides external co-processor support using a bus mapped interface.

Nios-II is a soft-core FPGA-based processor that uses a similar approach of integrating Custom Instruction Logic (CIL) [NiosCIL] into the datapath of the processor. The main difference between the Nios CIL and the Stretch ISE fabric is that an application developer can design the structure of the arithmetic units and synthesize the design for hardware implementation. The CIL provides a defined protocol to the Instruction Set Architecture (ISA) of the processor for building custom instructions to control the hardware in the CIL. However, the scope of this Nios-CIL solution is quite limited due to the small resource area that is dedicated for the custom logic. Moreover, there are only 2-input and 1-output registers available to stream data to the CIL. Even with these shortcomings, it can still efficiently handle Instruction Level Parallelism (ILP) that is mainly found in applications with recursive loop operations.

2.1.4 Polymorphous Computing Architectures (PCA)

PCA is a relatively new area of research aimed at providing innovative features such as adaptable execution models and distributed resources that will increase performance and flexibility. Re-targetability is also intended with emphasis on enhancing simulation and compiler support to enable exploration of different architectures.

The most recent to emerge is the Monarch [*Monarch*] architecture that combines a processor-in-memory with fine-grained elements. In threaded mode, the fabric can be formed into slices with a coarse-grained processor and an independent path to the on-chip RAM. Two slices can be combined to support longer word-size operations. In streaming mode, clusters of ALU and memory can be formed to map fine-grained elements in a dataflow graph. Generally, the Monarch architecture can cover a wide variety of high performance embedded applications.

The RAW [*Taylor*] architecture presents a coarser-grained approach. The composition of an array of MIPS R2000 processor tiles and programmable switches makes it highly regular and easy to replicate, but an expensive solution. However, since it consists of hardware structures of a standard processor, compilation for the RAW architecture is fast and instruction sequencing is flexible compared to FPGA-based systems. Moreover, exploiting parallelism is at a coarser-level and low-level parallelism may have to be partitioned across several tiles. A stronger support for software compilation optimization is required.

M3T [*Renau*] which is similar to RAW is also composed of RISC cores interleaved with memory blocks. To support the three architectural traits (VLIW, TaskScalar and MIMD), each core can execute its own stream of instructions and synchronize between cores, execute parallel code independently in each core, or dismantle an instruction into fine-grained tasks and assign them to the various cores.

As PCAs continue to evolve, it is imperative to construct retargetable software tools that can explore code generation and optimization of these architectures. The individual components in the microarchitecture must be parameterized in the modeling framework

so that these components can be reused to support rapid prototyping and exploration of scalable architectures.

2.2 DYNAMIC RUN-TIME SYSTEMS

A dynamic run-time system needs to manage the hardware-software compilation, task scheduling and reconfiguration activities for an underlying reconfigurable architecture that can be based on a fine-grained or coarse-grained structure. The framework in these run-time systems normally adopts a layered structure that presents a high-level interface to the application developer for performing the various activities. These activities can be viewed as roles of the Operating System (OS) since they are mostly implemented as software services in an OS. Resource partitioning has to be defined first before the compilation or scheduling methods can be implemented. This section first describes the different types of resource models that can be implemented in the reconfigurable hardware. We will also survey a few of these run-time systems in terms of the hardware-software compilation and scheduling methods adopted by the various frameworks.

2.2.1 Resource Model

A key difference between FCCM and coarse-grained reconfigurable architectures is in the setup of the hardware resource models. FCCM has a dynamic hardware resource utilization compared to coarse-grained types where the resource model is static.

We will explore various area models that have a dynamic usage as these models are closely related to the routing and placement at run-time. Two types of models shown in Figure 2-2 were discussed in [Walder]: 1D and 2D area models. The 1D model resembles that of the current Virtex II FPGA technology. Xilinx has documented two flows for partial reconfiguration [Xapp290] where the fabric is vertically partitioned as fixed width columns. Tasks will have to fit into these spanning columns. It can be clearly seen that potential interference can occur when there is an attempt to allocate other tasks to the same column. Internal fragmentation may result with these fixed partitions, with resulting area wastage when tasks are smaller than the column. However, it does offer simpler routing and placement as the columns (blocks) are at a fixed position. Static and dynamic modules can be partitioned clearly and the re-routing of modules at run-time is more ordered.

Figure 2-2 Resource models: (a) 2D model. (b) 2D fixed block partition. (c) 1D variable block partition. (d) 1D fixed block partition.

The 2D variable block partition model gives more flexibility to which the area can be used over the 2D fixed block partition which resembles the Xilinx Virtex 4 FPGA architecture. The disadvantage is the complexity of the scheduling and resource management. Re-routing to the external devices imposed a difficult proposition as re-route and timings have to be re-analyzed. In [Dyer], a virtual socket is proposed to provide a template for creating paths for the re-routing of wires. The resource model proposed in [Walder] allows different sized blocks to achieve a better match for the tasks. This decreases fragmentation within the blocks and yet simplifies scheduling and placement.

Obviously, the goal is to place tasks so that the resource area can be optimized and also have partitions that are flexible to allow adaptation to different sized tasks.

2.2.2 Hardware-Software Compilation

The design of a computing system is generally subjected to the hardware-software partitioning problem that is based on design tradeoff decisions such as performance, power and area. The objective of a partitioning algorithm is to translate the requirements of a system functional behaviour into hardware and software designs, and to ensure the

correctness of the system behaviour. The current partitioning approach often uses a high-level abstraction layer of the hardware that is visible to the software model. The use of a Hardware Description Language (HDL), e.g. VHDL, Verilog, etc, is a good example of providing that hardware abstraction layer. This approach requires partitioning to occur at an early stage of the design flow where the hardware and software partitions are developed independently. Integrating the hardware and software at the later stage presents critical issues if the performance of the hardware and software do not match the system requirements. It is imperative that refinements to the system can be performed concurrently to both the hardware and software designs at different levels of the design flow.

▪ HDL-based design

Hardware compilation in the context of a run-time system must handle the issue of transforming the HDL of the designs into hardware circuits that can be manipulated by the framework during run-time. A simple approach is adopted in [Dyer] where different co-processor circuits are designed and partitioned at compile-time, and can be ‘swapped’ within a designated area of the FPGA device during run-time. Co-processor cores can be selected based on high-level requirements matching without having to perform any tedious low-level translation.

In [Burns], the run-time framework builds hardware circuits from pre-defined structures (i.e. Lookup tables (LUT), multiplexer for routing, etc.) of the underlying FPGA device. These generated circuits are normally of a finer granularity than the circuits used in [Dyer] since they exploit the fine-grained cell structure of the device. The framework also uses partial evaluation which is a methodology to make a certain portion of the circuits for the current design static so that it can be reused for the next circuit compilation. This requires the framework to provide services that can transform (i.e. rotation, scaling, etc) a circuit to fit into a designated area and must monitor the resource utilization at run-time. Resource management and transformation services must also be available for software compilation to perform scheduling of the hardware tasks in the resource. However, since this platform is implemented with an external host, computing power is not a critical issue as compared to those platforms that employ self-reconfiguration on an embedded CPU.

There are other types of hardware compilation methods that can translate a hardware description automatically into synthesized circuits. A complete flow from HDL to bitstream generation is introduced in [Lava]. This approach uses a HDL to describe the hardware circuits in terms of the CLBs and multiplexers of an FPGA device and is fairly similar to [Burns]. The major difference is that this framework uses the partial reconfiguration of an FPGA device. The circuit description is first compiled and parsed as EDIF, and then generated as a bitstream using JBits [Guccione] to achieve the mapping, and place and route tasks during run-time. The processing time from the parse EDIF phase to the partial configuration generation is more than 90 seconds using a Pentium-based computer. Compared to the time to reconfigure an entire FPGA device, this setup latency presents a challenge to overcome especially if targeted at an embedded CPU. However, it presents an interesting mechanism for a client-server architecture where partial evaluation can be located on the server and actual reconfiguration at the client.

Advancements in the hardware design and modeling have allowed the above frameworks to simplify the hardware compilation process. However, this has also increased the complexity in the software generation. We can see that the software development and compilation in HDL-based design is not automatically generated and does not occur until the hardware is designed and verified. It is also possible to use a VHDL/Verilog to C compiler [Tenison] to generate the C program if a HDL is used to specify and model the system. However, it requires a significant amount of analysis to compile a HDL program. The C program is also not targeted to any ISA and assumes a static schedule. It needs to be re-compiled or re-written for the target RTOS. Since the hardware-software partitioning occurs at an early stage, the software development cycle has to cope with any mismatch between the hardware specifications and the system requirements. It is also difficult for HDL-based modeling to capture the polymorphism of the reconfigurable architecture (e.g. PCA, etc) and requires novel scheduling algorithms to manage the hardware configurations. Moreover, these frameworks are not retargetable and use multiple sources to describe the hardware and software which makes co-design/simulation very complicated.

▪ Software-based design

To facilitate efficient co-design/simulation, there is an emergence of mixed hardware-software modeling platforms driven from a software-oriented approach. A C-based design platform like Open SystemC Initiative [OSCI] consists of software defined class libraries for system behavioural and register-transfer-level (RTL) designs. It is based on standard ANSI C/C⁺⁺, with the simulation kernel providing hardware clocked modeling for the behavioural and RTL designs. A system designer can begin with an un-timed SystemC model of the overall system and subsequently the same model can be used for hardware-software partitioning and compilation. In the next section, we will describe a framework that adopts this type of software-based design environment to enable hardware-software co-simulation.

Software generation in these C-based designs employs a preprocessing stage to extract the kernel loops from the system specifications. The IMPACT compiler [Chang] accepts the C program of a computational intensive algorithm and extracts a set of the most frequently executed innermost loops. An intermediate representation of the compiler, Lcode (a meta-assembly language) is generated. Dataflow analysis is performed on the detected kernel loop to derive the relevant information (e.g. dependencies between the loop iteration, register live range, etc) for mapping the kernel into the hardware datapath. The datapath can then be extracted by matching each software instruction with a corresponding execution of a functional unit in the hardware. This software design methodology presents a more dynamic approach since different algorithm tasks can be profiled to determine if it can be efficiently optimized in the hardware.

Another programming paradigm, Molen [Vassiliadis], is targeted for FCCMs. It uses a Control Flow Graph (CFG) to include sequences of instructions that control the reconfigurable hardware. The code generated from this programming model consists of instructions to pass input parameters, execute the hardware, reconfigure the hardware and return the results. Though it produces a simple scheduling for the reconfigurable hardware, the drawback is in the degradation of performance when large reconfiguration sequences are required. Improvements must be made to consider the placement constraints in the reconfigurable hardware.

Compiler directed frameworks [*Palem*][*Fan*] can automatically synthesize high-level specifications for the customized microarchitecture and increase the capabilities of embedded systems to meet the demands of complex algorithms. These frameworks emphasized the simplification of the synthesis problem and mainly exploit ILP of the underlying VLIW architecture or parallel custom logic. While these frameworks provide a convenient approach using a single source specification to translate to RTL, there is no support for co-simulation and refinement efforts at the system design level, and the compiler optimization only targets a particular level of parallelism.

▪ Architectural Description Language (ADL)-based design

An ADL is a language that can be used to describe the microarchitecture in terms of the behavioural information (i.e. mainly the ISA of the microarchitecture), the structural information (i.e. the microarchitecture structure that consists of the hardware resources), and the linkage information that joins the behavioural and structural information. Currently, no ADL has been standardized and several types of ADLs [*Bashford*], [*Halambi*], [*Gyllenhaal*], [*Pees*], exist which are basically grouped into behavioural based (i.e. for instruction set simulators), structural based (i.e. for synthesis) and mixed type (i.e. for generating micro-architecture simulators). The architectural information defined by an ADL is used to generate retargetable compilers and simulators and is seen as a promising approach to model PCA [*Hardnett*].

A compilation process can also be formulated for frameworks that are based on an ADL approach. The platform in [*Li*] uses an ADL to define the components (e.g. processor and reconfigurable array, memory hierarchy, etc.) and parameters of the system. The C-based application code is first profiled into kernels as CFGs using the architectural information defined by the ADL. The CFGs contain the software execution time and execution frequencies of each kernel. The CFGs are then partitioned into hardware kernels as data CFGs using the same ADL descriptions to include hardware delay and area details. However, the hardware-software partitioning algorithm only targets loops and basic block levels to exploit ILP. The advantage of this platform is that the ADL supports the flexibility of adding new components or changing the parameters of individual components without re-defining the entire system design.

2.2.3 Scheduling Methods

Task scheduling is similar to instruction scheduling in a single microprocessor in that they are both subjected to a resource or dependence constraint. The simplest method is to place the task immediately upon request. However, short tasks with little or no waiting time (slack) will get rejected compared to a long task with slack and so some priority must be given to the task performance.

A structure that uses a number of queues to interface arriving tasks to the block's partition was defined in [Walder]. The number of queues and position depends on the device's block partitioning. Tasks on the queue are placed accordingly and new tasks are placed at the head of the queue. The placer located on the host CPU can either allocate a task to any block that can accommodate it or restrict a task to a specific block only. Non-preemptive scheduling is more straightforward as the current task in a resource does not need to be preempted. Some simple scheme can be implemented, such as:

- First Come First Served (FCFS) – the queues are sorted in FIFO order.
- Shortest Job First (SJF) – queues are sorted with respect to task execution time.

As Run-time Reconfiguration (RTR) systems have greater demands over the usage of resources across a time-span, the layout of the fabric needs to be disturbed quite frequently. It is stated that each policy may be subjected to certain constraints. For example, a FCFS scheduler may schedule a later arriving smaller task before an earlier arriving longer task.

Compaction can be used to minimize fragmentation. It involves rearranging a group of tasks (currently being executed or reserved) to accumulate contiguous space for the new task. The use of this compaction technique on a partially reconfigurable FPGA was investigated [Diessel]. Tasks slide from left to right along the rows of the FPGA cells in a 1-way, 1-dimension manner. Using a directed graph of active tasks, a number of possible candidate spaces can be selected for the new task. The space with the minimum cost associated with freeing other active tasks will be chosen. This compaction technique did not consider task deadlines which would place a time constraint on the schedule. However, a table can be constructed based on the time available for each task, the slack associated with each task, the deadline of the task and the service time. Tasks can then be

prioritized and sorted based on this information. Most of these scheduling methods assume unrelated or independent tasks and the performance figures may become unrealistic in practice. It can be seen that preemptive and non-preemptive tasks have different scheduling needs. There is no generic scheduling method that works for all types of tasks in an application. To be practical, these scheduling methods have to be subjected to some form of heuristics and can only apply to a range of applications.

The framework presented in [Mooney] allows the ability to evaluate the performance of different hardware-software partitions with an automatically generated run-time scheduler for a robotics system. System modeling is performed using a collection of tasks that are specified in C or Verilog and also includes the details of the constraints (e.g. relative timing, resource, etc.) of each task. A control/data flow graph can then be constructed from each invoked software or hardware tasks and can be used to produce a schedule based on the worst case execution time for the application. However, the scheduling algorithm is based on heuristics to improve execution time by allowing tasks to be suspended or ‘swapped’ out of context and this may not be the exact solution. In general, the framework enables a convenient approach for designing a run-time scheduler at system-level and also provides a method to synthesize the scheduler in hardware and software respectively.

A dynamic approach to manage the ‘swapping’ of hardware cores in an FPGA device can be seen in [Panainte]. It uses a graph to capture the data flow of the hardware configurations for the various tasks. The graph can then be used to produce an instruction schedule for the software to control the configuration sequences. This schedule is also subjected to FPGA-area placement constraints that measure the degree of conflict in terms of area placement when replacing one task with another. This metric can be used to decide if the task is to be executed on the hardware or as software using a general-purpose CPU.

2.3 FPGA PLACE-AND-ROUTE TOOLS

The advent of RTR has placed new constraints on the place-and-route methodology as compared to conventional methods. With RTR, these constraints are magnified as routing and placement rules have to be fast and accurate to support the run-time

flexibility of these systems. Thus, simplifying placement and routing at run-time remains a key focus. Conventional place-route methods do not provide a flexible and lightweight approach for the embedded CPU to perform self-reconfiguration of the hardware resources.

2.3.1 Limitation of conventional place-and-route tools

EDA companies have invested heavily in providing efficient FPGA design tools to deal with the complexity of these devices. Designers use EDA tools with a design flow that may consist of:

- Design using HDL
- Functional design simulation
- Netlist generation
- Map, place and route netlist design (vendor specific)
- Timing and functional constraint verification

After the verification process, an executable bitstream is produced that can be loaded into the device. The entire flow can be completed in a matter of days and this gives FPGA design the edge in terms of time to market over their ASIC counterpart. FPGA designs are modular in approach, where sub-modules can be designed and validated separately. The challenge of any vendor place-and-route strategy is to place and make the connections in the smallest area that satisfies the performance and technology constraints. Thus, module placement has a significant impact on the routing performance. A metric to quantify this challenge is the possible cost involved in the place-and-route algorithm employed. This must also take into consideration the area and timing constraints to minimize latency. Of course, the correctness of the algorithm, where modules can be physically placed and easily routed with no cell overlap within the boundaries of the chip, has to be honoured.

Finding an optimum layout in a limited time is imperative yet difficult to achieve as traditional place-and-route is time consuming. This is especially so with large designs. In the context of Compile-time Reconfiguration (CTR), this is less obvious as the device configuration remains throughout the application life-time. Generally, modification to the design in an application is limited to a small active area but if required, the whole

design has to be recompiled and reprocessed starting from the top of the design hierarchy. Unfortunately, this will also involve re-placement and re-routing of the entire design, and is therefore undesirable.

Although EDA companies have tried to accelerate the processing time through high-speed compilation and a more efficient development cycle by integrating the HDL design flow into the initial stage of the development cycle, the design flow is still non-incremental and requires the transversal of the entire flow no matter how trivial the changes to the design may be.

There are incremental compilation types that only re-synthesize, re-place and re-route the changes while reusing the unchanged portion. Yet, more is required to allow direct manipulation of the bitstream, rather than being based on standard FPGA synthesis, so that configuration can be modified on demand and fast access to the device made possible.

2.3.2 JBits and Xilinx Partial Reconfiguration Toolkit (XPART)

JBits [Guccione] is a set of Java classes that provide Application Programmable Interface (API) access to generate and read the bitstream from a Xilinx FPGA or the Xilinx design tools. Designers can use JBits to produce static digital circuits, or design or modify a bitstream generated from conventional FPGA tools. As Java has fast compilation times, and programming control is at the CLB level, the bitstream can be readily modified. To aid designers in large complicated designs, JBits 2.5 allows the building of user reusable cores as well as using RTPCores supplied by FPGA vendors. With these parameterizable cores provided as an abstract class, placing cores in a specific location is achieved by simply manipulating the vertical and horizontal coordinates through the JBits API functions.

Routing in Xilinx FPGAs can be provided by the JRoute tool [Keller]. This interface allows the user to define ports for automatic routing. It also provides several levels of control. These include: routing a bus connection; routing a single source to several sinks; etc. JBits and JRoute are only built for the Xilinx FPGA architecture (JRoute is currently only available for the Virtex architecture) and are not targeted for other FPGA

architectures; however, they do provide a mechanism for allowing an original design to be modified, re-placed and re-routed using their API.

The use of a standard FPGA design flow integrated with JBits for hybrid systems was demonstrated in [Dyer]. To facilitate ‘swapping’ of dynamic cores at run-time, initial full configuration bitstreams are generated using the standard FPGA design flow. Using JBits, a partial bitstream which represents the pattern differences between two dynamic cores can be extracted from the full configuration bitstreams. The direct manipulation of the partial bitstream is rather simple and is implemented by modifying only the contents of the Lookup Tables (LUTs) and the BlockRAMs. While the locations of the reconfigurable cores and the interface between the static and dynamic cores are fixed, constraints to the design must be defined in order to place and re-route these modules dynamically. This can be achieved by defining local routing, where connections are made in the same core, or by identifying disturbing lines that cross the coprocessor area and using special hard macros to complete the routing for routes that do not guarantee a sufficient safety margin. The above approach does not take into account the requirement that multiple tasks can coexist and function at the same time and space. As each task may have a different requirement in terms of power and bandwidth, place-and-route on demand will need to abstract this statistical data so that run-time management can schedule resources to satisfy task performance.

Unfortunately, the JBits API does not provide a lightweight approach for enabling self-reconfiguration of the FPGA resources on an embedded CPU. To achieve this, XPART provides a minimal set of JBits API features for on the fly resource modification. A self-reconfiguring platform that employs XPART is presented in [Blodget]. XPART is build on top of the Xilinx Virtex II and Virtex II Pro internal reconfiguration access port (ICAP) interface. The ICAP interface provides the hardware dependent features to modify the configuration caches and active configuration memory of the FPGA device. By providing a high-level interface to the hardware dependent layer, XPART becomes hardware independent and can be used on any embedded CPU that is targeted for the Virtex-II family of FPGA devices.

2.3.3 Versatile Place and Route (VPR)

As JBits supports only Xilinx FPGA devices, VPR [Betz] presents a flexible tool capable of targeting a broad range of FPGA architectures. It has the provision to use different technology-mapped netlists and can add new routing architectures by using the architecture description file to specify the architectural parameters. It can also accept placement generated from another Computer Aided Design (CAD) tool to perform either global or combined global/detailed routing.

The placement algorithm of VPR can allow different CPU time / placement quality tradeoffs. By reducing the number of moves to a factor of 10% per temperature in the annealing schedule which VPR placement employs, a speed up of 10% in placement can be achieved with a reduction in placement quality of only about 10%. The main advantage of VPR is that its flexibility allows vendor independent FPGA architectural studies. It can also provide statistics such as routed wirelength and track count, etc, that may be used for power or performance estimation.

As VPR is only intended to examine place-and-route techniques for FPGA research, it requires modification for it to work with commercial FPGA architectures. For example in [Xu], VPR has been modified for Xilinx Virtex FPGAs by integrating it with the JBits interface. VPR is used to generate a netlist, placement and routing file for the Virtex architecture and these files are provided to JBits to produce the bitstream configuration.

2.4 SIMULATION ENVIRONMENTS

Simulators are important tools for exploring new microarchitectures especially in the area of reconfigurable computing systems. Conventional simulation environments normally target a specific microarchitecture design and the code used for simulating the system behaviour must be re-defined with the functional and timing information so that hardware synthesis for the target device can be achieved. However, this makes cycle-accurate system verification very difficult to accomplish using only abstract behavioural descriptions. As the design of reconfiguration architectures for embedded applications requires a tightly coupled co-design/simulation of the hardware and software components, it is imperative for emerging simulation environments to adopt a single source approach for all the design information. The environment must also be

retargetable so that code re-use is feasible and can be rapidly modified to evaluate design changes. In this section, we describe a number of retargetable simulators that are based on a software modeling, hardware modeling or mixed hardware-software modeling approach.

▪ **Software-based modeling**

SimpleScalar [Austin] is based on a software modeling approach to simulate a superscalar, out-of-order issue CPU. It provides a parameterizable simulation tool suite and supports the Portable Instruction Set Architecture (PISA) which is a derivative of the MIPS instruction set. It uses a C-based language to model the behaviour of the components (e.g. instruction decode, cache, memory, branch prediction, etc) in the microarchitecture and the communication between the components is accomplished through procedure calls and message passing. Although SimpleScalar provides a simple mechanism to change component parameters so that different system configurations can be explored, adding non-standard processor components into the framework requires a considerable amount of design effort. Moreover, adding custom instructions into the instruction set requires the re-compilation of the entire environment which is not a trivial process. In spite of this, many co-simulation platforms use SimpleScalar to provide an instruction accurate model of the CPU and the subsequent behavioural model can be verified by integrating it with a logic simulator to provide a time accurate model.

The Trimaran [Tri] simulator is intended for exploring new ILP architectures. It is targeted for a virtual HPL-PD processor which is configurable by a machine description, MDES [Gyllenhaal]. The compilation for the HPL-PD architecture is provided by IMPACT, a front-end compiler and Elcor, an intermediate representation to the executable code. This simulator is able to support limited changes to the machine configurations of the processor architecture, e.g. configure number of load/store units, to optimize the ILP architecture. It also provides a C⁺⁺ interface to external monitoring tools for performance analysis.

▪ **Hardware-based modeling**

HDL-based simulators (e.g. ModelSim, Synopsys, etc) are widely used in simulating logic circuits that can be directly synthesized for hardware. The hardware designs can be described at various abstraction levels; behavioural, functional and Register-Transfer-

Level (RTL). The main feature of the HDL-based framework is the ability to simulate hardware modules concurrently unlike a sequential model used in the software modeling approach. HDL-based design also follows a hierarchy structure which allows the modules to be reused across the entire design. Simulating at the RTL level can provide very accurate details, but this is very time consuming and the low-level descriptions are not easy to modify. Platforms that use a HDL-based approach do not normally include software details (i.e. instruction set) as it cannot efficiently capture this information. Partitioning between hardware and software must be performed early at the system-level and this means integration issues are more likely to surface at a later stage.

- **Mixed hardware-software modeling**

SystemC [Grötter1], SpecC [STOC] and Handel-C [CEX] are examples of mixed hardware-software modeling languages that use software class libraries to efficiently describe hardware circuits and perform clocked hardware evaluation. SystemC models hardware at a higher abstraction level than VHDL and Verilog, and can produce various models that are either abstract and un-timed or cycle/time-accurate and pin-accurate. One key feature of SystemC is that a single source can be used for all design information. This allows hardware and software components to be verified concurrently and any refinements to either component can be performed rapidly. Currently, the SystemC simulation kernel does not contain necessary specifics to facilitate an ISA model. However, it will continue to evolve to support a complete software generation methodology on an RTOS [Grötter2]. It is also possible to include a CPU model in a SystemC-based simulation environment that can emulate a RISC instruction set.

2.5 HARDWARE-SOFTWARE CO-SIMULATION PLATFORMS

Enabling hardware-software co-simulation is a powerful concept whereby the hardware and software requirements can be modeled concurrently in an integrated design environment. We can see that the simulation environments described in the previous section need to be extended or integrated together to provide an efficient co-simulation framework. A single source approach presents a more attractive option to implement a co-simulation platform. However, other platforms exist that can also achieve co-simulation by combining hardware-based modeling with software-based modeling.

▪ SystemC-VHDL-LISA

The co-simulation framework in [Hoffman] introduces a two-step approach for hardware modeling where SystemC is used in the beginning at the system-level specification stage. The behavioural C⁺⁺ models are only in abstract form (i.e. abstract data type and interface between modules), and cannot be used directly for hardware synthesis since they do not provide cycle/time accurate details. The behavioural models require a manual VHDL translation in the next step that can provide the functional and timing details for system verification using an external logic simulator. The software portion of the C⁺⁺ models can also be integrated into a software-based CPU simulator called LISA [Pees] through the compiled simulator's API that targets the ARM 7 instruction set. Although this framework provides virtual system prototyping that has several abstraction layers e.g. a system designer can work on the behavioural models ignoring low-level details, it requires a considerable amount of integration effort using multiple simulation environments. Moreover, this framework is not flexible enough to support design space exploration for reconfigurable architectures

▪ SimpleScalar-VHDL

The work in [Carrillo] extends the SimpleScalar simulator to model the reconfigurable units that are integrated into the datapath of a RISC superscalar processor. The reconfigurable unit is based on the Dynamically Programmable Gate Array in [DeHon] and can store multiple contexts of the array. However, the reconfigurable unit is implicitly modeled and cannot provide cycle accurate details. This type of framework is suitable for exploring software configurable architectures, but only an instruction accurate model can be obtained which does not represent the actual implementation.

Similar to the above work, a two-source framework is presented in [Enzler]. The framework also uses the SimpleScalar simulator to model the reconfigurable unit of multi-context hybrid architecture. The SimpleScalar CPU model has been extended to include a bus interface for the reconfigurable unit where custom instructions can be added based on this interface. The reconfigurable unit is explicitly described in VHDL which can be verified using a logic simulator and synthesized to the target hardware. To enable co-simulation, the SimpleScalar simulator is integrated into a VHDL simulator, ModelSim, using the vendor specified foreign language interface. However, the interface between the two simulators is of an abstract type using message passing to communicate

and does not give a realistic indication of the actual implementation. Moreover, a considerable amount of synchronization is required between the message passing models.

▪ **Mixed hardware-software co-simulation**

A co-simulation framework for multi-processor architectures using the SystemC simulation kernel is introduced in [Benini]. This framework combines an Instruction Set Simulator (ISS) with SystemC and uses an interprocess communication and bus wrapper to communicate between the two simulators. SystemC provides the simulation back-end where hardware descriptions (using SystemC) and software descriptions (using ISS) can be connected together. It also provides an interface module to translate the ISS models into cycle-accurate models. The main feature of this framework is that it can support various co-simulation interfaces using remote ISS co-simulation, linked ISS co-simulation, virtual in-circuit emulation and instantiating bus wrappers as SystemC modules. However, this framework assumes that hardware-software partitioning is already pre-determined and the use of the bus wrapper and interprocess communication do not give any realistic representation of the actual implementation.

Commercial types of design platforms like Celoxica DK Design Suite [CEX] provide C-based design (Handel-C) and synthesis tools that can accelerate the co-design/verification process of C-to-RTL and direct C-to-FPGA synthesis. This integrated tool suite provides a complete design methodology where a system-level model can be specified using Handel-C and directly partitioned for hardware and software. The subsequent hardware-software partitions can be co-verified in the RTL simulator and ISS simulator and then synthesized for the target device and RTOS respectively. DK Design Suite also provides a co-simulation bridge for modules that are described in SystemC. It uses a wrapper interface to translate a SystemC module into Handel-C form. Celoxica also offers an off-the-shelf compiler (Agility Compiler) to automatically synthesize the SystemC design. The generated netlist can be imported into the DK Design Suite as EDIF format. Although this design environment provides a fast simulation and hardware-software co-verification, it does not readily facilitate design space exploration since the software ISS and compilation only targets a specific embedded CPU. The software co-verification part does not provide an API for a compiled or interpretive CPU simulator. Moreover, the hardware synthesis is automatically optimized for the fine-

grained structure of the underlying target device and makes it difficult to include coarse-grained microarchitecture details.

Impulse CoDeveloper [*ImpC*] is a more recent commercial C-based design environment (Impulse C), which unlike Handel-C, is based on standard ANSI C. The main distinction between Impulse C and Handel-C is that the former does not provide a direct control over parallelism where the application developer can explicitly specify the clock timing and identify which portion of the C code to parallelize (using '*par*' and '*seq*' statements) in hardware. Instead, the Impulse C compiler will extract the kernels from sequential C code and automatically provide the hardware compilation for the parallel behaviour. Thus, the hardware-software partitioning and compilation are totally transparent to the application developer. The limitation of this platform to facilitate design space exploration is similar to Celoxica DK Design Suite. Moreover, the current Impulse version does not provide co-simulation capabilities with SystemC.

SystemCrafter SC [*SysCr*] is based on the OSCI framework and provides a synthesis tool for SystemC designs. Although SystemCrafter SC can provide system simulation, gate-level simulation and VHDL generation, it lacks co-simulation features. The advantage of this framework is that the generated RTL VHDL can be used by commercial back-end tools (only for Xilinx FPGA device) for synthesis and implementation on the target device. It presents a more economical approach and allows the use of hardware tools and an existing C⁺⁺ development environment that the designer is familiar with. Furthermore, coarse-grained microarchitecture details can be efficiently modeled and re-targeted for various FPGA vendor place and route tools.

2.6 SUMMARY

We have discussed the existing landscape of dynamically reconfigurable systems by examining the respective elements in their framework. From our review above, we can summarize the system characteristics in the followings.

- **Microarchitecture;** the underlying microarchitecture which can consists of fine or coarse-grained structures determines the level of parallelism that can be exploited. The modeling and compilation process will use the architectural information (including dynamic properties) to discover or retarget the optimization in hardware.

- **Design/modeling and compilation;** a hardware-based approach translates specifications to circuits and can simplify the hardware compilation but this also increased the complexity of software generation. Software-based approaches provide a flexible methodology to discover parallelism at a higher-level but neglects low-level details (i.e. synthesis) to enhance the hardware design. With a mixed hardware-software approach, the same specification source can be used to rapidly synthesize the hardware and software components.
- **Synthesis and simulation;** this depends on the framework approach. With hardware-based modeling/design, simulation is conducted at the synthesis level and the software is simulated separately without any hardware information. Software-based methodologies focus on validating system behaviour and use cycle-accurate information to simulate the hardware while mixed hardware-software design utilizes the integrated structure to perform co-simulation.

We can observe the strengths of a mixed hardware-software approach and especially their importance in FCCM where multiple designs at different granularities can be implemented. To facilitate design space exploration, existing landscapes must also emphasize on co-simulation and refinement efforts at the system design level by using explicit information in their simulation models to verify the system. This enables hardware and software design to be refined concurrently based on the top-level (system behaviour) and low-level (synthesis) results. Performing such early verification at the system-level can potentially reduce integration issues during implementation. It is these basic ideas that have influenced our design. The following chapters describe the proposed framework in more detail.

CHAPTER 3

IMPLEMENTING FPGA-BASED RECONFIGURABLE CO-PROCESSORS IN SYSTEMC

This chapter outlines the design of two reconfigurable co-processor models using the SystemC framework. We describe the individual components used in the co-processor models that enable hardware-software co-simulation and their implementation on the respective FPGA targets. These two co-processor models can then be analyzed for a range of kernel tasks in terms of the hardware-accelerated and software-only execution in our simulation framework.

3.1 SYSTEM ARCHITECTURAL MODEL

Figure 3-1 FPGA-based architectural model.

A basic FPGA-based architectural model is shown in Figure 3-1 where the embedded CPU and co-processor reside on the same chip. It can be seen that this architectural model has close resemblance to the FCCM and software configurable architecture types described in Chapter 2. The embedded CPU executes load/store operations and also controls the reconfiguration of the co-processor. The reconfigurable co-processor may contain a controller which provides a state machine to control the execution and the dynamic reconfiguration of the datapath. This controller provides a synchronization

method when the custom instructions (i.e. involving the co-processor) running on the embedded CPU require multiple clock cycles to complete. We have created two architectural models from the basic type which will form the basis for our simulation framework. The two models can be categorized by the type of coupling interface between the CPU and the reconfigurable co-processor.

Figure 3-2 Loosely-coupled extension model.

In the first model, an extension is defined as a loosely-coupled reconfigurable co-processor for an embedded RISC CPU. Figure 3-2 shows our target system architecture of the loosely-coupled extension model. This extension model is based on the Xilinx Fast Simplex Link (FSL) interface [xilinxFSL]. The FSL interface is a 32-bit wide FIFO-based communication interface with a maximum speed of 600 MHz during standalone operation. A total of 16 FSL channels are available where up to 8 channels each can be used for writing data from the CPU to the extension (slave operation) and reading from the extension to the CPU (master operation) respectively. The reconfigurable extension is first initialized by the extension control unit. It also provides ‘start’ and ‘stop’ signals to control multi-cycle execution and the dynamic reconfiguration of the reconfigurable logic. The switch network supplies the operands and reconfiguration data to the extension. It contains a set of multiplexers that allows the network to be reprogrammed so that the FSL channels can be re-assigned dynamically. The reconfigurable logic works as the computational core of the co-processor. Depending on the target application, the core can adopt various execution models (e.g. SIMD, MIMD, systolic, etc) and with different granularities (e.g. fine-grained or coarse-grained).

Figure 3-3 Tightly-coupled extension model.

Figure 3-3 shows a block diagram of the second target architectural model. This tightly-coupled extension model is based on the Nios-II processor CIL [*NiosCIL*] interface. The custom logic blocks are adjacent to the Arithmetic Logic Unit (ALU) in the CPU's datapath. This presents an Instruction Set Extension (ISE) of the CPU, where different custom functions can be designed and synthesized for a target application. The custom functions must conform to the CIL architectural specifications which can be a combinatorial, multi-cycle, extended, internal register file or external interface. Each of the functions implemented on the custom logic must share the available 2-input and 1-output 32-bit wide registers. Internal registers can also be built in the custom logic and selected by the custom instruction to drive the output register appropriately. Despite the small logic area compared to the reconfigurable logic in the loosely-coupled model and the limited amount of I/O registers to stream data, the custom logic can still efficiently handle ILP in recursive loop operations. In the next section, we will describe in detail the components in each of the extension models.

3.2 LOOSELY-COUPLED EXTENSION MODEL

There are three main components in our loosely-coupled extension model. The RISC CPU, the reconfigurable co-processor and the FIFO-based datapath interface. The following section describes the functionality of each component and the interaction between them using the SystemC framework.

3.2.1 SystemC RISC CPU and datapath interface to Co-processor

The objective of our CPU model is to provide a basic ISS that closely emulates the ISA of the target embedded CPU. Our CPU model does not consist of a formal compilation process where C code can be automatically compiled into object code for the target, nor does it exploit compiler optimization techniques like loop unrolling, etc, although these techniques could be in the future added to our model. Instead, due to the time limitations associated with this project, we focus on using simple assembly language code that is not subjected to any compiler optimization or scheduling algorithm. In this way we are able to facilitate a co-simulation environment so that the performance of executing a particular kernel task in software (i.e. CPU) or in hardware (i.e. CPU with co-processor support) can be analyzed concurrently.

Using a similar approach to that used in the MicroBlaze soft-core processor to connect custom IP through the FSL channel [Rosinger], we have modified a basic RISC CPU example from OSCI [OSCI] to include the FSL interface features. Our 32-bit RISC CPU model which is similar to the MicroBlaze [uBlaze] processor is a 3-stage pipeline architecture and consists of instruction fetch, decode and execute stage. Pipeline stalling of the pipeline is supported and data from the execute stage is forwarded back for storing into the register file. Instruction pre-fetching, branch prediction and speculation are not included in our CPU model.

Figure 3-4 shows an example of the CPU’s pipeline architecture for various types of instruction that may require multiple clock cycles to execute. To prevent conflicts in the functional units between instructions, the pipeline may need to be stalled, which will increase the latency if the preceding instruction takes more than one clock cycle to complete its execution.

Figure 3-4 Examples of the pipeline architecture.

Generally, the CPU scheduling algorithm will handle these conflicts and stall the pipeline appropriately. Since the FSL interface is integrated into the decode stage of the processor through the 32-bit register file [Rosinger], the performance of the CPU is not directly affected by the longer execution time of the co-processor for a computational intensive task. The FSL instruction can be executed in the hardware concurrently with the preceding instruction as the critical path of the entire system is independent of the co-processor execution compared to the case when it is attached as an internal datapath element.

Figure 3-5 Block diagram of the FSL interface to the internal CPU datapath.

A detailed connection between the internal pipeline components and the FSL block is shown in Figure 3-5. Instructions are fetched by the fetch unit and dispatched to the decode unit which provides the operation code (opcode) and the source and destination addresses of the fetched instruction. The opcode can be an arithmetic operation in the execute unit or an FSL operation to write or read data to/from the co-processor. The MicroBlaze CPU instructions that are supported by our model are shown in Appendix A1. Our model has been modified to give the exact instruction latency to that of the MicroBlaze ISA.


```

Decode IDU("Decode Unit"); //instantiate decode unit

    IDU << clk << reset << instruction << instruction_valid <<
    program_counter << alu_op << alu_src << src_A << src_B <<
    forward_A << forward_B << du_valid << fsl_valid << fsl_m_id <<
    fsl_m_data << fsl_m_control << fsl_m_write << fsl_m_full <<
    fsl_s_id << fsl_s_data << fsl_s_control << fsl_s_read <<
    fsl_s_exists;

FSL    FSLU("FSL unit"); //instantiate FSL unit

    FSLU << clk << fsl_valid << fsl_m_id <<fsl_m_data <<
    fsl_m_control << fsl_m_write << fsl_m_full << fsl_s_id <<
    fsl_s_data << fsl_s_control << fsl_s_read << fsl_s_exits;

```

Figure 3-6 Top-level interface description.

Our focus is on the FSL interface where a set of master and slave signals are used to implement the protocol. Figure 3-6 shows the top-level interface connection between the decode unit and the FSL unit using SystemC. We use two vector signals '*fsl_m_id*' and '*fsl_s_id*' instead to identify which master or slave channel to write to or read from. The rest of the FSL signals are as described in [xilinxFSL] and [uBlaze]. Basically, the '*fsl_x_control*', '*fsl_x_read*', '*fsl_x_write*', '*fsl_s_exists*' and '*fsl_m_full*' signals control the sequences of events for the write and read operations. The '*fsl_x_data*' carries a 32-bit data input to the master or slave interface of the FSL bus. An additional signal '*fsl_valid*' is included in our model to indicate if the data sent or received is valid. A '0' indicates that the data is valid, otherwise the data is invalid due to unavailable free space in the buffer during writing or reading. This signal is similar to setting the carry bit '*c*' in the FSL instruction set [uBlaze] when the buffer is full or the data does not exists. The master and slave interface can also be driven from different clock ports. For our model, the same clock is used for both the master and slave interface which is derived from the CPU clock.

We develop the SystemC model for the decode and the FSL unit in accordance to the FSL read and write bus operations which are illustrated in Appendix A3. Figure 3-7 shows an excerpt code listing of the decode unit.

```
switch(opcode) {

    /** arithmetic functions **/
    case 0x01:    //add R1, R2, R3
        src_A.write(srcA_tmp);
        src_B.write(srcB_tmp);
        alu.src.write(regC_tmp);
        alu_op.write(3);
        du_valid.write(true);
        wait();
        du_valid.write(false);
        wait();
        break;

    /** FSL functions **/
    case 0x6C:    //GET R1, FSL1 (blocking)
        fsl_s_id.write(fsl_tmp);    //which FSL channel

        do {
            wait();
            while(fsl_s_exists.read() == false);
            fsl_s_read.write(true);
            wait();

            //get FSL -> R?
            cpu_reg[regC_tmp] = fsl_s_data.read();
            fsl_valid.write(true);    //read data valid
            wait();
            fsl_s_read.write(false);
            wait();
            break;

        case 0x6E:    //PUT R1, FSL1
            do {    //wait if no space available
                wait();
                while(fsl_m_full.read() == true);

                //space in buffer available
                fsl_m_data.write(srcA_tmp); //put Rx -> FSL
                fsl_m_id.write(fsl_tmp);    //which FSL channel
                fsl_m_write.write(true);    //write valid
                wait();
                fsl_m_write.write(false);
                fsl_valid(true);            //data written is valid
                wait();

            break;
```

Figure 3-7 Excerpt code listing of the decode unit.

```

case 0x6D:    //NGET R1, FSL1 (non-blocking)
    fsl_s_id.write(fsl_tmp);    //which FSL channel

    //get FSL -> Rx
    fsl_s_read.write(true);
    wait();
    cpu_reg[regC_tmp] = fsl_s_data.read();

    //FSL valid if fsl_s_exists ->1, otherwise invalid
    fsl_valid.write(fsl_s_exists.read());
    wait();
    fsl_s_read.write(false);
    wait();
    break;

case 0x6F:    //NPUT R1, FSL1
    fsl_m_data.write(srcA_tmp); //put Rx -> FSL
    fsl_m_id.write(fsl_tmp);    //which FSL channel
    fsl_m_write.write(true);    //write valid
    wait();
    fsl_m_write.write(false);
    wait();

    /* if buffer full, no stall, fsl_m_full return 1-
       data invalid. Otherwise 0 - data valid
    */
    fsl_valid.write (fsl_m_full.read());
    break;

```

Figure 3-7 Excerpt code listing of the decode unit (cont'd).

Figure 3-7 shows four FSL instructions supported in our model that are similar to the MicroBlaze instruction set [*uBlaze*]. The breakdown of the 32-bit instruction set for the FSL functions is:

- Opcode – specified by the first 6 least significant bits, and bits 16 and 17.
- ‘PUT’ and ‘NPUT’ functions – bits 11-15 of the instruction specify the source register where the content is stored in ‘srcA_tmp’. For ‘GET’ and ‘NGET’ functions, bits 6-10 are used instead to select the source register.
- FSL channel – last 3 bits of the instruction are specified by ‘fsl_tmp’.

The FSL unit writes the data received from the master interface port into the FIFO buffer and indicates to the slave that the data exists. The data in the buffer is read when the slave activates the read signal at the rising edge of the clock. When the FIFO buffer is full, the FSL unit will indicate to the decode unit which blocking FSL instructions will stall the pipeline. Figure 3-8 shows an excerpt of the code listing for the FSL unit.


```

struct fsl_int : sc_module {
    sc_in_clk    clk;

    enum e {max = 3};
    int buf[max], ptr;

    SC_CTOR(fsl_int);
    {
        SC_CTHREAD(entry, clk.pos());
    }
    void entry();
};

void fsl_int::entry()
{
    while(1)
    {
        if(ptr == max) //fifo full
        {
            fsl_m_full.write(true);
            wait();
        }

        if(ptr !=0) //fifo not empty
        {
            /** FSL read **/
            if(fsl_s_read.read() == true)
            {
                fsl_s_data.write(buf[0]); //pop the first data
                fsl_m_full.write(false); //fifo have space now

                fsl_s_exists.write(false); //data is now invalid to slave
                                           //slave must deactivate read
                ptr--;                      //decrement count
                wait();

                for(i=0; i<=ptr; i++) //update the fifo queue
                    buf[i] = buf[i+1];
            }
        }

        /** FSL write **/
        if(fsl_m_write.read() == true)
        {
            if(ptr < max) //fifo not full
            {
                buf[ptr++] = fsl_m_data.read();
                fsl_s_exists.write(true); //data valid to slave
                wait();
            }
        }
    }
}

```

Figure 3-8 Excerpt code listing of the FSL unit.

3.2.2 Co-processor model

The co-processor model is split into two logic types; static and reconfigurable. The static logic consists of the extension control unit and the switch network that remains unchanged throughout the design. The reconfigurable logic contains the co-processor core as it can be reconfigured for different types of tasks.

▪ Reconfigurable logic

Currently, we have implemented 2 different types of co-processor core; a bit-serial and a multiply-accumulate (MAC) unit implementation of a 2-dimension processing element (PE) array. These two cores are coarse-grained, reconfigurable during run-time and can speedup kernel tasks using vector processing.

Figure 3-9 (a) Bit-serial PE block diagram. (b) MAC unit block diagram.

Figure 3-9 shows a block diagram of the bit-serial PE and the MAC unit implementation. Dynamic reconfiguration for both the implementations can be achieved using two context registers that are each 25-bits wide. When the array is active using the current context register, a new configuration can be loaded into the buffer register. The bit-serial PE and MAC unit uses a constant multiplication that is stored in the context register to accomplish a 16x16 multiplication. We can also reduce the reconfiguration cycles by

modifying the design so that the PE can rapidly update the multiplication field without changing the entire context register. The function of each PE is a multiply-add/sub operation (i.e. $A * K \pm B$) and produces a result sign extended to 32 bits. The multiply function can be disabled by multiplying by a constant '1'. Likewise, the add function can be bypassed by adding a constant '0'. This actually represents a delay unit. The multiplier architecture of the PE is based on the multiplier unit of the Systola 1024 parallel computer [Lang] which is a signed integer bit serial-parallel implementation (Refer to Appendix A4: Bit-serial PE). The significance of the bit-serial approach is the area advantage, but for an n -bit multiplier it requires $2n$ cycles to complete. The interconnection between the PEs is indicated in Figure 3-10. Each PE has the capability of connecting to 16 of its nearest neighbours and each connection is a single-bit channel. This reconfigurable feature allows support for computational problems that have non-regular data flow patterns.

Figure 3-10 PE interconnection pattern.

We have created an RTL model in SystemC for the above PE design and a 16x8 array structure. Figure 3-11 shows the hierarchy structure of the array. This hierarchy also shows how the components in the library can be reused. The library will contain basic components like full adder, register buffers and interface specifications. These components can be further specified in terms of their construction using common components such as gates (AND, OR, XOR, etc), flip-flops and latches etc. This makes the verification process easier as it can be performed at different levels of the hierarchy and isolated from the other levels.

Figure 3-11 Structural hierarchy of the PE array.

The reuse capability can be achieved by the attributes of module inheritance and polymorphism associated with C++ to create the regular structures. The array structure can be easily modified to any other type (e.g. linear array, tree, hypercube etc) by customizing the array interface description while the rest of the components can be reused. Figure 3-12 shows an excerpt code listing of the internal interconnection of a 16x8 array based on the PE interconnection pattern illustrated in Figure 3-10.

```

//16x8 array
PE *pe[16*8]; //instantiate 16*8 PEs

SC_CTOR(bs_array)
{
    cnt = 0;
    //connect all PE ports
    for(row=3; row<=(row_size+3-1); row++) {
        for(col=3; col<=(col_size+3-1); col++) {
            pe[cnt]->clk(clk);
            pe[cnt]->reset (reset);
            pe[cnt]->stall (stall);
            pe[cnt]->bus_config (bus_config[row-3]);
            pe[cnt]->config_row (config_row[row-3]);
            pe[cnt]->config_col (config_col[col-3]);
            pe[cnt]->wr_row (wr_row[row-3]);
            pe[cnt]->wr_col (wr_col[col-3]);
            pe[cnt]->bus_operand (bus_operand);
            pe[cnt]->rd_row (rd_row[row-3]);
            pe[cnt]->rd_col (rd_col[col-3]);
            pe[cnt]->bus_result (bus_result);

            pe[cnt]->node_n0(peConn[15*(row-1)+col]);
            pe[cnt]->node_n1(peConn[15*(row-2)+col]);
            pe[cnt]->node_n2(peConn[15*(row-3)+col]);
            pe[cnt]->node_nw(peConn[15*(row-1)+(col-1)]);
            pe[cnt]->node_ne(peConn[15*(row-1)+(col+1)]);
            pe[cnt]->node_w0(peConn[15*row+(col-1)]);
            pe[cnt]->node_w1(peConn[15*row+(col-2)]);
            pe[cnt]->node_w2(peConn[15*row+(col-3)]);
            pe[cnt]->node_e0(peConn[15*row+(col+1)]);
            pe[cnt]->node_e1(peConn[15*row+(col+2)]);
            pe[cnt]->node_e2(peConn[15*row+(col+3)]);
            pe[cnt]->node_s0(peConn[15*(row+1)+col]);
            pe[cnt]->node_s1(peConn[15*(row+2)+col]);
            pe[cnt]->node_s2(peConn[15*(row+3)+col]);
            pe[cnt]->node_sw(peConn[15*(row+1)+(col-1)]);
            pe[cnt]->node_se(peConn[15*(row+1)+(col+1)]);
            pe[cnt]->result(peConn[15*row+col]);

            cnt++;
        } //for(col);
    } //for(row);
}

```

Figure 3-12 Excerpt code listing of the bit-serial PE array interconnection.

The MAC-unit has a coarser-granularity than the bit-serial PE with a parallel multiplier and adder implementation. The MAC-unit has a two-stage pipeline using a register to hold the intermediate data so that the multiplication and addition can operate concurrently. The output data is transferred to any of its nearest 16 neighbours in a word length rather than a bit-serial format. This also applies to the transferring of the configuration data into the MAC-unit context registers. However, it is not possible to transfer the entire configuration data for a row of MAC-units (e.g. 8 MAC-units in a

row) since the FSL channels have limited bandwidth and so the FSL resources have to be shared. Using the same hierarchical design structure as for the bit-serial PE, we can also create an 8x8 MAC-unit array model for the co-processor core.

▪ Switch network

The switch network provides the multiplexing of the FSL channels to the array. Since 16 channels are available (8 channels each for input and output), the FSL channels can be allocated for the respective models as illustrated in Figure 3-13.

Figure 3-13 FSL resource allocation: (a) MAC-unit array (b) Bit-serial PE array.

The write channels (wFSL_x) are used to load operands and configuration data. All the PEs on the top row are connected directly to the FSL channel (to operand inputs in the routing logic) and operands are transferred in a row format. For the MAC-unit array, 2 PEs are allocated to 1 FSL channel as compared to 4 PEs per FSL channel for the bit-serial PE array. The rest of the PEs use the 16-bit operand bus (bus_operand) to load data into the register buffers using the same channel allocation. The operand loaded into the register buffer is then streamed to the next PE depending on the configuration of the interconnection pattern. Configuration data is transferred in a column format using 1 FSL channel for 4 PEs in one column. The rest of the write FSL channels are used to carrying control signals to the extension control unit. Results read from the array are also in a column format since parallelism occurs vertically. In the streaming format, results are produced at the rightmost column of the PEs. Each PE producing a 32-bit result is read

by 1 FSL channel (rFSLx) and a total of 8 read channels are used. In the accumulation format, individual column results can be read by selecting the appropriate column.

From the above, we can see that each type of array has a different switch network pattern. We can also build a programmable switch network to support dynamic “swapping” between both types of array. The addresses to the programmable switches can then be supplied by the FSL channel which takes the cue from the main processor. However, for our case, each switch network needs to be individually specified and it forms part of the overall co-processor core design which can be implemented onto the reconfigurable hardware.

▪ Extension control

Figure 3-14 Block diagram of the extension control unit.

The extension control unit controls the internal state machine of the co-processor cores. Figure 3-14 shows a detailed block diagram of the extension control unit. The communication between the extension control unit and the main processor is synchronized using the FSL communication protocol. The FSL unit on the co-processor is identical to the ones on the main processor. A communication path from the main processor to the co-processor will use a master-to-slave connection and vice versa for the reverse path.

The extension control unit manages the execution of the array using the control signals (e.g. stall, reset, etc.) that are carried on a dedicated FSL channel issued by the main

processor. It also loads operands and stores data to/from a specific element in the array. For example, the 4 bits in *'write_row'* can be used to select elements in any of the 16 columns to receive operands from the main processor. The subsequent operands will be forwarded to the designated FSL channel based on the resource allocation shown in Figure 3-13. The extension control unit also handles the reconfiguration of the context registers. It loads new configuration data into the buffer of the designated element (specified by *'config_row'*, *'config_col'* signals) when the array is active. When the array is next inactive, the new configuration data can then be loaded into the current context register.

The extension control unit for the MAC-unit array and bit-serial PE array are different. However, the FSL unit on the co-processor remains the same since both types conform to the FSL communication protocol. We can easily interchange these two cores in our framework to analyze the performance for a range of kernel tasks.

3.3 TIGHTLY-COUPLED EXTENSION MODEL

Our tightly-coupled model, shown in Figure 3-3, presents an instruction set extension where a RISC CPU is combined with the Nios-II custom instruction logic interface. We have extended the SystemC RISC CPU of the loosely-coupled model to include a 5-stage (fetch, decode, execute, memory, writeback) pipeline architecture with no branch prediction. This RISC CPU model resembles the Nios-II/s (small core size) core [Nios-II]. We have also implemented the Nios-II CPU ISA and it is detailed in Appendix A2. The execution performance breakdown for the group of Nios-II CPU instructions is also provided in Appendix A5 for the Nios-II/s core.

Figure 3-15 (a) Custom logic-ALU connection. (b) Extended multi-cycle custom instruction block diagram.

Figure 3-15(a) shows the ALU connected directly to the custom logic. Each component in the custom logic block operates independently and shares the same decoder unit in the CPU datapath. The custom instruction interface is more straightforward than the FSL interface as the behaviour of the custom instructions is similar to that of a conventional instruction. There are different types of custom instruction architecture [NiosCIL]. However, our custom instruction interface uses an extended multi-cycle model instead of a combinatorial one to examine the worst case where custom functions cannot complete in one clock cycle. This is illustrated in Figure 3-15(b). The ‘start’ and ‘done’ signals must be asserted by the custom instruction interface and the custom function respectively. Multiple custom functions can co-exist in the hardware logic but need to share the 2-input and 1-output registers. Our ISE supports the execution of one custom

function at any single time. If a multiple-input, multiple-output extension is available, we can easily extend our framework to include some control logic to support the execution of multiple custom functions.

Figure 3-15(b) shows three custom functions that are implemented in the custom logic for our experiment. All the custom functions have a two-stage pipeline which allows the multiplication and addition to operate concurrently. For the *custom0* and *custom2* functions, the multiplier will only accept two 16-bit operands supplied by one input register. Therefore, we must pack two 16-bits operands in one input register before presenting the data to the two functions. The *custom1* function can realize a 32x32-bit multiplication and accumulation using two input registers, however, only the lower 32-bit word of the result will be returned. As some Nios-II cores do not have an ALU that supports hardware multiplication, we can reuse the *custom2* function to perform a single multiply operation. In the *custom2* function, the 'readrb' signal controls the multiplexer to select between an internal register (hardwired to '0' – disable addition) and an input register. When the signal is low the internal register will be read instead. All the custom functions will run simultaneously but only the result from the desired function will be selected and subsequently returned to the CPU. The 'sel_result' multiplexer is used to select which custom function output will be valid on the result bus through the 'n[7..0]' field. This field is specified by the operation code of the function when the custom instruction is decoded. Each signal in our custom instruction interface model is derived from the opcode of the Nios-II processor custom instruction set. Our model must also conform to the multi-cycle custom instruction timing which is ensured by the appropriate 'done', 'start' and 'clk' signals. Refer to Appendix A5.

Figure 3-16 shows a code listing of the custom instruction interface. It shows the definition of the custom instruction interface and the method for the custom functions. The hardware descriptions for the function must be accompanied with this definition and can be easily included in our framework. The CPU (i.e. decoder unit) asserts the 'start' (active high) and 'n' signals and puts valid data onto the 'dataA' and 'dataB' signals. The 'n' signal is 8-bits wide and is used to select the desired result from multiple custom functions (max of 255 functions). The CPU waits until the 'done' signal is asserted (active high) before reading the result. The custom instruction interface will put the valid result on the bus and also assert the 'done' signal on the same clock cycle. The

implementations of the hardware multiplication and addition in the custom functions are similar to those employed in the loosely-coupled co-processor model.

```

struct custom_intf:sc_module {

    //port declaration
    sc_in<sc_logic> clk;
    sc_in<sc_logic> reset, start, readrb;
    sc_in<sc_lv<32> > dataA, dataB;
    sc_in<sc_lv<8> > n;
    sc_out<sc_logic> done;
    sc_out<sc_lv<32> > result;

    //signal declaration
    sc_signal <sc_logic> l_done;
    sc_signal <sc_lv<32> > l_result0, l_result1,
        l_result2;

    SC_CTOR(custom_intf) {

        /** instantiate custom functions */
        func0 << clk << reset << start << dataA << dataB <<
            l_done << l_result0;

        func1 << clk << reset << start << dataA << dataB <<
            l_done << l_result1;

        func2 << clk << reset << start << readrb <<
            dataA << dataB << l_done << l_result2;

        //reading custom function result
        SC_CTHREAD(read_fn);
        sensitive_pos << clk;

    }

};

```

Figure 3-16 A code listing of custom instruction interface.

```
void custom_intf::read_fn()
{
    if(l_done.read() == true)
    {
        //select desired result
        switch (n) {
            case 0x01: //return custom0 result
                result.write (l_result0);
                done.write(true);
                wait();
                done.write(false);
                wait();
                break;

            case 0x02: //return custom1 result
                result.write (l_result1);
                done.write(true);
                wait();
                done.write(false);
                wait();
                break;

            case 0x03: //return custom2 result
                result.write (l_result2);
                done.write(true);
                wait();
                done.write(false);
                wait();
                break;
        }
    }
}
```

Figure 3-16 A code listing of custom instruction interface (cont'd).

3.4 COMBINATION OF TIGHTLY-COUPLED AND LOOSELY-COUPLED EXTENSION MODEL

We can modify the MicroBlaze extension model to include the custom functions of the Nios-II ISE. Conversely, the Nios-II ISE can also be extended to include one of the MicroBlaze co-processor core (e.g. MAC-unit array).

▪ **MicroBlaze extension with custom functions**

To use the custom functions shown in Figure 3-15(b) directly, we can assign two FSL write channels for ‘dataA’ and ‘dataB’ bus that are each 32-bit wide and one FSL write channel for the control signals (i.e. ‘start’, ‘reset’, ‘readrb’ and ‘n’). Another FSL read channel is required to read back the result to the MicroBlaze CPU. The signals from the FSL units can be dispatched to the custom functions using an interface unit shown in Figure 3-17. If the FSL buffer for the read operation is full, the interface unit will stall and wait until the buffer is available. The operations on the FSL channels and the custom functions are identical to the loosely-coupled and tightly-coupled extension models respectively.

Figure 3-17 Block diagram of the MicroBlaze extension with custom functions.

▪ **Nios-II ISE with MAC-unit array**

This modification is more complicated as the data I/O resources of the ISE are limited and need to be heavily shared for the MAC-unit array. Figure 3-18 shows the block

diagram of the control unit to interface the ISE with the MAC-units and is modified from the one used in our loosely-coupled model.

Figure 3-18 Block diagram of the ISE with MAC-unit array.

The two available I/O inputs are used to provide either the configuration data or the data operands for the MAC-units and are controlled by the custom instructions described in Table 3-1. We can use five custom instructions to load or configure the MAC-units in the array. These instructions do not read back any results from the hardware array. For the *custom4* instruction, only one type of operation (i.e. configure or write) can be performed at a time and the data register for the disabled operation must be set to '0'. Results can be read back using the *custom5* instruction which will also send start, stop and reset signals to the hardware. If a start command is issued through the 'dataA' bus, the results will be read back from the MAC-unit selected by the data in 'dataB' bus and subsequently supplied to the array through the 'read_row' and 'read_col' signals. For the rest of the commands (i.e. stop and reset), 'dataB' bus is not use and should be set to '0'.

Custom instructions	Operation mode
Custom0 & custom1	Load top row of MAC-units. Each instruction will place two 32-bit data ('dataA' and 'dataB') onto four separate 16-bit buses and every 16-bit data bus will provide operands to one MAC-unit in the top row.
Custom2	Load the rest of the MAC-units. Four MAC-units can be loaded with operands simultaneously supplied by the 'bus_operand' buses.
Custom3	Reconfigure all MAC-units in column format. Every two units per column can be configured concurrently supplied by two sets of configuration data.
Custom4	Select MAC-units to configure and load operands. 'dataA' will provide the data for 'config_row' and 'config_col' signals to indicate which MAC-unit to configure. 'dataB' will provide the data for 'write_row' and 'write_col' signals to indicate which MAC-unit to write operands to.
Custom5	Start or stop execution, reset hardware and select MAC-units to read result from. 'dataA' supplies the commands for start, stop and reset signals. 'dataB' will provide the data for 'read_row' and 'read_col' signals to indicate which MAC-unit to read from.

Table 3-1 Custom instruction operation modes for ISE with MAC-unit array.

CHAPTER 4

DESIGN OF A CO-SIMULATION AND COMPILATION FRAMEWORK

This chapter outlines the design of the various components in the framework that enables hardware-software co-simulation of the co-processor models. We also describe an economical approach whereby the same source used for co-simulating the models can be translated for synthesis and implemented using commercial back-end design tools. The use of the terms compilation and synthesis in our context are interchangeable.

4.1 CO-SIMULATION FLOW

Many computing systems consist of an embedded CPU coupled together with a co-processor. In our embedded computing context, the co-processor can be viewed as the hardware component and the CPU as the software component (i.e. by way of it running the software part of the application). The hardware component represents the microarchitecture of the co-processor core and is reconfigurable by programming the functionality and the interconnection patterns of the hardware resources via software CPU. The CPU will also manage the execution of the resources and the interface to the hardware component using either an FSL interface or a custom instruction interface. It can be seen that the synthesis of such a system requires a tightly-coupled co-design/simulation methodology that can efficiently model the individual components. Instead of using multiple sources to perform hardware-software co-simulation (e.g. C-based ISS with VHDL), our main objective is to adopt a single-source (i.e. SystemC) approach throughout the hardware-software co-simulation and compilation process flow. Figure 4-1 shows the block diagram of our co-simulation and compilation framework.

Figure 4-1 Co-simulation and compilation framework.

The framework is modified from a generic SystemC system-level design process [Bhasker] for our target architectural models. The whole framework is divided into two major flows; hardware-software co-simulation flow and compilation flow. The compilation flow will generate the simulation executable based on the co-simulation flow and uses the same source for implementation purposes. We will describe this in the following section and reserve the rest of this section to describe the individual components that enable hardware-software co-simulation.

We begin with an un-timed functional description of the system-level model and add the timing details of the hardware or software progressively once the models are available. For our case, we will use a timed model at the beginning of the flow since we have already pre-designed our cycle-accurate models to explore the given computational problem with different co-processor microarchitecture cores. The input to this timed model is a C source code block of the target application. The co-simulation flow can be sub-divided into three domains; modeling, exploration and refinement domains.

▪ **Modeling domain**

The modeling domain deals with the different abstraction levels of modeling the hardware and software components in our target architectural models. Table 4-1 shows the characteristics of the various components in the loosely-coupled and tightly-coupled extension models. The advantage of using a SystemC platform is the ability to describe and simulate using either an abstract, a cycle-accurate or an RTL form. However, the complexity of the simulation increases with the amount of detail included in the design. To speed up simulation time, we do not simulate designs at the RTL level during system-level analysis. Instead, it is more appropriate from a cost-effective approach to use commercial back-end tools to validate the design. However, it is imperative that the components used at the system-level are cycle-accurate and conform to the communication protocol (i.e. functional and timing details) of the vendor specifications. This will provide a close representation of the actual hardware circuits.

Type	Models	Compute time	Communication protocol	Pin-accurate
Loosely coupled	RISC CPU (3-stage)	approximate	abstract and detailed bus channel	No
	FSL interface	cycle-accurate	detailed bus channel	Yes
	Reconfigurable extension: Bit-serial PE array MAC-unit array	cycle-accurate cycle-accurate	detailed bus channel detailed bus channel	Yes Yes
Tightly coupled	RISC CPU (5-stage)	approximate	abstract and detailed bus channel	No
	Custom instruction interface	cycle-accurate	detailed bus channel	Yes
	ISE	cycle-accurate	detailed bus channel	Yes

Table 4-1 Characteristics of the various components in the extension models

The RISC CPU models provide an ISS and differ from SimpleScalar in that they do not use message passing channels to communicate between the components (e.g. fetch, decoder, master-FSL unit, etc.) in the CPU. These message-passing types in SimpleScalar use software procedure calls with abstract interfaces to simulate the various CPU components. Our RISC CPU models employ clocked hardware modeling enabled by the SystemC simulation kernel to provide not only an instruction-accurate model but also the compute time of the software performance. The reconfigurable extension and the

ISE are cycle-accurate models with detailed bus channels. These models are also pin-accurate making them readily translatable for synthesis and implementation. We can vary the clock speed of each CPU model and the co-processor cores to compare the resulting performances. However, the upper limit depends on the maximum ratings of the vendor specifications and also the hardware synthesis results for the target FPGA device.

▪ **Exploration domain**

To aid system designers in making system trade-off decisions, we must provide an interface to present the characteristics of the modeling domain to the system designers. The exploration domain allows us to profile the target application and select a set of kernel candidates to implement in hardware. Since the components in the extension models are parameterized, we can interchange the hardware components (e.g. bit-serial PE or MAC-unit array) in each extension type or change the parameters of the components (e.g. clock speed, array size, interconnection pattern, FSL channel allocation, custom function, etc.) to meet the requirements of the kernel task. We can also use different extension types (i.e. loosely-coupled or tightly-coupled) to produce multiple hardware solutions or create software-only solutions with the RISC CPU models.

The ability to switch between the various extension models and the individual components creates multiple views/contexts of the mapping of the computational problem with respect to the selected hardware core. These views highlight the performance in terms of computation time, mapping space and routing space. To facilitate this switching between multiple views, we created a software sequencer program that runs on top of the SystemC simulation kernel and contains the interface information of all the hosts (RISC CPUs) and the hardware cores (co-processor cores). The sequencer and the multiple views are only in simulation space and not the physical hardware space, meaning that the views can be created at compile-time and used to explore possible hardware implementations. At run-time, the reconfiguration of the co-processor cores can be achieved using software control.

▪ **Refinement domain**

To cater for design space exploration, refinements to the individual components in the extension models can be made when exploring the different hardware solutions for the

given computational problem. This requires the simulation executable to be re-compiled as the co-processor cores are an integral part of the entire simulation environment. However, the re-compilation is quite straightforward as the compilation for the simulation executable is separated from the compilation for implementation. The use of the sequencer program to switch between the created multiple views of the hardware setup during compile-time can also eliminate the need to re-compile the simulation sources. The refinement domain also ensures that the parameter changes are verified at the system-level according to the correctness of the hardware solution and also to fulfill any real-time constraints.

4.2 COMPILATION FLOW

One of the key features of proprietary mixed hardware-software design platforms like Impulse CoDeveloper and Celoxica DK Design Suite is the simplicity of transforming a requirement described in a C-like language into a description that is ready for hardware implementation. The hardware-software partitioning and compilation of these platforms are transparent to the application developer and mostly target a specific microarchitecture. For an FPGA target, the descriptions generated by the non-customized compiler in these platforms will only exploit the fine-grained structure of the underlying device technology. In our case where coarse-grained architectures are used, we need a flexible framework to support the compilation of both fine-grained and coarse-grained extension models. From Figure 4-1, the compilation flow of our framework mainly consists of the compilation domain and the validation domain.

▪ **Compilation domain**

The compilation domain will generate two executables for simulation and implementation purposes. The key feature of this compilation domain is that the same source used for generating the simulation executable can also be translated for synthesis and implementation using commercial back-end tools. Our approach is to use existing implementation platforms where the synthesized netlist of our hardware designs can be imported like customized IPs and the corresponding software assembly code can be recompiled for the RTOS (e.g. uClinux RTOS for MicroBlaze) running on the target CPU. Essentially, the compilation domain consists of two parts; software compilation and hardware compilation.

▪ Validation domain

The validation domain closely relates to the hardware compilation. The validation domain ensures that the synthesis results (i.e. maximum clock performance and critical path analysis) from the compilation process for the target FPGA corresponds with the parameters (See Figure 4-1) used for co-simulating the models. Since we do not simulate the RTL design at system-level, we need to validate the synthesized RTL netlist using commercial back-end FPGA tools. The synthesis results must be fed back to our exploration domain to decide if the system requirements are satisfied. Any violation of the requirements will involve either making refinements to the co-processor models or fine-tuning the optimization level of the back-end synthesis tools. The former will require the re-compilation of the sources and the latter merely involve a re-synthesis of the design with the new optimization level. Either way, our framework facilitates a convenient mechanism for this validation-refinement process since the same source is used for both simulation and synthesis.

4.2.1 Software Compilation

The software compilation will partition the hardware-software relative to the FSL interface or the custom instruction interface. We define hardware specific instructions as the pre-defined FSL functions and custom functions that perform hardware execution on the respective co-processor resources. These instructions must be analyzed in the RISC processor ISA together with non-hardware specific ones as their timings can affect the outcome of the instruction scheduling on the CPU. The information can then be supplied to the compiler for determining the optimum instruction schedule.

```
void mm (int mm_1[row*col], int mm_2[row*col])
{
    for(i=0; i<4; i++)
        for(j=0; j<4; j++)
            for(k=0, mm[i][j]=0; k<4; k++)
                mm[i][j] += mm_1[i][k] * mm_2[k][j];
}
```

Figure 4-2 Example of a C code for 4 x 4 matrix multiplication.

To an application developer, the software compilation presents an explicit programming model. Currently, our compilation framework for both the simulation and implementation executable does not have a compiler that can automatically transform a C source code to the target hardware. An application developer will have to translate a C code shown in Figure 4-2 into an assembly code that corresponds to the instruction set of each CPU extension model (See Figure 4-3).

Figure 4-3 Pseudo assembly code: (a) MicroBlaze extension using FSL functions.
(b) Nios-II ISE using custom instructions.

Since the generated assembly code for the simulation phase is not subjected to any software compiler optimization, we need to analyze the dependencies between the hardware specific instructions. Figure 4-4 shows a typical example of the instruction schedule for the CPU model that resembles the MicroBlaze CPU. In this example, the co-processor waits for 4 sets of operands to be loaded from the CPU using the FSL interface before starting the hardware execution. The hardware execution at cycle 6 is the execution on the co-processor resource and not the functional units in the CPU datapath. Assuming the hardware execution time takes longer than the IF and ID stage and a read from hardware to the CPU is required immediately after execution; the earliest a hardware specific read instruction can be issued is after cycle 6. If the hardware execution is shorter than the ID stage time, then it is possible to pipeline the read instruction starting at cycle 5. If the former occurs, a compiler can discover these constraints and schedule two wait cycles for a conservative approach or issue independent instructions (non-dependent on the previous instructions) to keep the CPU datapath occupied. Such independent instructions can be reconfiguration sequences to configure the context registers when the hardware resources are active or even non-hardware specific instructions. Likewise for the multi-cycle custom instruction model, a

custom instruction that depends on the result of the preceding instruction must wait for its completion.

Figure 4-4 RISC CPU (3-stage) instruction schedule for hardware execution.

However, since we do not have a customized compiler at the present moment, we adopt the conservative approach by inserting ‘nop’ or wait instructions which represents the worst case scenario. The only optimization we implement manually in our software code is to minimize the amount of spilling in the registers so that memory accesses are kept to the minimum. This applies to all instructions supported by our RISC CPU models.

To implement the software onto the target CPU, the assembly code used for the simulation requires re-compilation using the software compilers in the appropriate commercial back-end design platforms (e.g. Impulse CoDeveloper, Celoxica DK Design Suite, Altera SOPC builder, etc.) and do not need to be re-written. The software compiler associated in these platforms can provide different levels of optimization which ranges from aggressive transformation on the execution ordering to optimization for size that leads to smaller memory footprint and fewer page faults. However, not all optimizations may produce an optimum schedule for the case described above. At least, through our co-simulation we have verified the functionality and performance of the non-optimized assembly codes using our RISC CPU models during the initial system-level design phase. Further optimization can be exploited using the back-end tools depending on any prerequisites that need to be satisfied.

4.2.2 Hardware Compilation

The same co-processor models (See Figure 4-1) used for co-simulation can be translated for synthesis and implementation on the target device. For the reconfigurable extension, we use SystemCrafter [SysCr] to generate a RTL VHDL description for the Xilinx FPGA device. Since SystemCrafter does not support Altera FPGA devices at the present moment, we have to manually convert our SystemC descriptions of the custom functions into its equivalent VHDL form. This process is relatively simple as SystemC and VHDL use a similar modular design approach. The RTL VHDL descriptions can then be synthesized to a netlist using Synplify Pro which is a high performance logic synthesis engine for the respective FPGA devices. From the synthesis results, we can derive the maximum clock performance and details of the critical path delay for each of the extension models. We use a clock speed of 50MHz for both the bit-serial and MAC-unit array, although 100MHz and 60MHz respectively, can be achieved using a Xilinx Virtex-II FPGA. For the multi-cycle custom instruction interface, the custom functions can operate at a maximum clock frequency of 100MHz on Altera Stratix-II FPGA. Currently, we only use the automatic optimization selection provided by the Synplify Pro framework to simplify the synthesis process. Other advanced optimization techniques [Synplify] are available such as retiming to “balance” the registers across a critical path, the ability to extract a finite state machine automatically from the code and apply the best encoding styles based on design constraints and area/delay requirements, pipelining, resource sharing to improve area at the expense of timing, etc. However, some of the optimization techniques (e.g. pipelining) cannot be applied automatically without satisfying several prerequisites in the design. A hardware designer can use these optimization options to fulfill any area/delay constraints or even refine the original design. The validation domain described above provides a flow to ensure that the refinements to the design and the re-synthesis process can be performed and validated easily. It also ensures that the synthesized results correspond to the parameters selected for our co-simulation models.

Our hardware compilation excludes the RISC CPU models, the FSL interface and the custom instruction interface as these models can be provided by the commercial back-end design platforms as synthesized cores. We can import the netlist of our co-processor cores like a custom IP into the Impulse CoDeveloper and Altera SOPC builder

framework and combine with the synthesized cores for the respective target device implementation.

In summary, we have introduced a flexible framework that provides an economical approach for enabling hardware-software co-simulation and implementation of the two CPU extension models. By using formal descriptions for co-simulation at the initial design phase we can achieve early system verification during high-level modeling using a single source throughout the co-simulation and implementation phase. We have also provided a flow to ensure that the refinement-validation process can be performed easily. System/hardware designers and application developers can then work on the same source concurrently, which has the potential of reducing system integration issues.

CHAPTER 5

APPLICATION EXAMPLES AND
EXPERIMENTAL RESULTS

5.1 APPLICATION EXAMPLES

In order to demonstrate our integrated framework, we experiment with three kernel (integer) tasks applied to the coarse-grained cores described in Section 3.2.2. They are the discrete cosine transform (DCT), matrix multiplication (MM) and finite impulse response (FIR) filtering as these are commonly found in mainly DSP and image processing applications.

▪ DCT

We use the MediaBench [Lee] JPEG and MPEG compression C program for an 8x8 DCT block on a 512x512 pixel image size. For the ISE, we have translated the C sources into an assembly code that corresponds to the Nios-II CPU instruction set and use the custom instructions to achieve speed up. We use a Nios-II core that consists of an ALU that does not have a hardware multiplier unit. We can replace the multiply instructions in the benchmark code with the equivalent *custom2* function shown below.

<code>z1 = MULTIPLY(z2, -FIX_7373);</code>	<code>custom 2, r1, r2, r2 r1 = (r2(hi) * r2(lo)) + 0 (r1 ->z1, r2(hi)->z2, r2(lo)-> constant)</code>
<code>z3 = MULTIPLY(z3, -FIX_16069); z3 += z5;</code>	<code>custom 2, r1, r2, r3 r1 = (r2(hi) * r2(lo)) + r3 (r1-> z3, r2(hi)-> z3, r2(lo)-> constant, r3-> z5)</code>

The first *custom2* instruction only performs a 16x16 integer multiplication. The multiplicand (z2) and the multiplier (constant) are loaded as the upper half-word (16-bits) and the lower half-word of register, r2, respectively. We can accomplish the alignment of the data in the register using the ‘*movhi*’ instruction to load the appropriate half-words. The addition stage in the *custom2* function is disabled by activating the ‘*readrb*’ signal to select adding ‘0’. In the second *custom2* instruction, we can use the pipeline stage between the multiplication and addition stage to provide

speedup. The multiplier operands are loaded into register, r2 and the add operands are supplied in register, r3. To enable the addition, the ‘readrb’ signal is deactivated. While the addition stage is computing the final result of the current multiplication, a new multiplication can start concurrently. We can substitute approximately 20% of the original benchmark code with the second type of *custom2* function and each substitution has a potential saving of two instruction cycles. For elaboration purpose, we have maintained the same instruction syntax for both modes of operation (i.e. by calling both operations, *custom2*). In practice, the encoding of the instruction for both of the modes is different which is specified by the ‘readrb’ bit (See Appendix A5).

For the MicroBlaze extension model, we will use a systolic mapping of a 2-D array of bit-serial PE and MAC-units to exploit parallelism. We will consider one of the possible systolic architectures for implementing the 2-D DCT algorithm [Lim]. Figure 5-1 shows a 1-D DCT mapping onto a 2-D mesh array. Each PE/MAC-unit performs a multiplication on the input operand with the constant multiplier stored in the context register. The multiplication result is added with the output from the preceding neighbour in a lateral format and the first set of values will emerged from the last column of the array at the end of the 3rd cycle. Thereafter, each set of results will be produced at every cycle.

Figure 5-1 A 1-D DCT mapping.

A 2-D DCT can be accomplished by performing a 1-D DCT on each column followed by another 1-D DCT on the row using vector processing [Pillai] given by $[Z] = [C][X][C]^T$

where Z is the output, X is the input, C is the cosine coefficients and C^T is the transposed coefficients. The intermediate value $[Y] = [X][C]^T$ (i.e. a 1D DCT operation) can be reordered and then streamed back to perform the final value, $[Z] = [C][Y]$. This reordering is performed on the CPU when the intermediate values are read from the hardware through the FSL read channels and then streamed back using the write channels. In this way, we do not require to reconfigure each PEs with $[C]$ and $[C]^T$ coefficients when computing the intermediate and final values. In addition, the hardware resource only needs to be initialized once at the beginning of the computation. The reconfiguration time can then be amortized in the overall execution time. To increase the performance further, we can also interleave the second intermediate computation shown in Figure 5-2 once the first column of PE/MAC-units has completed executing the first set of results.

Figure 5-2 Interleaving intermediate computed values.

We have replaced the benchmark code of the DCT algorithm with the resulting assembly code that contains the hardware specific instructions (FSL functions – ‘PUT’, ‘GET’, etc.) for the above operating descriptions and also in accordance to the FSL channel allocation for the bit-serial PE and MAC-unit array.

▪ MM

3D geometry computation [Watt] such as transformation, shading, etc can be achieved using matrix multiplication where a matrix is multiplied by a k -element vector. Using a systolic array of PE/MAC-units, we can achieve two different hardware maps. The first map is identical to the 1D DCT where we can perform the matrix multiplication. The operands are loaded into the array and results are streamed in a similar fashion to the

DCT mapping. The size of the vectors is also proportional to the array size. Therefore, we can extend the array easily to support larger vectors by adding PEs to the right of the array. The second mapping of the MM is shown in Figure 5-3 where a single element of the matrix is computed using 7 PEs. We can see that the mapping is quite straightforward involving 4 multiply and 3 add. The data-flow for this mapping will cause one of the PEs to be redundant for each computed element in the matrix. Thus, it can be seen that scaling the array to support wider vectors result in some wastage in the resources and is not easily adaptable compared to the first mapping. Furthermore, a larger resource usage will require longer reconfiguration cycles especially to reconfigure the constant multiplier in the context register.

Figure 5-3 Alternative 4x4 matrix multiplication mapping.

We can directly write the assembly code for both of the mappings as it is fairly straightforward. The matrix multiplication is repeated over 5000 cycles using the worst case where a different vector is used for every cycle. For the MicroBlaze extension model, we can reuse the same DCT code for the first MM mapping. The second mapping will require writing FSL instructions to customize the interconnection pattern of each PE to the mapping shown in Figure 5-3. The setup of the array will only need to be initialized once. Four FSL write instructions (*PUT*) are needed to load the first row of PEs/MAC-units that perform the multiplication. At the first cycle, the four 16x16 multiplications are executed concurrently. The result of each multiplication is summed by using two PE/MAC-unit adders at cycle 2. The final summation is performed at cycle 3 and one FSL read instruction (*GET*) is required to read back the result. In conjunction to the add operations at cycle 2, the row of PE/MAC-unit multipliers can execute the next set of operands. Although the mapping in Figure 5-3 uses a larger resource area, it can compute all the results in only 3 cycles if the entire MM can be mapped into the

array. Otherwise, the PEs will have to be reused. Even so, the computation time is faster than the first MM mapping.

For the ISE, we can generate the assembly code using the pipelined *custom0* function to achieve speedup as shown below.

<code>custom0, r1, r2, r3</code>	<code>r1 = r2(hi)*r2(lo) + r3(hi)*r3(lo)</code>
----------------------------------	---

The *custom0* instruction will multiply two half-words supplied in two registers. The results are stored in two internal 32-bit registers. While the two products are being added in the next cycle, the next pair of multiplications can be executed. The final addition can be performed by the ALU of the processor. In total, we require 2 custom instructions and 1 ALU add instruction to compute a single element in the matrix. This amount to about a two times savings in the total number of instructions used.

▪ **FIR**

```
int i, j, sum;
for(j=0; j<num_out; j++)
{
    sum = 0;
    for(i=0; i<num_coeff; i++)
        sum += x[i+j]*b[i];

    r[j]= sum >> 15;
}
```

Figure 5-4 A basic FIR filter procedure.

The basic C source code for the FIR filter, shown in Figure 5-4, can be directly translated to assembly code for the ISE. The main operation is the multiply and accumulate operation for which we can use the *custom1* function shown below. As the *custom1* instruction is a direct exchange of the C statement, there is no potential savings in the number of instructions used. However, since the *custom1* function is pipelined, a performance gain of more than two times can be achieved.

<code>Sum += x[i+j]*b[i];</code> <i>(where x-input samples</i> <i>b-coefficients)</i>	<code>custom 1, r1, r2, r3</code> <code>r1 = (r2 * r3) + ACC</code> <i>(r1 ->sum, r2->x, r2-> b, ACC->accumulation)</i>
---	---

Similar to matrix multiplication, we can also use a systolic array to implement the FIR filter mapping shown in Figure 5-5 for our MicroBlaze extension model. The mapping is a simple translation of the basic operations to perform the FIR filter. The input samples are shifted serially into the array and we can dedicate more rows of PE/MAC-units to execute in parallel for a larger number of input samples. There is a limit, restricted by the mesh array size (i.e. 8x8 for MAC-unit array and 16x8 for bit-serial PE array), in which case the PE/MAC-units will have to be reused.

Figure 5-5 A M-tap FIR mapping.

5.2 EXPERIMENTAL RESULTS

We can use our simulation framework to produce sets of experimental results to analyze in terms of the resource usage, reconfiguration patterns and the performance for the different extension models.

5.2.1 Mapping structure

We can analyze the different mapping structures that are used to implement a kernel task. Table 5-1 shows the comparison between the different mapping structures of the bit-serial PE array in terms of the number of PEs and the number of cycles used for computation, performing I/O and reconfiguration.

Operations	# PEs	# Cycles
4x4 Matrix multiplication mapping 1	Total: 20	Config: 520
	Compute: 16	Compute: 112
	I/O: 4	I/O: 44
4x4 Matrix multiplication mapping 2	Total: 144	Config: 3744
	Compute: 128	Compute: 48
	I/O: 16	I/O: 4
2D DCT (4x4 block)	Total: 20	Config: 520
	Compute: 16	Compute: 176
	I/O: 4	I/O: 76

Table 5-1 Comparison between number of PEs and cycles for computation, I/O and reconfiguration.

The comparison made here is to observe how the mapping space in regular structure design can be affected by the different implementations of the same computation problem and does not consider a host and its interface to the hardware array. As such, the number of cycles obtained from our simulation only indicates the hardware execution of the PE array and not the total clock cycles or the computation timings. From the results, we can begin to suggest improvements in the array or host interface parameters. The second mapping of the matrix multiplication shows that the configuration cycles are more than 7 times that of the first mapping which is proportional to the number of PEs used in each case. However, the compute cycle is more than 2 times faster, which is expected. To improve the second mapping we can either increase the configuration bandwidth or schedule configuration phases to inter-leave with other non-computational

phases. It can be noted that the configuration cycles mentioned above are the time it takes to reconfigure the PE's function and connection to take up a new shape in the hardware. The device configuration timings are not included.

Another point to observe is the I/O bandwidth. By allocating more PEs to service I/O operations does not necessary reduce the time it takes to load or read the PEs. The bottle neck is still on the host data bus bandwidth and the layout of these buses across the array.

5.2.2 Configuration patterns

From a system-level, a hardware design can be partitioned into regions, some where reconfiguration can occur and others that remain functionally static. The reconfigurable regions can be built from a hierarchy of medium-grained tasks or a compilation of fine-grained ones to form a medium-grained task. Building specialized circuits from a description at compile-time or run-time is a non-trivial process and is very time consuming for embedded platforms. This demanding process can be simplified using an approach to generate design by building from known hardware structures that have a well defined map for a particular type of task. This concept of reconfigurable by design must be presented in terms of the configuration patterns during the initial system-level design phase.

We can produce a graphical diagram of the configuration patterns for the second mapping of the matrix multiplication example. It can be seen from Figure 5-6 that an 8x8 matrix multiplication can be constructed by using two 4x4 structures with one of them a mirror image of the other. In addition, we need to configure the 3rd PE of the mirror image stripe to be an adder. Since in our resource model we have restricted the PE connection to 3 of its lateral nearest neighbours, connections between inter-strips need to obey this restriction. As can be seen, if there is no mirror image and to keep the same type of PEs on both sides, the output connection will tend to shift outwards and would potentially violate the connection restriction. So the actual configuration patterns only require specific interconnections and not the type of PE connections described in our design. This pattern will have a smaller area as in general each PE only needs to route to the nearest PE and the context register can also be reduced. We can stored this new pattern and extend it to support wider sized operations. However, our framework only presents a flexible system-level view of designing from pre-compiled circuits. We still

need to rely on the dynamic run-time support of the platform to provide services to transform, and to place and route the pre-built circuits at run-time.

Figure 5-6 A 4x4 and 8x8 matrix multiplication configuration patterns.

5.2.3 Performance analysis

We can analyze the performance of a few implementations using the two different types of extension interface (i.e. MicroBlaze extension and Nios-II ISE) that are presented in our framework. These performance metrics can then be compared and are classified in the following.

▪ MicroBlaze extension models

As the MicroBlaze extension models can be implemented using two different granularity execution cores (i.e. bit-serial PE and MAC-unit), we can compare their performance for the various application tasks. This is shown in Table 5-2.

Execution Models	Operations		
	MM (4x4) – 5000 times	2D DCT (JPEG)	2D DCT (MPEG)
Bit-serial			
Compute [cycles]	560000	1474560	2027520
Configure [cycles]	641280	1024	1024
MAC-unit			
Compute [cycles]	35000	92160	126720
Configure [cycles]	40080	64	64

Table 5-2 A comparison between bit-serial PE array and MAC-unit array for MicroBlaze extension model.

The compute cycles includes the number of FSL cycles for load/store operations and the number of CPU instruction cycles to perform the hardware execution. From Table 5-2, it can be seen that the bit-serial implementation is about 16 times slower than the MAC-unit implementation and this is expected. For the MM task, since it is computed over 5000 cycles where a different vector is used for every cycle, we can see that the configuration cycles are mainly to configure the multiplier constants and will exceed the number of computational cycles. To minimize the reconfiguration overhead, we can stagger these configuration cycles between non-computation cycles (e.g. load/store operations) using the dynamic reconfiguration attributes inherent in the microarchitecture design.

If a smaller resource area is desired then the bit-serial implementation can be selected. Otherwise the MAC-unit implementation can achieve a better speedup if required. Alternatively, for the relatively slower bit-serial implementation, a faster clock speed (maximum 100MHz on a Virtex-II FPGA) can be used to compensate for the slower execution datapath and the longer delay in loading operands to all the PEs.

▪ **MicroBlaze extension vs. Nios-II ISE**

The application examples described above were performed on the three models we have created; MicroBlaze CPU (software), MicroBlaze extension (MAC-unit array) and Nios-II extension (ISE). For the software-only model we have chosen the MicroBlaze CPU as we have made the assumption that the Nios-II CPU does not have an ALU that supports a multiplication operation. Using the parameters specified in our simulation framework

Application examples and Experimental results

for each component in the various models, we generated the computational time shown in Figure 5-7 for performing all the tasks on each of the models. The computational time is the sum of the execution time and reconfiguration time on the hardware, the FSL communication time (for MicroBlaze extension model only) to load/store operands/results between CPU and the hardware, and the CPU execution time (instruction fetch and decode, and write-back). The CPU execution time also includes executing memory access instructions to fetch and store data from the memory to the registers and vice versa.

Figure 5-7 Compute time comparison of the various kernel tasks.

Figure 5-8 Speedup comparison between the various models.

Figure 5-8 shows the speedup comparison between ISE vs. software, MAC-unit vs. software and MAC-unit vs. ISE. Both the ISE and MAC-unit extension show better performance than the software implementation with an average of 2.8 and 4.9 times speedup respectively over the software-only solution. In contrast, the speedup between

the ISE and MAC-unit is only an average of 1.65 times in favour of the MAC-unit solution. One possible contribution to this is that both extensions have inputs of similar width (i.e. 32-bit wide). Although the MAC-unit extension has more input channels to supply data to the hardware, it needs to find a large number of data sets to exploit its vertical parallelism, horizontal pipeline structure in order to gain significant speedup. Another explanation is that results are produced serially from the MAC-unit extension and the CPU will have to incur additional cycles to read the intermediate outputs. However, with some modification to the design and interleaving these read cycles we can improve the performance of the extension significantly. Interleaving instructions in the CPU with non-computational ones are easier to discover in the MAC-unit extension since the core is more loosely coupled than the ISE. Compiler techniques that adopt a more aggressive scheduling algorithm can further optimize the performance of these extensions.

The ISE offers another attractive option. It can be seen that with careful design of the custom functions where the multiplication and addition stages are pipelined, we can achieve a significant speedup with the same degree of flexibility over non-custom instructions in the software-only implementation. It is also possible to reduce the number of instructions used in some of the cases we highlight above. Generally, not all applications using these custom functions will have an improvement in the performance, but in the case where the speedup from a co-processor and the ISE are only marginal then it may be more economical to adopt a simpler ISE solution since the hardware design effort is less complex.

▪ **Nios-II ISE with MAC-unit array vs. MicroBlaze extension with custom functions**

We interchanged the co-processor core of the MicroBlaze extension (MAC-unit array) with the Nios-II ISE (custom functions) without actually implementing them so that the impact to the overall performance for each type of co-processor core on different architectural models (i.e. MicroBlaze based and Nios-II based) can be analyzed rapidly. The following indicates the clock performances used for the various components in each of the model to perform the simulation.

- MicroBlaze CPU: 100MHz, MAC-unit: 60MHz.
- Nios-II CPU: 100MHz, ISE (Custom functions): 100MHz.

Figure 5-9 Comparison between MicroBlaze with custom functions, MAC-unit and software-only implementation.

Firstly, we compared the compute time as shown in Figure 5-9 of the MicroBlaze-custom function model with the MAC-unit and software-only implementation for the various application tasks. The comparison shows that the MicroBlaze with custom function implementation degrades the performance by an average of about 40% compared to the MAC-unit implementation. This is mainly due to the slower custom function designs that only have a 2-stage pipeline and a small amount of data-level parallelism that can be provided by the design. However, the performance over software-only solution is still relatively decent indicating an average speedup of about 2.8 times which is similar to merely using the ISE implementation.

Figure 5-10 Comparison between ISE, ISE with MAC-unit and software-only implementation.

We performed the same comparison for the ISE with MAC-unit implementation shown in Figure 5-10. The ISE performance has been enhanced by an average of 10% using the MAC-unit core. The small gain in performance is contributed by the faster MAC-unit design and this also compensates for the limited data I/O bandwidth that is available to the co-processor core. Moreover, the implementation is complicated which requires more custom instructions to perform the same application tasks and the custom instructions have to be carefully designed so that the pipeline of the CPU does not stall unnecessarily while waiting for a longer execution time to complete. If the MAC-unit array size is doubled to support wider sized operations, then the performance gain is negligible as the number of load/store custom instructions will also be doubled. With a mere 10% gain margin, this implementation requires a considerable amount of design effort and also it does not scale well with large designs.

We can also examine the MAC-unit implementation on both the Nios-II CIL and the MicroBlaze FSL interface. It can be noted that the systolic array MAC-unit design favours the MicroBlaze extension model and supports the FSL design strategy to accelerate custom hardware execution without affecting the soft-core CPU performance (i.e. related to the overall system performance). The Nios-II CIL interface requires a more predictable hardware execution to allow the CPU to run at a higher speed utilizing the deep pipeline level of the CPU architecture. In our case, the speedup achieved over the software-only solution for both the ISE and the MicroBlaze-custom function implementations is only marginal with the latter a fraction slower. This is due to the longer delay on the FSL interface and the smaller pipeline level of the custom function designs requiring a large number of consecutive load/store operations to keep the CPU pipeline fully occupied. This shows that the MicroBlaze FSL interface model can adapt reasonably for small data level parallelism design without suffering a severe performance penalty. This is also aided by the FIFO-based FSL interface which is loosely-coupled to the internal datapath and does not degrade the CPU performance. However, the design effort is considerable compared to the ISE implementation which is simpler.

CHAPTER 6

CONCLUSION AND RECOMMENDATIONS

6.1 CONCLUSION

Enabling hardware-software co-simulation at the system-level is an important concept in embedded system design. A common approach is to use multiple sources and platforms (e.g. software instruction set simulators with HDL logic simulators) to achieve hardware-software co-simulation. This approach requires a considerable amount of integration effort between the various simulation environments and is less flexible to customize for other types of microarchitectures. Moreover, it is difficult to manage the synchronization between the environments to give a realistic indication of the actual implementation. The motivation for using software-based modeling techniques (e.g. SystemC) is that a common language can be used to design and model the hardware and software characteristics concurrently for hybrid architectures. Hardware design quality can also be maintained since it employs a similar methodology to HDL. Consequently, this improves the productivity of embedded system design as a single source can be used throughout the design flow.

To enable hardware-software co-simulation using software-based modeling platforms, we have developed a framework using SystemC that consists of distinctive hardware and software models of a reconfigurable architecture. The software model (i.e. CPU model) represents the ISA and provides an instruction set simulator. The hardware component will represent the microarchitecture of the co-processor core. Both models can be simulated concurrently using hardware clocked modeling of the SystemC simulation kernel. We showcased two different types of CPU extension models that have a loosely-coupled and tightly-coupled interface between the CPU and the co-processor. These two models closely resemble the MicroBlaze and Nios-II soft-core CPU-based implementation respectively. An application task can be partitioned accordingly to the hardware and software models for each type of architecture as they are presented as an explicit programming model to the application developer. Experiments on some commonly used application tasks are conducted and multiple hardware solutions are analyzed in terms of the speedup achieved over the software-

Conclusion and Recommendations

only implementations. The key feature of our framework is that the same design sources to generate the simulation executable can also be translated directly for synthesis, implementation and further optimization using commercial back-end tools. This presents a more economical approach for our compilation flow. We have also shown a mechanism in our compilation flow whereby the refinement-validation process can be performed easily.

The experiments presented in our studies have shown great potential of using the SystemC platform to enable hardware-software co-simulation for reconfigurable architectures. With the continual improvement in software-based modeling techniques (e.g. support for abstract RTOS and scheduler modeling), it will become a widely adopted approach for future embedded system design.

6.2 RECOMMENDATIONS

Although we have presented a flexible simulation and compilation framework that can rapidly explore different reconfigurable hardware implementations during the initial design phase, other interesting areas should be investigated and are highlighted in the followings.

- i) Our framework only models the static execution cores where the reconfiguration characteristics are inherent in the microarchitecture designs during compile-time. It would be advantageous to model the mechanism (e.g. Xilinx Virtex II ICAP interface) that enables device reconfiguration especially for self-reconfigurable platforms. This will allow the analysis of the system's run-time requirements to support dynamic 'swapping' of the static cores. Also the API to provide a lightweight approach for an embedded CPU to enable self-reconfiguration should be incorporated in the framework.
- ii) The configuration patterns generated by our framework should be further analyzed with respect to the synthesis results produced by the employed commercial back-end tool. This will allow the optimum patterns for each type of tasks with the least device configuration overhead to be identified and stored as pre-compiled circuits. Designs that support wider-sized operation can be customized from these pre-compiled circuits for the particular task.
- iii) Our framework can also be extended to evaluate an instruction scheduling algorithm for dynamic hardware configurations. This will provide a compiler technique to minimize the number of hardware reconfiguration instructions by considering some area placement constraints when multiple tasks or circuits can coexists in the same physical space.

REFERENCES

- [Arnold] Jeffery M. Arnold. "S5: The Architecture and Development Flow of a Software Configurable Processor". In *IEEE Conference on Field-Programmable Technology*, Dec 2005.
- [Austin] T. Austin, E. Larson and D. Ernst. "SimpleScalar: An infrastructure for computer system modeling". In *IEEE Symposium on Field-Programmable Custom Computing Machines (FCCM)*, 1997, pp12-21.
- [Bashford] S. Bashford et al. "The MIMOLA Language Version 4.1". *Technical Report, Lehrstuhl Informatik XII, Univ. Dortmund*, September 1994.
- [Benini] [62] L. Benini, D. Bertozzi, D. Bruni, N. Drago, F. Fummi, and M. Poncino, "SystemC cosimulation and emulation of multiprocessor SoC designs", *IEEE Computer*, Vol. 36, No.4 pp. 53-59, April 2003.
- [Betz] V. Betz and J. Rose. "VPR: A New Packing, Placement and Routing Tool for FPGA Research". *1997 International Workshop on Field Programmable Logic and Applications*.
- [Bhasker] J. Bhasker. "A SystemC Primer". *Star Galaxy Publishing*, 2002.
- [Blodget] B. Blodget, P. J-Roxby, E. Keller, S. McMilian and P. Sundararajan. "A Self-reconfigurable Platform". *Proc. 13th Int'l Conf. on Field-Programmable Logic and Application (FPL '03)*, pp. 565 – 574, Sept 2003.
- [Burns] J. Burns, A. Donlin, J. Hogg, S. Singh and M. de Wit. "A Dynamic Reconfiguration Run-Time System". *FCCM 1997*, pp 66-76.
- [Carrillo] J. E. Carrillo et al. "The effect of reconfigurable units in superscalar processors". In *Proc. 9th ACM International Symposium on Field Programmable Gate Arrays (FPGA)*, 2001.
- [CEX] Celoxica. <http://www.celoxica.com>
- [Chang] P.P. Chang et al. "IMPACT: An architectural framework for multiple-instruction-issue processors". In *Proc. of the 18th International Symposium on Computer Architecture*, pp. 266-275
- [DeHon] A. DeHon. "DPGA-coupled microprocessors: Commodity Ics for the early 21st century". In *Proc. 2nd IEEE Workshop on FPGAs for Custom Computing Machines (FCCM)*, 1994, pp. 31-39.
- [Diessel] O. Diessel and H. ElGindy. "Run-Time Compaction of FPGA Designs". *Proc. 7th Int'l Workshop on Field-Programmable Logic and Application (FPL '97)*.
- [Dyer] M. Dyer, C. Plessl and M. Platzner. "Partially Reconfigurable Cores for Xilinx Virtex". In *Proc. of the Reconfigurable Computing Is Going Mainstream, 12th International Conference on Field Programmable Logic and Applications*, 2002.
- [Enzler] Rolf Enzler et al. "Co-simulation of a Hybrid Multi-Context Architecture". In *Proc. of the International Conference on Engineering of Reconfigurable Systems and Algorithms*, June 2003.
- [ImpC] Impulse. <http://www.impulsec.com>
- [Fan] K. Fan, M. Kudlur, H. Park, S. Mahlke, "Compiler-directed Synthesis of Multifunction Loop Accelerators". *Workshop on Application Specific Processors (WASP), in conjunction with CODES+ISSS*, September 2005.
- [Grötke] T. Grötke et al. *System Design with SystemC*. Kluwer, 2002.

- [Grötke2] T. Grötke. Modeling Software with SystemC 3.0. In *6th European SystemC Users Group Meeting*, 2002.
- [Guccione] S.A. Guccione and D. Levi. "JBits: A Java-Based Interface to FPGA Hardware". *SPIE Proceedings, Vol. 3526*, pp. 97-102, November 1998.
- [Gyllenhaal] J.C.Gyllenhaal, B.R. Rau and W.W. Hwu. "*Hmdes Version 2.0 Specification*". Technical Report, IMPACT-96-3, IMPACT Research Group, Univ. of Illinois, Urbana, IL, 1996.
- [Halambi] A. Halambi et al. "EXPRESSION: A language for architecture exploration through compiler /simulator retargetability". In *Proc. DATE*, 1999.
- [Hardnett] Charles R. Hardnett, Ajay Jayaraj, Tushar Kumar, Krishna V. Palem and Sudhakar Yalamanchili. "Compiling Stream Kernels for Polymorphous Computing Architectures". *The Twelfth International Conference on Parallel Architectures and Compilation Techniques*, Sept.-Oct. 2003.
- [Hauser] J.R. Hauser and J. Wawrzynek. "Garp: a MIPS processor with a reconfigurable coprocessor". In *Proc. of the 5th IEEE Symposium on FPGA-Based Custom Computing Machines (FCCM '97)*.
- [Hoffman] Andreas Hoffman et al. "A Framework for Fast Hardware-software Co-simulation". In *Proc. of the Conference on Design, automation and test in Europe*, 2001.
- [Kapasi] U.J. Kapasi, W.J. Dally, S. Rixner, J.D. Owens and B. Khailany. "The Imagine Stream Processor". *Proc. IEEE Conference on Computer Design, Sept 16-18, 2002, Freiburg, Germany*, pp. 282-288.
- [Keller] E. Keller. "JRoute: A Run-Time Routing API for FPGA Hardware". In *7th Reconfigurable Architectures Workshop, Lecture Notes in Computer Science 1800*, pp 874-881, Cancun, Mexico, May 2000.
- [Lang] H.-W. Lang, R. Maab and M. Schimmler. "The Instruction Systolic Array – Implementation of a Low-Cost Parallel Architecture as Add-On Board for Personal Computers". In *Proc. HPCN'94, LCNS 797*, Springer-Verlag, 1994.
- [Lava] S. Singh and P.James-Roxby. "Lava and JBits: From HDL to Bitstream in Seconds".
In *Proceedings of the IEEE Symposium on FPGAs for Custom Computing Machines (FCCM)*, 2001.
- [Lee] C. Lee et al. "MediaBench: A Tool for Evaluating and Synthesizing Multimedia and Communication Systems". In *Proc. of the 30th Annual International Symposium on Microarchitecture*, 1997.
- [Li] Y. Li, T. Callahan, E. Darnell, R. Harr, U. Kurkure and J. Stockwood. "Hardware-Software Co-Design of Embedded Reconfigurable Architectures". *IEEE DAC 2000, Los Angeles, California*.
- [Lim] H. Lim, P.Vincenzo, E. Swartzlander. "A Serial-Parallel Architecture for Two-Dimensional Discrete Cosine and Inverse Discrete Cosine Transforms." In *IEEE Transactions on Computers, Vol. 49, No. 12*, December 2000.
- [Mirsky] E. Mirsky and A. DeHon. "MATRIX: A Reconfigurable Computing Architecture with Configurable Instruction Distribution and Deployable Resources." In *Proc. IEEE Sym on FPGAs for Custom Comp. Mach.*, 1996, pp. 157-66
- [Monarch] <http://www.isi.edu/asd/monarch/>

- [Mooney] V. Mooney and G. De Micheli. "Hardware/Software Co-Design of Run-Time Schedulers for Real-Time Systems." *Design Automation for Embedded Systems*, vol. 6, pp. 89-114, 2000.
- [Nios-II] Altera Corporation. "Nios II Processor Reference Handbook". October 2005.
- [NiosCIL] Altera Corporation. "Nios II Custom Instruction". *User Guide*, December 2004.
- [OSCI] Open SystemC Initiative. <http://www.systemc.org>
- [Panainte] E. M. Panainte et al. "Instruction Scheduling for Dynamic Hardware Configurations". In *Design, Automation & Test in Europe (DATE '05)*, vol 1, pp. 100-105.
- [Palem] K.V. Palem, S. Talla, and W.-F. Wong, "Compiler Optimizations for Adaptive EPIC Processors". In *Proc. ACM Conf. Embedded Software*, pp. 257-273, 2001.
- [Pees] S. Pees et al., "LISA Machine Description Language for Cycle-Accurate Models of Programmable DSP Architectures," *Proc. 36th Design Automation Conference (DAC 99)*, New Orleans, LA., June 1999, pp. 933—938.
- [Pillai] L. Pillai. "Video Compression using DCT". In *Application Note: Virtex-II series*, March 2005.
- [Renau] J. Renau, J. Tuck, W. Liu and J. Torrellas. "Morphable Multithreaded Memory Tiles (M3T) Architecture". *University of Illinois UIUC-CS Technical Report*, September 2002.
- [Rosinger] H. Rosinger. "Connecting Customized IP to the MicroBlaze Soft Processor Using the Fast Simplex Link (FSL) channel". In *Application Note: MicroBlaze*, March 2004.
- [Singh] H. Singh et al. "MorphoSys: An Integrated Re-configurable Architecture". In *Proc. of the Nato Symposium on System Concepts and Integration*, Monterey, CA, April 1998.
- [SysCr] SystemCrafter. <http://www.systemcrafter.com>
- [STOC] SpecC Technology Open Consortium. <http://www.specc.org>
- [Synplicity] Synplicity Inc. "Synplify Pro", *User Guide and Tutorial*, Feb 2001.
- [Taylor] M. Taylor et al. "The Raw Microprocessor: A Computational Fabric for Software Circuits and General Purpose Programs". *IEEE Micro*. Mar/Apr 2002.
- [Tenison] <http://www.tenison.com>
- [Tri] <http://www.trimaran.org>
- [uBlaze] Xilinx. "MicroBlaze Processor Reference Guide". *User guide*, April 2005.
- [Vassiliadis] S. Vassiliadis et al. "The molen programming paradigm". In *Proc. of SAMOS, LNCS, vol 3133, pp 1-10, Greece*, July 2003.
- [Walder] H. Walder and M. Platzner. "Online Scheduling for Block-partitioned Reconfigurable Devices". In *Design, Automation & Test in Europe (DATE'03)*
- [Watt] A. Watt. "3D Computer Graphics". *2nd Edition*, Addison-Wesley Publishers Ltd., Essex, England.
- [Wei] K. Wei, R. Kistner, A. Kunzmann and W. Rosenstiel. "Analysis of the XC6000 Architecture for Embedded System Design". *IEEE Symposium on FPGAs for Custom Computing Machines*, April 15-17, 1998.

- [Wolinski] C. Wolinski, M. Gokhale and K. McCabe.” A Polymorphous Computing Fabric”. *IEEE Micro, Design and Test of Systems on Chip*, Sept/Oct 2002.
- [XApp290] Xilinx. “XAPP290: Two Flows for Partial Reconfiguration: Module Based or Difference Based”. November, 2003.
- [xilinxFSL] Xilinx. “Fast Simplex Link (FSL) Bus (v2.00a)”. *Product Specification*, May 2004.
- [Xu] <http://www-unix.ecs.umass.edu/~wxu/jbits/>

APPENDIX A

A1. Supported MicroBlaze instruction set

Mnemonic, Operand	Description	Cycles	Encoding: 0-5, 6-10
ADD Rd, Ra, Rb	Add contents of Ra and Rb	1	000000, ddddd
ADDC Rd, Ra, Rb	Add contents of Ra and Rb with carry	1	000010, ddddd
ADDI Rd, Ra, k	Add contents of Ra and immediate	1	001100, ddddd
ADDIC Rd, Ra, k	Add contents of Ra and immediate with carry	1	001010, ddddd
RSUB Rd, Ra, Rb	Subtract contents of Rb with Ra	1	000001, ddddd
RSUBC Rd, Ra, Rb	Subtract contents of Rb with Ra and carry	1	000011, ddddd
RSUBI Rd, Ra, k	Subtract contents of Rb with Ra and immediate	1	001001, ddddd
MUL Rd, Ra, Rb	Multiply contents of Ra with Rb	3	010000, ddddd
MULI Rd, Ra, k	Multiply contents of Ra with immediate	3	011000, ddddd
OR Rd, Ra, Rb	Contents of Ra are ORed with Rb	1	100000, ddddd
AND Rd, Ra, Rb	Contents of Ra are ANDed with Rb	1	100001, ddddd
XOR Rd, Ra, Rb	Contents of Ra are XORed with Rb	1	100010, ddddd
BEQ Ra, Rb	Branch if Ra =0, to the offset value in Rb	1/2	100111, 00000
BNE Ra, Rb	Branch if Ra ≠ 0, to the offset value in Rb	1/2	100111, 00001
BLT Ra, Rb	Branch if Ra < 0, to the offset value in Rb	1/2	100111, 00010
BLE Ra, Rb	Branch if Ra ≤0, to the offset value in Rb	1/2	100111, 00011
BGT Ra, Rb	Branch if Ra > 0, to the offset value in Rb	1/2	100111, 00100
BGE Ra, Rb	Branch if Ra ≥= 0, to the offset value in Rb	1/2	100111, 00101
LBU Rd, Ra, Rb	Loads an unsigned byte from memory location specified by adding contents of Ra and Rb	2	110000, ddddd
LHU Rd, Ra, Rb	Loads a halfword from memory location specified by adding contents of Ra and Rb	2	110001, ddddd
LW Rd, Ra, Rb	Loads a word (32 bits) from memory location that results by adding Ra and Rb.	2	110010, ddddd
SB Rd, Ra, Rb	Stores LS byte of Rd into memory location specified by adding contents of Ra and Rb.	2	110100, ddddd
SH Rd, Ra, Rb	Stores LS halfword of Rd into memory location specified by adding contents of Ra and Rb	2	110101, ddddd
SW Rd, Ra, Rb	Stores a word of Rd into memory location specified by adding contents of Ra and Rb	2	110110, ddddd
LBUI Rd, Ra, k	Loads an unsigned byte from memory location specified by adding Ra and immediate k value	2	111000, ddddd
LHUI Rd, Ra, k	Loads a halfword from memory location specified by adding Ra and immediate k value	2	111001, ddddd
LWI Rd, Ra, k	Loads a word from memory location specified by adding Ra and immediate k value	2	111010, ddddd
SBI Rd, Ra, k	Stores LS byte of Rd into memory location specified by adding contents of Ra and k value	2	111100, ddddd
SHI Rd, Ra, k	Stores LS halfword of Rd into memory location specified by adding contents of Ra and k value	2	111101, ddddd
SWI Rd, Ra, k	Stores LS word of Rd into memory location specified by adding contents of Ra and k value	2	111110, ddddd

Ra: source operand a, general purpose register R0-R31
Rb: source operand b, general purpose register R0-R31
Rd: destination operand, general purpose register R0-R31
k: 16-bit immediate value
FSLx: 3-bit FSL port number
&: Concatenate
LS: Least significant

Mnemonic, Operand	Description	Cycles	Encoding: 0-5, 6-10, 11-15, 16-20, 21-31
GET Rd, FSLx	Read from the FSLx interface and put contents into Rd (blocking)	-	011011, dddddd, 00000, 00000000000000& FSLx
PUT Ra, FSLx	Write the value from Ra to the FSLx interface (blocking)	-	011011, 00000, aaaaaa, 10000000000000& FSLx
NGET Rd, FSLx	Read from FSLx interface and put contents into Rd (non-blocking)	2	011011, dddddd, 00000, 01000000000000& FSLx
NPUT Ra, FSLx	Write the value from Ra to the FSLx interface (non-blocking)	2	011011, 00000, aaaaaa, 11000000000000& FSLx

Ra: source operand a, general purpose register R0-R31
Rb: source operand b, general purpose register R0-R31
Rd: destination operand, general purpose register R0-R31
k: 16-bit immediate value
FSLx: 3-bit FSL port number
&: Concatenate

A2. Supported Nios-II instruction set

Mnemonic, Operand	Description	Encoding: 21-17, 16-6, 5-0
add Rd, Ra, Rb	Calculates the sum of Ra and Rb. Stores result in Rd.	dddd, 11000100000, 111010
addi Rd, Ra, k	Sign-extends the 16-bit immediate value and adds it to Ra.	kkkkk, kkkkkkkkkkkk, 000100
sub Rd, Ra, Rb	Subtract Rb from Ra.	dddd, 11100100000, 111010
subi Rd, Ra, k	Sign-extends the immediate value to 32 bits and subtracts it from Ra. (addi Rd, Ra, -k)	kkkkk, kkkkkkkkkkkk, 000100
and Rd, Ra, Rb	Bitwise logical AND of Ra and Rb.	dddd, 00111000000, 111010
or Rd, Ra, Rb	Bitwise logical OR of Ra and Rb.	dddd, 01011000000, 111010
orhi Rd, Ra, k	Bitwise logical OR of high halfword of Ra and immediate.	kkkkk, kkkkkkkkkkkk, 110100
xor Rd, Ra, Rb	Bitwise logical XOR of Ra and Rb.	dddd, 01111000000, 111010
beq Ra, Rb, k	Branch if Ra = Rb.	kkkkk, kkkkkkkkkkkk, 100110
bge Ra, Rb, k	Branch if Ra >= Rb.	kkkkk, kkkkkkkkkkkk, 001110
bgt Ra, Rb, k	Branch if Ra > Rb.	kkkkk, kkkkkkkkkkkk, 010110
bne Ra, Rb, k	Branch if Ra $\neq$ Rb.	kkkkk, kkkkkkkkkkkk, 011110
blt Ra, Rb, k	Branch if Ra < Rb.	kkkkk, kkkkkkkkkkkk, 010110
mov Rd, Ra	Moves contents of Ra to Rd. (add Rd, Ra, Rb->0).	dddd, 11000100000, 111010
movhi Rd, k	Writes the immediate value into high halfword of Rd. (orhi Rd, Ra, k)	kkkkk, kkkkkkkkkkkk, 110100
movi Rd, k	Sign-extends immediate value to 32 bits and writes to Rd. (addi Rd, Ra->0, k)	kkkkk, kkkkkkkkkkkk, 000100
ldb Rd, Ra(k)	Load register Rd with the desired byte, sign extending the 8-bit value to 32-bit value from address location specified by the sum of Ra and immediate value k.	kkkkk, kkkkkkkkkkkk, 000111
ldbu Rd, Ra(k)	Load register Rd with desired byte, zero extending the 8-bit value to 32-bit value from address location specified by the sum of Ra and immediate value k.	kkkkk, kkkkkkkkkkkk, 000011
ldh Rd, Ra(k)	Load register Rd with desired halfword, sign extending the 16-bit value to 32-bit value from address location specified by the sum of Ra and immediate value k. Byte address must be multiple of 2.	kkkkk, kkkkkkkkkkkk, 001111
ldw Rd, Ra(k)	Load register Rd with desired word from address location specified by the sum of Ra and immediate value k. Byte address must be multiple of 4.	kkkkk, kkkkkkkkkkkk, 010111
nop	Idle operation. (add Rd, Ra->0, Rb->0)	dddd, 11000100000, 111010
stb Ra, Rd(k)	Stores the low byte of Ra to the address location specified by the sum of Rd and immediate value k.	kkkkk, kkkkkkkkkkkk, 000101
sth Ra, Rd(k)	Stores the low halfword of Ra to the address location specified by the sum of Rd and immediate value k. Byte address must be multiple of 2.	kkkkk, kkkkkkkkkkkk, 001101
stw Ra, Rd(k)	Stores Rb to the address location specified by the sum of Rd and immediate value k. Byte address must be multiple of 4.	kkkkk, kkkkkkkkkkkk, 010101
cmpeq Rd, Ra, Rb	Stores 1 to Rd if Ra = Rb; otherwise stores 0 to Rd.	dddd, 10000000000, 111010
cmpeqi Rd, Ra, k	Sign extends the 16-bit immediate value to 32 bits and compares it to Ra. Stores 1 to Rd if Ra = k; otherwise stores 0 to Rd.	kkkkk, kkkkkkkkkkkk, 100000
cmpge Rd, Ra, Rb	Stores 1 to Rd if Ra >= Rb; otherwise stores 0 to Rd.	dddd, 00100000000, 111010
cmpgei Rd, Ra, k	Sign extends the 16-bit immediate value to 32 bits and compares it to Ra. Stores 1 to Rd if Ra >= k; otherwise stores 0 to Rd.	kkkkk, kkkkkkkkkkkk, 001000
cmpgt Rd, Ra, Rb	Stores 1 to Rd if Ra > Rb; otherwise stores 0 to Rd.	dddd, 01000000000, 111010
cmpgti Rd, Ra, k	Sign extends the 16-bit immediate value to 32 bits and compares it to Ra. Store 1 to Rd if Ra > k; otherwise stores 0 to Rd.	kkkkk, kkkkkkkkkkkk, 001000
cmplt Rd, Ra, Rb	Stores 1 to Rd if Ra < Rb; otherwise stores 0 to Rd.	dddd, 01000000000, 111010

Mnemonic, Operand	Description	Encoding: 21-17, 16-6, 5-0
cmplti Rd, Ra, k	Sign extends the 16-bit immediate value to 32 bits and compares it to Ra. Store 1 to Rd if Ra < k; otherwise stores 0 to Rd.	kkkkk, kkkkkkkkkkkk, 010000
cmple Rd, Ra, Rb	Stores 1 to Rd if Ra <= Rb; otherwise stores 0 to Rd.	dddd, 00100000000, 111010
cmplei Rd, Ra, k	Sign extends the 16-bit immediate value to 32 bits and compares it to Ra. Store 1 to Rd if Ra <= k; otherwise stores 0 to Rd.	kkkkk, kkkkkkkkkkkk, 010000
cmpne Rd, Ra, Rb	Stores 1 to Rd if Ra ≠ Rb; otherwise stores 0 to Rd.	dddd, 01100000000, 111010
cmpnei Rd, Ra, Rb	Sign extends the 16-bit immediate value to 32 bits and compares it to Ra. Store 1 to Rd if Ra ≠ k; otherwise stores 0 to Rd.	kkkkk, kkkkkkkkkkkk, 011000
custom n Rd, Ra, Rb	Custom function operation.	See Page A7.

Ra: register index of source operand A
Rb: register index of source operand B
Rd: register index of destination operand
k: 16-bit immediate value
n: 8-bit number that selects instruction

A3. FSL

Figure of FSL write operation

Figure of FSL read operation

A4. Bit-serial PE

Figure of a 16-bit serial multiplier structure

Symbol
a: operand *a*
b: operand *b*
s: sum (least significant bit first)
FA: Full Adder
C_{in}: Carry in
C_{out}: Carry out
&: Contents of *a* ANDed with *b*

A5. Custom instruction logic

Figure of custom instruction set

Assembler Syntax: *custom N, rC, rA, rB*

Instruction fields: *A- register index of operand A*
 B- register index of operand B
 C- register index of operand C
 N – 8-bit number that selects instruction
 readra – 1 if instruction uses rA, 0 otherwise.
 readrb – 1 if instruction uses rB, 0 otherwise.
 writerc – 1 if instruction provides result for rC, 0 otherwise.

Figure of Multi-cycle custom instruction timing diagram

Nios-II/s core instruction execution performance

Instructions	Cycles
Normal ALU instructions	1
Multi-cycle custom instructions	1
Branch (not taken/taken)	1 / 2
Load / store	1
Shift/rotate (without hardware multiply present)	1 to 32