Evaluation of the OneChip Reconfigurable Processor

by

Jorge Ernesto Carrillo Esparza

A thesis submitted in conformity with the requirements
for the degree of Master of Applied Science
Graduate Department of Electrical and Computer Engineering,
University of Toronto

Evaluation of the OneChip Reconfigurable Processor

Jorge Ernesto Carrillo Esparza

Master of Applied Science, 2000

Department of Electrical and Computer Engineering

University of Toronto

Abstract

The concept of a reconfigurable processor comes from the idea of having a general-purpose processor coupled with some reconfigurable resources that allow implementation of custom application-specific instructions. This thesis describes *OneChip*, a third generation reconfigurable processor architecture that integrates a Reconfigurable Functional Unit (RFU) into a superscalar Reduced Instruction Set Computer (RISC) processor's pipeline. The architecture allows dynamic scheduling and dynamic reconfiguration.

To evaluate the performance of the OneChip architecture, several off-the-shelf software applications were compiled and executed on *Sim-OneChip*, an architecture simulator for OneChip which includes a software environment for programming the system. The architecture is compared to a similar one but without dynamic scheduling and without an RFU. OneChip achieves a performance improvement and shows a speedup range from 2.16 up to 32.02 for the different applications and data sizes used.

Acknowledgements

I would like to thank my supervisor, Professor Paul Chow, for his advice and academic support during my graduate studies; for his guidance in organizing and writing this thesis; and for giving me the opportunity and freedom to pursue my goals in such an interesting area.

I also want to thank my family for their encouragement and emotional support throughout my life; for their love, which makes it easier to believe in myself and motivates me to keep going on; and for teaching me that the long term results of being persitent and patient make the effort worth while.

It is the friends we encounter along life's path who help us appreciate the journey. To all of them that once asked me the question "How is your thesis going?" and to which I never answered, here is the answer: IT CAME OUT GREAT! Thank you all.

Finally, I would like to thank the University of Toronto Fellowships and Chameleon Systems Inc. for providing the financial support needed to complete this work.

Jorge E. Carrillo E.

Table of Contents

List of Figures

List of Tables

Acronyms

ALU	Arithmetic Logic Unit
BLT	Block Lock Table
CAD	Computer Aided Design
CPU	Central Processing Unit
DPGA	Dynamically Programmable Gate Array
EEPROM	Electronically Erasable and Programmable Read-Only Memory
EPROM	Electronically Programmable Read-Only Memory
FPGA	Field Programmable Gate Array
HDL	Hardware Description Language
HPL	High-level Programming Language
ILP	Instruction-Level Parallelism
LRU	Least Recently Used
LSQ	Load-Store Queue
RAM	Random Access Memory
RecQ	Reconfigurable Instructions Queue
RFU	Reconfigurable Functional Unit
RISC	Reduced Instruction Set Computer
RBT	Reconfiguration Bits Table
RUU	Register Update Unit
SRAM	Static Random Access Memory

VHDL	VHSIC Hardware Description Language
VHSIC	Very High Speed Integrated Circuit
VLIW	Very Long Instruction Word

CHAPTER 1

Introduction

Recently, the idea of using reconfigurable resources along with a conventional processor has lead to research in the area of reconfigurable computing. The main goal is to take advantage of the capabilities and features of both resources. While the processor takes care of all the general-purpose computation, the reconfigurable hardware acts as a specialized coprocessor that takes care of specialized applications.

With such platforms, specific properties of applications, such as parallelism, regularity of computation, and data granularity can be exploited by creating custom operators, pipelines, and interconnection pathways. By allowing the programmer to change the hardware of the machine to match the task at hand, computational efficiency can be increased and an improvement in performance realized.

1.1 Motivation

There has been research done in the Department of Electrical and Computer Engineering at the University of Toronto in the area of reconfigurable processors. Namely, the OneChip processor model has been developed. At first, this model tightly integrated reconfigurable logic resources and memory into a fixed-logic processor core. By using the reconfigurable units of this architecture, the execution time of specialized applications was reduced. The model was mapped into the Transmogrifier-1 field-programmable system. This work was done by Ralph Wittig [1][2].

A follow-on model, called OneChip-98, then integrated a memory-consistent interface. It is a hardware implementation that allows the processor and the reconfigurable array to operate concurrently. It also provides a scheme for specifying reconfigurable instructions that are suitable for typical programming models. This model was mapped into the Transmogrifier-2 field-programmable system. This work was done by Jeff Jacob [3][4].

It is desirable to extend the architecture to a superscalar processor that allows multiple instructions to issue simultaneously and perform out-of-order execution. This should lead to much better performance, since the processor and the reconfigurable logic may execute several instructions in parallel.

Up to now, subsets of the OneChip architecture have been modeled by implementing them in hardware. However, to properly determine the feasibility of the architecture, it is necessary to build a full software model that can simulate real applications.

It is expected that most of the performance improvement that this architecture will show will come from memory streaming applications, that is, those applications that read in a block of data from memory, perform some computation on it, and write it back to memory. Multimedia applications have this characteristic and will be used to evaluate the architecture.

1.2 Objectives

The main objective of this research is to study the behavior of the OneChip[1] architecture model and measure its performance by executing several off-the-shelf software applications on a software model of the system. This requires the development of a simulator for the

1. *OneChip* will be used to refer to the OneChip-98 architecture.

OneChip system, as well as generating a software environment for programming it. This includes a C compiler that targets the model and the required libraries to support the architecture.

Other goals are to build a set of multimedia-type benchmarks suitable for the OneChip model. These applications will be profiled and analyzed to find pieces of code that can be ported and changed to a hardware version of them. The tests and results of the new modified versions will be discussed and, finally, this will lead to ideas and a proposal for future research in the area.

1.3 Thesis Organization

This thesis is divided into six chapters. Chapter 2 provides the reader with background information on existing reconfigurable processor architectures and configurations, as well as the current software technologies that support them. Chapter 3 describes the architecture of the OneChip reconfigurable processor. Chapter 4 talks about Sim-OneChip, an architecture simulator for OneChip. Chapter 5 describes the experimental setup, benchmark applications used for evaluating the processor, and results of the tests. Chapter 6 concludes this thesis and offers recommendations for future work.

CHAPTER 2

Background

The concept of a *reconfigurable processor* comes from the idea of having a general-purpose processor coupled with some reconfigurable resources that allow implementation of custom application-specific instructions. The goal is to improve performance by migrating certain parts of the software into specialized hardware.

This chapter presents a brief overview of reconfigurable processor architectures, as well as current compilation technologies that support them. The reader already familiar with processor architectures, programmable logic, or compiler technologies may skip the corresponding sections. A brief survey of current reconfigurable systems is also presented. This overview will introduce the reader to current issues and research efforts done in the area of reconfigurable computing. Major proposed schemes done by other research groups are also presented.

2.1 Reconfigurable processor architectures

When measuring the performance of a processor, we could consider two things: execution time, which is the time elapsed between the start and the completion of a task; or throughput, which is how much work can be done in a given period.

If we consider execution time, also referred to as response time, as a metric for measuring performance, we are concerned about reducing it to get better performance. Several ways to accomplish this will be briefly discussed.

Execution time is affected by three major factors: Instructions per program, Clock cycles per instruction, and Seconds per clock cycle as shown on (EQ 1) [6]. Sometimes by improving one of these, the others may be affected.

$$\text{Execution time} = \frac{\text{Instructions}}{\text{Program}} \times \frac{\text{Clock cycles}}{\text{Instruction}} \times \frac{\text{Seconds}}{\text{Clock cycle}} \tag{EQ 1}$$

For the purpose of this thesis, the last factor is not a major concern because the clock rate is related to the hardware technology rather than the architecture and is difficult to estimate. Hence only the first two will be taken into account and execution time will be measured in Clock cycles per program. By reducing the execution time of a program, we are improving the performance of a system, The program will take less time to execute, so we say that we are obtaining “speedup”.

2.1.1 Techniques to speed up processors

There are several techniques used now a days to speed up processors: Pipelining, superscalar pipelining, dynamic scheduling and very long instruction word (VLIW) packing. The purpose is to take advantage of a potential execution overlap among independent instructions. This characteristic is called Instruction-level parallelism (ILP).

Pipelining is a technique in which multiple instructions are allowed to overlap in execution. The idea of pipelining is to divide the execution of one instruction into steps, which are called pipeline stages. Each stage makes some contribution to the instruction and can operate in parallel with other stages. When one instruction is done in one stage, the instruction moves on to the next stage until there are no more stages and the execution of the instruction is completed. This way, while some stage is making its own contribution to the instruction, other stages can be making their respective contribution to other instructions at the same time. Pipelining improves performance by increasing instruction throughput. It does not decrease the time to execute individual instructions, but it will execute more instructions per time unit, resulting in speed up.

There are some situations, called pipeline hazards, that prevent the ideal pipeline from operating as intended. They are classified into:

- Structural hazards, which arise when two stages try to use the same resource at the same time with no hardware support for it. These are solved by duplicating hardware in the processor.
- Data hazards, which arise when an instruction depends on a result of a previous instruction. These are solved by adding structures to the pipeline to make the data available at the required stage.
- Control hazards, which arise from instructions that can change the program flow. These are solved by predicting the program flow and doing speculative execution (i.e. executing instructions before knowing if they should actually execute), or by using a delayed decision and inserting instructions with no dependencies in between.

There can be some cases when it is not possible to solve the hazards or the solution is too expensive. In this case, a stall of the pipeline can be used, which will stop any following instructions for a number of cycles until the hazard disappears.

The second technique used to speed up processors is called *superscalar pipelining*. The idea is to use more hardware resources, referred as functional units, so that the processor can launch multiple instructions in every pipeline stage as well as handling multi-cycle operations.

With more complex instructions such as floating point, it would be impractical to require all operations in a processor to complete in one clock cycle. The approach is to make some of the instructions use more clock cycles for their execution. This will produce instructions that have longer operation latency than others (i.e. with different clock cycles needed to produce a result), as well as different issue latencies or initiation intervals (i.e. the number of cycles before another operation of the same type can be issued to the same resource). This means that each functional unit could also be internally pipelined. With this technique, when the processor is executing simple short-latency instructions, there will be a high throughput and only when a long-latency instruction is executing, the instructions behind will wait for it to complete.

The third technique is called *dynamic scheduling* or dynamic pipelining. The idea is to add extra hardware resources so that later instructions with no hazards can proceed in parallel in the presence of an instruction stall. For this, multiple functional units are required in the architecture.

There are two ways of implementing dynamic scheduling. The first is by using a scoreboard, which takes control of issuing and executing instructions, as well as detecting hazards between them. The scoreboard basically decides when each issued instruction can read its source operands, begin execution and write the result to the destination.

The second approach to implement dynamic scheduling is by using Tomasulo’s scheme [6]. It is a very complex scheme that shares ideas from the scoreboard scheme. The idea is to introduce instruction buffers for the functional units, where the instructions will be queued. These buffers are called reservation stations and they hold the instruction operands as soon as they become available, otherwise they hold the information of which other reservation station will produce the operands they need. This is a form of register renaming, which reduces some types of hazards.

Lastly, the fourth technique is referred to as *very long instruction word* (VLIW) packing. It allows multiple instruction issue per cycle as in superscalar pipelining, but this one relies on compiler technologies. The idea is to pack a fixed number of instructions into a large one so that it can be executed as only one instruction. The compiler is the one responsible for packing the data, resolving any hazards and statically scheduling the instructions. This reduces all the hardware needed compared to a dynamic scheduling approach.

Further reading and a more detailed explanation and description on pipelining, superscalar pipelining, dynamic scheduling and VLIW packing, along with advantages and limitations of each one, can be found in [5][6][7][8][9].

2.1.2 SRAM-based programming technology

An FPGA (Field-Programmable Gate Array) is a general-purpose, multi-level, programmable device with a very high logic density, that allows the user to implement logic circuits in a very short period. It consists of an array of unconnected logic blocks that can be connected by some interconnection resources.

The logic blocks are the building blocks for implementing a circuit in the FPGA. They consist of some logic gates, multiplexers, look-up tables, flip-flops or any other logic used for implementing circuits. The structure and content of a logic block is what defines the FPGA architecture.

The interconnection resources are used to make the connections between logic blocks and are also referred to as the interconnect. It consists of wire segments and programmable switches that allow connections of wires to logic blocks and between wire segments. This switches are implemented using pass-transistors, transmission gates, or multiplexers. The structure and content of the interconnect is what defines the FPGA routing architecture.

All the switches in the interconnect of an FPGA, as well as some inputs in the logic blocks are controlled by programming elements. These can be implemented with fuses, anti-fuses, EPROM or EEPROM transistors, or static RAM (SRAM) cells. The implementation used is what defines the FPGA programming technology. This thesis will focus on SRAM-based programming technology.

The SRAM programming technology is the one most commonly used by FPGA companies. The programming elements in these FPGAs are SRAM cells, which are distributed among the logic they control. These cells are referred to as configuration memory cells.

The RAM cells bits may be loaded into the FPGA in two ways. The first one is in a serial mode, where all the configuration cells in the FPGA are arranged as a very large shift register and bits are shifted in one by one. The second is by indexing each RAM cell as an element of an array and accessing it through an address. A third approach is to use a combination of both serial and parallel. The process of loading a program into the FPGA is called *configuration*.

Although the chip area required by each SRAM cell is larger than the other implementations because of the number of transistors needed, the advantage of it is that it can be configured very quickly. Other advantages of this technology are that it has low power consumption and can be built using a CMOS process.

A more detailed introduction to FPGA architectures, routing architectures and programming technologies; along with characteristics, advantages and limitations of each one can be found in [10]. Also, a more detailed discussion on SRAM-based programming technology can be found in [11]. A description of the architectural design, circuit design and layout issues of an SRAM-based FPGA can be found in [12] and [13].

One variation of an SRAM-based FPGA that some researchers have looked into is a DPGA (Dynamically Programmable Gate Array) [14][15] also known as Multiple-Context FPGA. This architecture can have more than one configuration cell multiplexed for each programmable element. The cells are arranged in a manner that they provide other new sets of configuration cells, which are known as a *configuration planes* or *contexts*. One configuration plane will be active at a time by selecting it through the multiplexors, hence defining the current configuration of the FPGA. This allows a quick context switch between configurations in the FPGA.

DPGAs also provide a desired characteristic for reconfigurable computing which is form of dynamic reconfiguration. At the same time that one configuration is active and executing, other configurations can be loaded or modified independently in the background without affecting the rest of the configurations. This can help reduce configuration overhead. The design and implementation of an FPGA with these characteristics that also allows data sharing among contexts is described in [16].

The reprogramming capability of the SRAM-based FPGAs is what motivates their use in reconfigurable computing. By using this technology, we can build a compute engine in hardware to do some specialized computations. The goal is to have a shorter processing time with the hardware implementation than by executing a sequence of instructions on a general purpose engine. In general, by coupling a general purpose processor with reconfigurable hardware, one could take advantage of the capabilities and features of both. While the processor takes care of all the general-purpose computation, the reconfigurable hardware acts as a specialized coprocessor that takes care of specialized applications.

2.1.3 Coupling Levels

There are several ways in which an FPGA[1] can be coupled to a CPU[2]. They are classified here according to the degree of integration and the communication interface.

1. FPGA will be used in the rest of the thesis to refer to any form of reconfigurable hardware.
2. CPU will be used to refer to a general purpose processor.

STAND-ALONE PROCESSOR.

Systems in this category are the most loosely coupled, where the FPGA acts as a stand-alone processor. This integration requires the FPGA to communicate with the CPU through an I/O interface. Because the communication though I/O interfaces is relatively slow, this integration is useful only on systems where the communication occurs with a very low frequency.

ATTACHED PROCESSOR.

In this category, the FPGA acts as an attached reconfigurable processing unit. It behaves as an additional processor in a multi-processor system. The communication between the FPGA and the CPU is done over a bus in the same way processors communicate in a multi-processor system.

COPROCESSOR

The FPGA can also be used as a coprocessor that aids the processor with certain computations. This allows the FPGA to do big amounts of computations in parallel with the CPU, as well as getting access to its memory resources.

RECONFIGURABLE FUNTIONAL UNIT

Finally, the FPGA can be integrated into the CPU as a Reconfigurable Functional Unit (RFU). This unit can be located inside the processor pipeline in parallel with the existing functional units of the CPU and has access to the processor’s register file. This allows dynamic addition of application-specific instructions to the existing instruction set.

A more extensive survey on reconfigurable systems can be found in [17].

2.2 Software technologies for reconfigurable processors

Research in the area of reconfigurable computing has shown the performance benefits of using this type of architecture for doing computations. However, there is also a need of a software environment that lets a programmer use the reconfigurable resources without requiring much knowledge of the underlying configurable hardware.

Typical programming of a general purpose processor, does not require complete and detailed knowledge of the hardware. The programmer usually uses a High-level Programming Language (HPL), such as C/C++, Java or any other, and a compiler takes care of the translation and generation of the correct machine code for the architecture. These compilers are specialized tools that take advantage of the CPU architecture they are designed to target.

In the same way compilers assist in the use of a processor, CAD tools are needed to use programmable devices. To use an FPGA, a circuit is designed using a Hardware Description Language (HDL) such as VHDL or Verilog, and a CAD tool is responsible of translating or synthesizing the circuit for the FPGA architecture. There is also no need for the user to know the details of the underlying hardware. The whole CAD flow that these tools follow involves steps such as technology mapping, placement and routing, which are very complex and out of the scope of this thesis.

2.2.1 Programming models

There is a wide range of ways one can program a reconfigurable system. On one side there is a manual implementation of a circuit and adaptation to the processor as well as the software running on it. In this approach, the knowledge required of both hardware and software is high. On the other end, there is a concept of an automatic compilation that detects pieces of code suitable for implemention in a custom circuit, which makes the adaptation transparent to a programmer who should only use a standard HPL.

MANUAL CIRCUIT IMPLEMENTATION

This model for programming is the one that requires the most knowledge about the reconfigurable system and could be the one that requires the most design time. The programmer is responsible for designing a circuit that will do some task, and for programming the processor to use that circuit. The circuit is designed using an HDL, and the program using any common HPL.

CIRCUIT LIBRARIES

Circuit libraries are pre-synthesized and tested circuit modules that can be distributed by manufacturers. The programmer will only need to know how to use the library and how to interface it with the software. Each circuit in the library can be used by simply instantiating it in the software program. The advantage of this model is that the circuits have been previously optimized for its task by someone with plenty of knowledge of the hardware.

CIRCUIT GENERATORS

Circuit generator are similar to circuit libraries. The difference is that generators are parameterized non-static structures. This means that the programmer can specify some characteristics, such as size or bit-width of the circuits as arguments to it, and can be reused in different circumstances. Generators can be easily incorporated to a standard programming language.

ASSISTED COMPILATION

Assisted compilation involves the use of an extended HPL. By adding new statements to a programming language such as C, one can generate both the hardware and the software descriptions for the reconfigurable processor with the same language. These extensions involve statements that explicitly tell the compiler when to generate a circuit for the FPGA, and when to generate instructions for the processor. The programmer here needs to know only one language, but needs to help the compiler-synthesizer in taking decisions.

Tools in this category include Transmogrifier C [18] and Napa C [19].

AUTOMATIC COMPILATION

Finally, one of the major goals in reconfigurable computing is to adapt the hardware into the high level programming abstraction. Automatic compilation systems should be able to automatically analyze and detect which parts of an HPL abstraction of a program are suitable for implementing in hardware. Researchers have worked on this issue and have identified software structures of programs that can be ported to a hardware implementation of them. This is exactly what a synthesizer does, so the problem is not the generation of the circuit, but the decision of which of two implementations is better for each application. One way of achieving this is by identifying and matching patterns in the application that can lead to better performance as is done by CPR [20].

2.2.2 Compilation techniques

One great advantage of reconfigurable hardware is the capability of executing tasks in parallel by constructing custom circuits that exploit loop-level parallelism. In general, programs spend 90% of the time inside loops, this means there is a big potential for optimization in them. By generating a custom parallel pipeline for a loop, we can eliminate data and name dependencies in the same way modern compilers do. Also, compiler technology may be used for exposing parallelism in loops with techniques such as loop unrolling, software pipelining and trace scheduling, which will be briefly described next. Other compilation techniques for reconfigurable processors that will take advantage of these previous three techniques are partial evaluation and pipeline vectorization, which will also be described.

LOOP UNROLLING

Loop unrolling is a technique in which a replication of the code inside a loop is done in order to reduce the number of iterations of that loop. This reduces the loop overhead caused by instructions that update the loop control variables (i.e. increments and tests) in each iteration. Loop unrolling also exposes instructions for parallel execution since the statements in the loop can be executed at the same time if they are independent.

SOFTWARE PIPELINING

Software pipelining is another technique used for parallelizing loops. It does the same thing as Tomasulo’s algorithm, but in software. The idea is to reorganize the instructions between loop iterations, so that instructions need not to wait for data dependencies to be resolved. A new loop is generated with instructions from different iterations of the original loop. Also start-up code and finish-up code needs to be created to keep the functionality of the original loop.

TRACE SCHEDULING

Trace scheduling is also used to generate parallelism by combining instruction sequences along the most commonly executed path. This technique is focused on conditional branches instead of loop branches and is based on speculative execution, so speed-up will only be seen if predictions made are correct. It is composed of two parts: trace selection and trace compaction. Trace selection will try to select a sequence of instructions with a high probability of being executed and put them together into a smaller set of instructions. Trace compaction will pack this new set into a small number of wide instructions and try to schedule them as early as possible. A review of this technique for reconfigurable computing can be found in [21].

PARTIAL EVALUATION

When a sequence of logical and arithmetic instructions in a program depend on a few input variables, there is an opportunity for making an optimized circuit for making the computation. Partial evaluation is a program transformation technique based on expressions with known input values. If some inputs to the circuit are constants, their value could be propagated at compile time along the circuit through one or more gates, and only the datapath of the circuit that depends on inputs that are not known needs to be mapped to hardware. This technique is called Partial Evaluation [22].

PIPELINE VECTORIZATION

Pipeline vectorization is a technique based on vectorizing compilers [23]. The idea is to create application specific pipeline structures for loops in a program. The candidates here are selected based on the regularity of iterative computations that perform similar operations on large sets of data. The resulting circuit will behave as a vector coprocessor and loops in the program are replaced by a single vector instruction.

Vector processors [24] operate on linear arrays of data, called vectors, and greatly exploit loop-level parallelism. In general a vector processor will load two source vectors from memory, execute the vector instruction on the data and store back the resulting vector into memory. This architecture allows very deep pipelines and eliminates many of the data and control hazards in a program. Newer vector architectures perform operations on registers rather than memory. More on vector architectures can be found in [8][9].

OTHER COMPILER OPTIMIZATIONS

Other compiler optimizations exist that allow a much simpler circuit implementation. Among them are, Strength Reduction and Register Renaming. Strength reduction replaces expensive operations, such as multiplications and divisions, by less expensive ones, such as additions and subtractions. An FPGA will execute less expensive instructions faster than expensive ones, in the same way a CPU does. Register renaming increases the flexibility to parallelize loops by eliminating data dependencies between instructions that use the same register, in and among iterations. This is done by using of more registers, which can be easily implemented in an FPGA. A broader description of these and other compiler technologies can be found in [25] and [26].

2.2.3 Configuration management

Since hardware is a physical device, there is not unlimited resources for every application. When handling multiple configurations in a program, there is a need to dynamically load them to the FPGA and eventually replace them in a manner that results in the least configuration overhead possible. To achieve this goal on multi-context architectures, configuration management issues for loading, replacing and compressing configurations are used and will be discussed next. A complete study on configuration management for other architectures may be found in [27].

PRE-LOADING CONFIGURATIONS

When a configuration is needed for the execution of a reconfigurable instruction, the processor should configure the FPGA first before using the configuration. The processor will then have to wait while the configuration is loaded on the FPGA and time will be wasted resulting in poor performance. The compiler can statically analyze the source code to determine which configuration will be used ahead of time and pre-load the required configuration in parallel with the processor execution of computations. By the time the processor needs to execute the reconfigurable instruction, the configuration will be ready and no time will be wasted. The challenge is to pre-determine which configuration will be needed next. A study on configuration prefetching is presented in [28]. Reconfigurable processors should support pre-loading configurations.

LRU REPLACEMENT POLICY

When the available hardware is not enough to hold all configurations, there is a need to replace configurations with the ones that are needed next. Least Recently Used (LRU) is a widely used algorithm used in operating systems for virtual memory management. This approach selects the item to be replaced according to the time each one was used. The configuration that has not been used for the longest period will be replaced. Reconfigurable processors should support LRU replacement.

CREDIT-BASED REPLACEMENT POLICY

An extension to LRU is the credit-based replacement policy. Besides time, this policy takes into account the size of the configuration to be replaced. This approach takes advantage of the fact that smaller configurations take less time to load than big ones. Credit will be assigned to each configuration according to its size and credits will be decremented with time. The configuration with the least credit will be selected for replacement.

CONFIGURATION COMPRESSION

Compressing configurations further reduce the time to load a single configuration into the FPGA. A configuration compression algorithm proposed by Hauck et. al. is described in [29]. It achieves compression ratios of a factor of 4 and relies on hardware wildcard registers as implemented in the Xilinx XC6200 series FPGAs [30]. Li [31] further extends the algorithm for a higher compression ratio of up to a factor of 7, by finding irrelevant values in the configuration stream.

2.3 Survey of reconfigurable processors

Research groups and companies worldwide are currently working on combining the flexibility of general-purpose processors with the efficiency of custom application-specific hardware implemented in reconfigurable logic. These systems are in general known as Field-Programmable Custom Computing Machines (FCCMs). Major FCCM projects and systems will be briefly mentioned in this section.

2.3.1 University projects

PIPERENCH

PipeRench [32][33][34] was developed at Carnegie Mellon University. It is a reconfigurable fabric that belongs to the family of Reconfigurable Attached Processors since it is interfaced with a processor through a PCI bus. PipeRench consists of an interconnected network of processing elements organized as pipeline stages. Each processing element consists of registers and ALUs. An intermediate language is used to generate the fabric's configurations.

CHIMAERA

Chimaera [35] was developed at Northwestern University. It is a reconfigurable array that is integrated to a processor as a Reconfigurable Functional Unit. The array has access to shadow registers and consists only of configurable combinational logic.

GARP

Garp [36][37] was developed at University of California Berkeley. It belongs to the family of Reconfigurable Coprocessors as it integrates a reconfigurable array that has access to the processor's memory hierarchy. The reconfigurable array may be partially reconfigured as it is organized in rows. Configuration bits are included and linked as constants with ordinary C compiled programs.

MORPHOSYS

MorphoSys [38] was developed at University of California Irvine. It is a Reconfigurable Cell Array architecture that includes context memory and belongs to the family of Reconfigurable Coprocessors. A DMA controller transfers data to the context memory, and to and from a frame buffer that holds the data the array will operate on.

REMARC

Remarc [39] was developed at Stanford University. It is a Reconfigurable Coprocessor with 64 programmable units, that targets multimedia applications. Each 16-bit unit has an entry instruction RAM, ALUs, data RAM, instruction and several other registers. The reconfigurable array operates on the coprocessor data registers and a control unit transfers data between these registers and the processor.

PRISC

Prisc [40] was developed at Harvard University. It integrates combinational reconfigurable logic as Reconfigurable Functional Units with two inputs and one output. The compiler analyzes opportunities the generated code and identifies sets of sequential instructions to execute on the PFU.

SONIC

Sonic [41] is a project from the University of London. It is designed to exploit parallelism in video image processing algorithms. It consists of a set of processing elements, called PIPEs, interconected by a bus. It belongs to the family of Reconfigurable Attached Processors since it is interfaced with a processor through a PCI bus.

2.3.2 Commercial systems

CHAMELEON SYSTEMS

The Chameleon CS2000 family [42] combines a 32-bit embedded processor with a 32-bit reconfigurable processing fabric. The two are linked by a 128-bit, split-transaction bus. The Chameleon compiler generates the final application from a C compiled object code linked with the fabric’s previously generated bitstream configurations.

TRISCEND

The Triscend E5 family [43] is composed of configurable system-on-chp devices that combine an embedded microcontroller, a block of SRAM, a system bus and configurable logic interconnected inside a single chip. The configurable logic may be used through the circuit generator modules included in a library.

NATIONAL SEMICONDUCTOR

The National Adaptive Processing Architecture (NAPA) [44] combines an adaptive logic processor with a fixed instruction processor, memory and an interconnection network interface. The Napa C compiler [19] is an assisted compiler that requires user intervention to explicitly target the desired fabric.

ANNAPOLIS

The Annapolis Wildfire family [45] are Reconfigurable Attached multi-processor boards composed of an array of Xilinx FPGAs that are conected to a processor through a VME or PCI bus. The Wildcard is another Reconfigrable Attached Processor as it is PCMCIA sized card with an FPGA as processing element, memory and I/O connectors. They are programmed using standard C and VHDL tools.

2.4 Summary

This chapter defined what is considered to be a reconfigurable processor. It presented an overview of current reconfigurable processor architectures, as well as current compilation technologies that support them. The first part covered how performance is measured and current techniques used to speedup processors. It also talked about the opportunity of using programmable logic for doing computations, in particular, SRAM-based programming technology; and described various configurations for coupling reconfigurable logic with processors. The second part covered the software and talked about compiler technologies for reconfigurable processors. It described current programming models and compilation techniques that allow circuit implementation of instructions, as well as programmable logic configuration management issues. Finally, the third part presented an brief description of related university projects and currently available industry reconfigurable processor products.

CHAPTER 3

OneChip Architecture

In the last chapter, current research issues in reconfigurable computing were presented. This chapter will present the OneChip reconfigurable processor. The description focuses on extensions done to OneChip, in particular to support compiler technologies.

3.1 Architecture specification

OneChip is a reconfigurable processor that integrates a Reconfigurable Functional Unit (RFU) into a Reduced Instruction Set Computer (RISC) processor's pipeline. It is a superscalar architecture that allows out-of-order issue and dynamic reconfiguration. It also provides support for pre-loading configurations and for Least Recently Used (LRU) configuration management.

The processor's main features are listed below:

- MIPS-like RISC architecture - simple instruction encoding and pipelining.
- Superscalar pipeline - allows multiple instructions to issue per cycle.
- Dynamic scheduling - allows out-of-order issue and completion.
- Dynamic reconfiguration - can be reconfigured at run-time.
- RFU integration - programmable logic in the processor's pipeline.
- Configuration pre-loading support - allows loading configurations ahead of time.
- Configuration compression support - reduces configuration size.

- LRU configuration management support - reduces number of reconfigurations.

3.2 Previous work

Previous work on OneChip was done by Ralph D. Wittig [1][2] and Jeffrey A. Jacob [3][4]. The first OneChip implementation was a scalar architecture derived from the MIPS-like RISC CPU described by Patterson and Hennessy in [5]. It combined the processor core with reconfigurable logic resources. The approach taken was to interface the reconfigurable logic by adding it in parallel to the already existing basic functional units inside the processor pipeline. The reconfigurable logic acts as an RFU[1] that will be responsible for many application specific functions. One of the most important features in the architecture is its capability of handling variable, multiple clock cycle latency on RFU components.

The second implementation of OneChip was based on the DLX standard RISC processor described by Hennessy & Patterson in [6]. The extensions done to the OneChip architecture included superscalar and dynamic execution, RFU instruction specification, configuration of the RFU, and a memory interface. Since the RFU has direct access to memory and out-of-order execution and completion are allowed, there is a possibility for memory inconsistency. To solve this problem, a memory consistency scheme was developed.

3.3 Instruction Scheduling

Dynamic scheduling is supported by OneChip. The proposed processor pipeline and solutions to support dynamic scheduling will be described next.

1. RFU (Reconfigurable Functional Unit) was described in Chapter 2 and will refer to the FPGA logic and the FPGA controller together through the rest of this thesis.

3.3.1 Processor pipeline

The proposed OneChip's pipeline consists of five stages. These are: Instruction Fetch (IF), Instruction Decode (ID), Execute (EX), Memory Access (MEM) and Writeback (WB). A diagram of the pipeline is showed in Figure 3-1. The RFU is integrated in parallel with the EX and MEM stages. It performs computations as the EX stage does and has direct access to memory as the MEM stage does. The RFU contains structures such as the Memory Interface, an Instruction Buffer, a Reconfiguration Bits Table (RBT) and Reservation Stations, which will be described with detail later in this chapter.

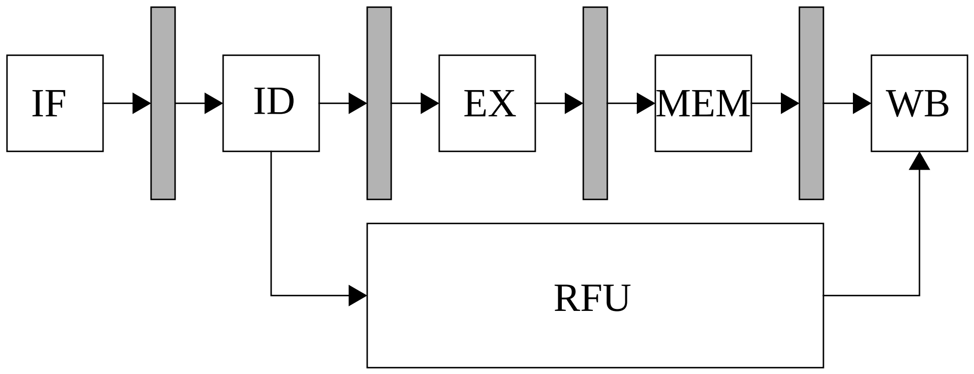

Figure 3-1. **OneChip's pipeline**

3.3.2 Instruction level parallelism

OneChip is capable of executing multiple instructions in parallel. The EX stage consists of multiple functional units of different types, such as integer units, floating point units and a reconfigurable unit. Due to the flexibility of the reconfigurable unit to implement a custom instruction, a programmer or a compiler can generate a configuration for the reconfigurable unit to be internally pipelined, parallelized or both.

3.3.3 Dynamic scheduling

Dynamic scheduling is implemented in OneChip by using Tomasulo's algorithm [6]. Data dependencies between RFU and CPU instructions are handled using RFU Reservation Stations. The reservation station layout proposed by Jacob [3] supports only two-operand RFU instructions. To add more flexibility for a programmer and a compiler to implement various kernel applications, such as filters, it is desirable to extend the reservation station layout to add for support three-operand RFU instructions. The enhanced RFU Reservation Stations are described next.

3.3.4 Reservation stations

The layout of an RFU Reservation Station (RFU-RS) is shown in Figure 3-2. The RFU-RS keeps track of 12 fields. *Tag* refers to the instruction identifier assigned by the CPU to each instruction. The *function* field refers to the configuration to be used for the RFU instruction. The FU_{src1}, FU_{src2} and FU_{dst} fields refer to the functional units that will produce the RFU sources (source 1 and source 2) and destination addresses. When the SRC1 address becomes available, it is stored in the ADD_{src1} field and FU_{src1} is cleared to zero. The same is done with ADD_{src2} and ADD_{dst} to indicate that the addresses are available in the corresponding field. The source and destination Blocks Sizes (BS) are kept in BS_{src1}, BS_{src2} and BS_{dst} respectively. The 1-bit *busy* flag indicates that the reservation station is busy.

tag	function	$\mathrm{FU}_{\mathrm{src1}}$	$\mathrm{FU}_{\mathrm{src2}}$	$\mathrm{FU}_{\mathrm{dst}}$	$\mathrm{ADD}_{\mathrm{src1}}$	$\mathrm{ADD}_{\mathrm{src2}}$	$\mathrm{ADD}_{\mathrm{dst}}$	$\mathrm{BS}_{\mathrm{src1}}$	$\mathrm{BS}_{\mathrm{src2}}$	$\mathrm{BS}_{\mathrm{dst}}$	busy

Figure 3-2. RFU Reservation Station Layout

3.4 Run-time reconfiguration

A reconfigurable processor has the characteristic of being able to change its functionality at run-time according to the current application. This characteristic is called Run-time Reconfiguration and it is implemented in OneChip using a Dynamically Programmable Gate Array (DPGA) or Multi-Context FPGA. The configuration architecture, configuration compression and management support, and the reconfiguration bits table are described in this section.

3.4.1 Configuration architecture

The RFU in OneChip contains an FPGA[1] and an FPGA Controller as shown in Figure 3-3. The FPGA is capable of holding more than one configuration for the programmable logic. These configurations are stored in the Context Memory. It is also capable of rapidly switching

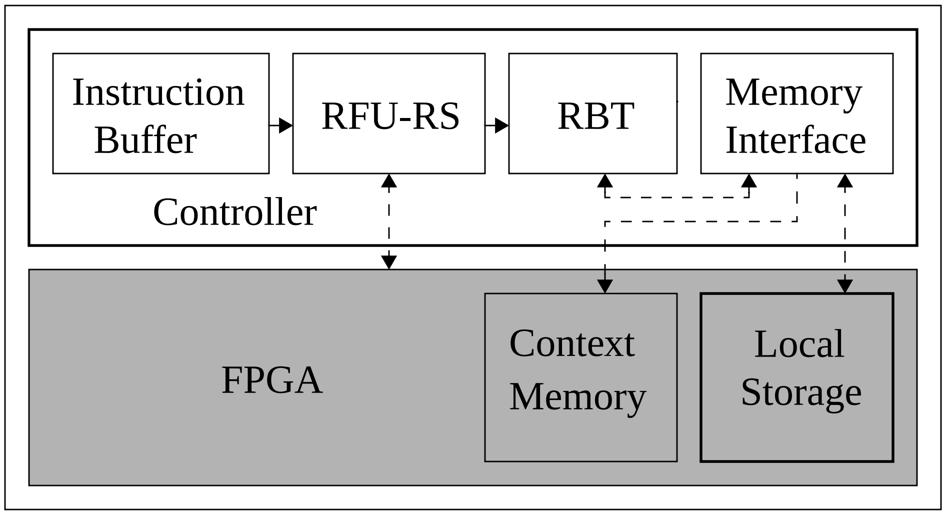

Figure 3-3. RFU Architecture

1. It is assumed that an SRAM-based Multiple-Context Field Programmable Gate Arrays is employed, hence FPGA will be used to refer to it through the rest of this thesis.

among configurations as described in Chapter 2. Each context of the FPGA is configured independently from the others and acts as a cache for configurations. Only one context may be active at any given time. This architecture reduces configuration overhead as it allows configuration loading into inactive contexts to take place while the processor is busy doing computations on any functional unit, including the RFU with the corresponding active configuration.

Instructions that target the RFU in OneChip are forwarded to the FPGA Controller, which contains the reservation stations described in Section 3.3.4 and a Reconfiguration Bits Table (RBT), which will be described in Section 3.4.4. The FPGA Controller is responsible for programming the FPGA, the context switching and selecting configurations to be replaced when necessary. The FPGA Controller also contains a buffer for instructions and the memory interface. The RBT, acts as the configuration manager that will keep track of where the FPGA configurations are located.

The memory interface in the FPGA Controller consists of a DMA controller that is responsible for transfering configurations from memory into the context memory according to the values in the RBT. It also transfers the data that the FPGA will operate on into the local storage. The local storage may be considered as the FPGA data cache memory.

3.4.2 Configuration compression support

The previous OneChip model [3] did not support any configuration compression. We feel that OneChip should be enhanced to support configuration compression and reduce the overhead involved in configuring the FPGA. This can be accomplished by using mask registers, as is done in the Xilinx XC6200 with wildcard registers [30]. The idea is to allow multiple cells to be written simultaneously by providing a single address and a mask of “don’t care” bits. The multiple decoded addresses will enable multiple configuration cells to be loaded at the same time. This approach will take advantage of regularity in configurations.

An example of four configuration decompressions is shown in Table 3-1. In the first column,

Address	0010 0100	0010 0100	0010 0100	0010 0100
Mask	0000 0000	0001 0000	0010 0001	0000 0111
Decoded Addresses	0010 0100	0010 0100	0000 0100	0010 0000
		0011 0100	0000 0101	0010 0001
			0010 0100	0010 0010
			0010 0101	0010 0011
				0010 0100
				0010 0101
				0010 0110
				0010 0111

Table 3-1. **Configuration Decompression**

none of the bits are masked and only one address is decoded. On the second column, bit number four[1] is set to 1 and is masking the fourth bit of the address, producing two different decoded addresses where the fourth bit of each one is a 0 and a 1 respectively. Similarly, four and eight addresses are decoded on the third and fourth columns by setting to 1 two and three masking bits respectively.

3.4.3 Configuration management support

Although the FPGA can hold multiple configurations, there is a hardware limit on the number of configurations it can hold. Therefore, a mechanism for swapping configurations in and out of the FPGA is required. The algorithm selected for configuration replacement is the Least Recently Used (LRU) algorithm. The previous OneChip model [3] did not support any configuration management.

1. Bits are numbered 7-0 from left to right.

LRU is implemented in OneChip by using a table of configuration reference bits. The approach is similar to the Additional-Reference-Bits Algorithm described by Silberschatz & Galvin in [46]. A fixed-width shift register is used to keep track of each loaded configuration's history. On every context switch, all shift registers are shifted 1 bit to the right. On the high-order bit of each register, a 0 is placed for all inactive configurations and a 1 for the active one. If the shift register contains 00000000, it means that it hasn't been used in a long time. If it contains 10101010, it means that it has been used every other context switch. A configuration with a history register value of 01010000 has been used more recently than another with value of 00101010, and this later one was used more recently than one with a value of 00000100. Therefore, the configuration that should be selected for replacement is the one that has the smallest value in the history register. Notice that the overall behavior of these registers is to keep track of the location of configurations in a queue, where a recently used configuration will come to the front and the last one will be the one to be replaced.

3.4.4 Reconfiguration bits table (RBT)

The Reconfiguration Bits Table (RBT) is shown in Table 3-2. The addition to the RBT proposed by Jacob is the History field, which supports configuration management.

FPGA function	Address	Active	Loaded	Context ID	History
0000	0x800000	No	Yes	2	0101
0001	0x80B000	No	Yes	0	0010
0010	0x824000	No	No	-	-
0011	0x81C000	Yes	Yes	1	1000
0100	0x810000	No	Yes	3	0000
0101	0x826000	No	No	-	-

Table 3-2. **Reconfiguration Bits Table**

In the table, the *FPGA function* field is the configuration identifier assigned by the compiler. It is different for each configuration. Since this is a 4-bit field, 16 different configurations (functions) can be used. The *Address* field holds the memory address where the configuration bits are stored. The *Active* flag indicates whether the configuration is active on the programmable logic. The *Loaded* Flag indicates whether the configuration is loaded in any of the DPGA contexts, and if it is, the context number will be stored in the *Context ID* field. The *History* field is used for selecting configurations to be replaced as described in Section 3.4.3.

3.5 Instruction specification

OneChip is designed to obtain speedup mainly from memory streaming applications in the same way vector coprocessors do. In general, RFU instructions take a block of data that is stored in memory, perform a custom operation on the data and store it back to memory. In the following sections, the format for reconfigurable instructions and configuration instructions will be described.

3.5.1 Reconfigurable Instruction formats

There are two versions for RFU instructions: two-operand format and three-operand format. Previously, only two-operand instructions were supported by OneChip [3]. By adding an extra three-operand instruction, there is more flexibility for applications. The instruction format for a two-operand instruction is shown in Figure 3-4. The *opcode* is the identification field for this instruction. The *function* field corresponds to the desired FPGA function to be used when executing this instruction; the corresponding configuration will be obtained from the RBT. The R_{SRC} and R_{DST} are the registers that contain the location for the source and destination blocks respectively. The address of the first element of the data that the instruction will operate on should be in these registers. The *src size* and *dst size* fields hold the source and destination block sizes respectively. These are encoded as described in [4]. In this instruction version, source and destination block sizes can be different. The *misc* field is left for future use.

opcode (16 bits)	misc. (12 bits)	function (4 bits)	R_{SRC} (8 bits)	R_{DST} (8 bits)	src size (8 bits)	dst size (8 bits)
63		32	31			0

Figure 3-4. RFU Two-operand Instruction Format

The format for a three-operand instruction is shown in Figure 3-5. In this instruction, one of the block sizes has been removed and replaced by another source register R_{SRC2}. This allows the RFU to get source data from two different memory locations, which need not to be continuous. The *blk size* field will refer to the size for the two source blocks and the destination block. In this instruction version, all three blocks should be the same size, as opposed to the two-operand version where different block sizes can be specified.

opcode (16 bits)	misc. (12 bits)	function (4 bits)	R_{SRC1} (8 bits)	R_{SRC2} (8 bits)	R_{DST} (8 bits)	blk size (8 bits)
63		32	31			0

Figure 3-5. RFU Three-operand Instruction Format

3.5.2 Configuration instruction formats

In OneChip, there are two configuration instructions. One of them is the Configure Address instruction, which is used for assigning memory addresses in the RBT. Its format is shown in Figure 3-6. The other configuration instruction is the pre-load instruction, which is used for pre-fetching instructions into the FPGA and reducing configuration overhead. Its format is presented in Figure 3-7.

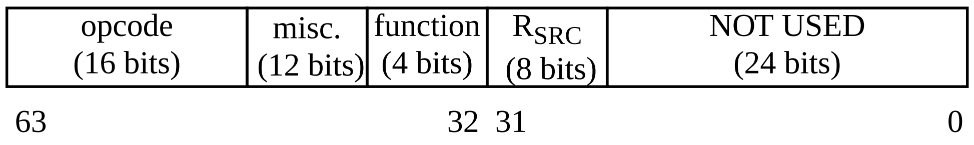

Figure 3-6. **RFU Configure Address Instruction Format**

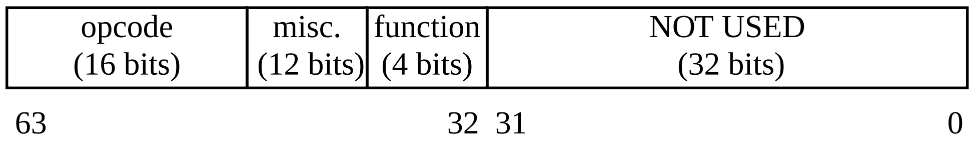

Figure 3-7. **RFU Pre-load Instruction Format**

3.6 Memory controller

OneChip allows superscalar dynamic scheduling, hence instructions with different latencies may be executed in parallel. When data hazards exist between instructions that use registers, the dependencies can be eliminated by stalling instructions with unresolved data. Also, the appropriate instructions may be scheduled by the compiler to eliminate the dependencies.

The RFU in OneChip has direct access to memory and is also allowed to execute in parallel with the CPU. When there are no data dependencies between the RFU and the CPU, the system will act as a multiprocessor system, providing speed up. However, when data dependencies exist between them, there is a potential for memory inconsistency which must be prevented. Several data hazards may occur in the system and will be described in the following section.

Due to the unpredictability of reconfigurable instructions, it is difficult for the compiler to schedule the appropriate instructions to eliminate data hazards. Thus, a memory consistency scheme is implemented in the processor, which will be described in Section 3.6.2.

3.6.1 Memory hazards

Memory data hazards are classified in three types [6].

- RAW hazards (Read after Write). A read of some data comes after a write to the same data, but due to reordering, the reading instruction may read the data before the preceding one writes it. The reading instruction will get a wrong old value. The relation between these two instructions is known as true dependence.
- WAR hazards (Write after Read). A write comes after a read, but due to reordering, the writing instruction modifies the value before the preceding one reads it. The reading instruction will not get the old value it should. The relationship between these instructions is known as anti-dependence.
- WAW hazards (Write after Write). A write comes after another write, but due to reordering, the value of the second write is modified by the first one. The value that will remain will be a wrong one. The relationship between these instructions is known as output dependence.

In OneChip each of these three types of hazards may occur twice. Once when CPU instructions follow RFU instructions, and also when RFU instructions follow CPU instructions. The six hazards will be discussed next.

3.6.2 Memory consistency scheme

The six possible hazards that OneChip may experience along with the actions taken to prevent them, are listed in Table 3-3.

Hazard Number	Hazard Type	Actions Taken
1	RFU read after CPU write	1. Flush RFU source addresses from CPU cache when RFU instruction issues. 2. Prevent RFU reads while pending CPU store instructions are outstanding.
2	CPU read after RFU write	3. Invalidate RFU destination addresses in CPU cache when RFU instruction issues. 4. Prevent CPU reads from RFU destination addresses until RFU writes its destination block.
3	RFU write after CPU read	5. Prevent RFU writes while pending CPU load instructions are outstanding.
4	CPU write after RFU read	6. Prevent CPU writes to RFU source addresses until RFU reads its source block.
5	RFU write after CPU write	7. Prevent RFU writes while pending CPU store instructions are outstanding.
6	CPU write after RFU write	8. Prevent CPU writes to RFU destination addresses until RFU writes its destination block.

Table 3-3. Memory Consistency Scheme

Hazards number 1 and 2 are RAW hazards. On hazard 1, the processor writes data to memory that the RFU needs. The solution is to flush all RFU source addresses from the CPU cache when the RFU instruction issues, and wait for all CPU store instructions to complete before performing any RFU read. On hazard 2, the CPU needs some data that the RFU will produce. The solution is to invalidate all RFU destination addresses in the CPU cache when the RFU instruction issues, and wait for the RFU to write its destination block before performing any CPU read. It is desirable to stall instructions only if there is a data dependence. Otherwise, they should be allowed to continue. This leads to the need of locking certain blocks of memory, which is accomplished with the use of a Block Lock Table (BLT).

Hazards 3 and 4 are WAR hazards. On hazard 3, the RFU is writing to memory after the issue of a CPU read. If the read is delayed and the RFU overwrites the data before being read, the CPU will load a wrong value. The RFU then needs to wait for the CPU to read the data before updating it with a new value. This is accomplished by preventing any RFU writes if there are any pending CPU loads. On hazard 4, the CPU is writing data before the preceding RFU instruction reads it. The CPU needs to wait for the RFU to read its source blocks before writing the data. The BLT is required also to lock source blocks.

Hazards 5 and 6 are WAW hazards. For both of these hazards, the latest instruction should be the last one to update the data in memory, so that the correct value remains in it. For hazard 5, the RFU should be prevented from writing to memory if there are outstanding stores. For hazard 6, the CPU should be prevented from writing to any FPGA destination address.

3.6.3 Memory consistency for multiple FPGAs

The previous memory consistency scheme works for an architecture with a single FPGA. It is desirable to have multiple FPGAs and allow them to execute concurrently. Since each FPGA may access data that another FPGA is also accessing, three new possible hazards may occur. It is needed to extend the memory consistency scheme to support multiple FPGAs. This scheme is shown in Table 3-4. When multiple FPGAs are introduced in the processor, they must share the same FPGA Controller that will be responsible for controlling the multiple FPGAs. The RFU will then have one FPGA Controller and more than one FPGA with its own context memory and local storage.

Assume there are two FPGAs in the RFU and that two consecutive RFU operations *A* and *B* will get executed on *FPGA A* and *FPGA B* respectively. On hazard 7, which is a RAW hazard, the RFU operation on FPGA B is reading data form a block that the FPGA A has not finished writing. The read will get a wrong value. The RFU needs to wait for the corresponding block of memory to be written before issuing the RFU read. On hazard 8, which is a WAR hazard,

Hazard Number	Hazard Type	Actions Taken
7	RFU read after RFU write	9. Prevent RFU reads from locked RFU destination addresses.
8	RFU write after RFU read	10.Prevent RFU writes to locked RFU source addresses.
9	RFU write after RFU write	11.Prevent RFU writes to locked RFU destination addresses.

Table 3-4. **Memory Consistency Scheme for Multiple FPGAs**

the RFU operation on FPGA B is writing to a memory location that the FPGA A has not finished reading. The RFU needs to wait for the corresponding block to be read before updating the data. On hazard 9, which is a WAW hazard, the RFU operation on FPGA B is writing to a block that the FPGA B has not finished writing, so the wrong value will remain after the two writes. The RFU needs to wait for the corresponding block to be written before updating the data.

3.6.4 Block lock table (BLT)

OneChip implements the memory consistency scheme described in Section 3.6.2 by using a Block Lock Table (BLT). A BLT contains four fields for each entry and locks memory blocks to prevent undesired accesses. An example of a BLT is shown in Table 3-5. The *Block Address* field is the location in memory of the start of each block. It will be obtained from the source and destination registers of each RFU instruction. The *Block Mask* field will record the size of the memory block to be locked. As described in [4], this mask is encoded in the block size fields of the RFU instructions. For a block size N, the mask will have 1's on the N+1 rightmost bits and 0's on the rest of them. The range of addresses for each block is obtained by adding the block address and the block mask. The range for each block will be [*block address* : *block*

Block Address	Block Mask	RFU Instruction Tag	src/dst
0001 1000 0000 0000	0000 0001 1111 1111	2	src
0010 1000 0000 0000	0000 0001 1111 1111	2	src
0011 1000 0000 0000	0000 0001 1111 1111	2	dst
0110 0011 0000 0000	0000 0000 0111 1111	1	src
0110 0011 1000 0000	0000 0000 0001 1111	1	dst

Table 3-5. **Block Lock Table**

address + *block mask*]. Notice that the Block Mask is not acting as such here. This BLT range check is more general and requires more hardware than the one used in [3], where the 1's in the mask act as "don't care" bits for the address, simplifing the hardware for checking memory ranges. This new approach does not require the compiler to align the data in memory. The *RFU Instruction Tag* identifies which instruction is locking that memory block and the *src/dst* flag indicates weather the memory block is locked as a source or as a destination.

3.7 Summary

This chapter described the architecture of the OneChip reconfigurable processor. The description is focused on the modifications and enhancements done to the previously proposed model. The system was implemented in a software model to simulate its performance. Details on this implementation will be the topic of the next chapter.

CHAPTER 4

Sim-OneChip

In the last chapter, the architecture of the OneChip reconfigurable processor was described. This chapter will present Sim-OneChip, an architecture simulator of the system. It is a software implementation that allows programs to run on it and to simulate the system's performance.

4.1 Architecture simulators

Architecture simulators are tools that reproduce the behavior of computing devices. In general, they are classified into two types of simulators: functional and performance. Functional simulators implement the architecture (i.e. they only simulate the system's external function). Performance simulators implement the micro-architecture (i.e. they model the system internal architecture in detail). Functional simulators are further divided into trace-driven, which read a "trace" or list of instructions captured during a previous execution; and execution-driven simulators, which read programs and actually "run" them by generating the list of instructions on-the-fly.

To create a simulator for the OneChip processor, several already existing architecture simulators were considered for modification. The two most attractive were RSIM and SimpleScalar. RSIM [47] is an execution-driven simulator for ILP-based shared-memory multiprocessors and uniprocessors. It was developed by Vijay S. Pai, Parthasarathy

Ranganathan and Sarita V. Adve at RICE University. Simplescalar [48][49] is a collection of simulation tools that include execution-driven simulators of modern microprocessors. It was developed by Doug Burger and Todd M. Austin at the University of Wisconsin-Madison.

SimpleScalar was the chosen platform for the OneChip architecture simulator. Besides being a complete set of tools, the annotations capability was an attractive feature, since it would allow the addition of new instructions in a very simple manner. Also, the documentation is clear and the code is modular and well commented.

4.2 The SimpleScalar tool set

The SimpleScalar tool set is a suite of simulation tools for modern microprocessors. It consists of compilers, an assembler, a linker, libraries and execution-driven simulators. The simulators are written in C. They take binaries compiled for the SimpleScalar architecture, which is similar to the MIPS-IV architecture, and simulate their execution on one of several RISC processor configurations.

The SimpleScalar architecture is derived from the MIPS ISA architecture. The main difference is that load, store and control transfer instructions do not execute the succeeding instruction, hence there are no architected delay slots. Other differences include support for indexed and auto-increment addressing modes, a single and double precision floating point square root instruction and an extended 64-bit instruction encoding.

The architecture contains 32 general-purpose registers, two result registers, 32 floating-point registers and one condition code register. There are three general formats for encoding instructions, which are shown in Figure 4-1. The register format is used for computational instructions and supports specification of three registers and an 8-bit constant for shift operations. The immediate format replaces one register to support an immediate 16-bit constant. The jump format supports the specification of 24-bit jump targets.

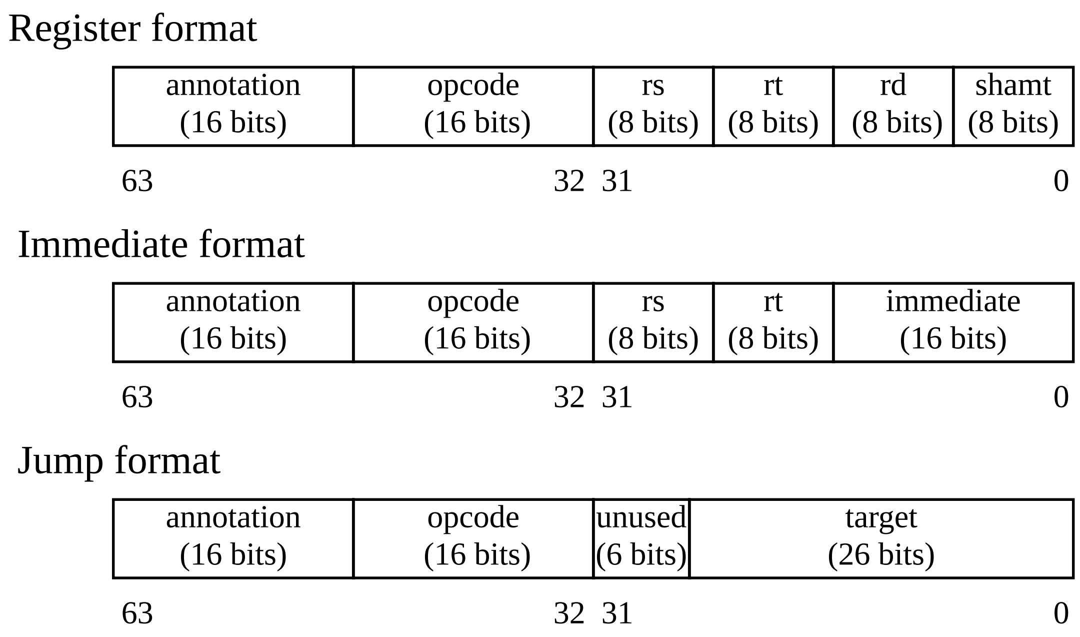

Figure 4-1. SimpleScalar Architecture Instruction Formats

As stated before, a very important feature in SimpleScalar is the 16-bit *annotation* field. It allows instructions to be annotated in the assembly files and is useful for adding new instructions without modifying or recompiling the assembler.

Several simulators are included with SimpleScalar. *Sim-fast* is the least detailed simulator and performs functional simulation only. *Sim-safe* is another functional simulator that further checks for correct alignment and access permissions for each memory reference. *Sim-cache* and *sim-cheetah* are two functional cache simulators. *Sim-profile* produces detailed profiles on the program execution. And lastly, *sim-outorder* is the most complete and detailed simulator. It is a pipelined architecture that supports out-of-order execution.

The architecture of *sim-outorder* consists of six pipeline stages. The *fetch* stage models the machine instruction bandwidth. It fetches instructions from the I-cache line and places them on the dispatch queue. The *dispatch* stage models instruction decoding and register renaming.

It takes instructions from the dispatch queue and puts them in the corresponding reservation stations in the scheduler queue, referred to as the Register Update Unit (RUU). Memory operations are split into two: the effective address computation and the memory access. Memory accesses are placed in a separate load-store queue (LSQ), while the effective address computation is placed in the RUU. The *issue* stage models instruction issue to the functional units. It marks ready instructions from the scheduler queues (RUU and LSQ) for future execution. The *execute* stage models the execution of instructions in available functional units and schedules writeback events on the event queue. The *writeback* stage models the writeback bandwidth and detects mispredictions. Finally, the *commit* stage models in-order retirement of instructions. It moves completed instructions from the event queue and retires them, freeing up resources.

4.3 OneChip implementation

The architecture of OneChip is modeled by *sim-onechip*. It is a simulator derived from *sim-outorder* from the simplescalar tool set. Modifications were done to *sim-outorder* to model OneChip's reservation stations, reconfiguration bits table, block lock table and the reconfigurable unit. The overall functionality of *sim-outorder* was preserved.

4.3.1 Reservation stations

The reservation stations for Sim-OneChip were implemented as a queue. Besides the already existing RUU and LSQ scheduler queues, a third scheduler queue was implemented to hold RFU instructions. This queue is referred to as the Reconfigurable Instructions Queue (RecQ). The dispatch stage detects instructions that target the RFU and places them in the RecQ for future issuing.

The OneChip RFU_RS fields described in the previous chapter are defined in the RUU_station structure in `sim-onechip.c`.

4.3.2 Reconfiguration bits table (RBT)

The RBT is implemented as a linked list where each entry holds the corresponding fields described on the previous chapter. The RBT models the FPGA controller by performing dynamic reconfiguration and configuration management. Functions are provided for assigning configuration addresses, loading configurations and to perform context switching.

The implementation of the RBT is done in `rbt.h` and `rbt.c`. The number of entries in the RBT (RBT_SIZE) and the number of DPGA contexts (RBT_DPGA_SIZE) are specified in `rbt.h`. The most important RBT interfaces are:

rbt_setAddress()	Assigns the configuration address to a function.
rbt_loadConfiguration()	Loads a configuration from memory to the FPGA.
rbt_selectConfiguration()	Select the active function.

4.3.3 Block lock table (BLT)

The BLT is implemented as a linked list. Each entry holds the fields described in the previous chapter for the two sources and the destination memory blocks for each RFU instruction. It ensures the OneChip's memory consistency scheme by modeling the actions taken for each of the hazards presented. By keeping track of the memory locations currently blocked, conflicting instructions are properly stalled.

The implementation of the BLT is done in the files `blt.h` and `blt.c`. The number of entries in the BLT (BLT_SIZE) is specified in the file `blt.h`. The most important BLT interfaces are:

blt_setEntry()	Locks the memory blocks for the specified instruction tag.
blt_clearEntry()	Clears the memory lock for the specified instruction tag.
blt_locked()	Checks for a memory address lock.

4.3.4 Reconfigurable unit

The RFU was included with the rest of the functional units. The new functional unit class was created in ss_fu_class in the file `onechip.h`. The corresponding description string was included in ss_fu2name in `onechip.c`. Also, the RFU was included in the resource pool in the functional unit resource configuration in `sim-onechip.c`. The initial RFU operation and issue latencies definition are irrelevant, since new FPGA configurations will override these values with the corresponding ones when executing each configuration.

4.4 Pipeline description

Sim-OneChip's pipeline, as in *sim-outorder*, consists of six stages: *fetch*, *dispatch*, *issue*, *execute*, *writeback* and *commit*. This section will describe the modifications done to each stage in *sim-outorder* and the places where each of OneChip's structures were included. Sim-OneChip's pipeline is shown in Figure 4-2.

FETCH STAGE

The *fetch* stage remained unmodified and fetches instructions from the I-cache into the dispatch queue.

DISPATCH STAGE

The *dispatch* stage decodes instructions and performs register renaming. It moves instructions from the dispatch queue into the reservation stations in the scheduler queue. Recall that the scheduler queue consists of two queues (RUU and LSQ) plus the newly added RecQ queue for

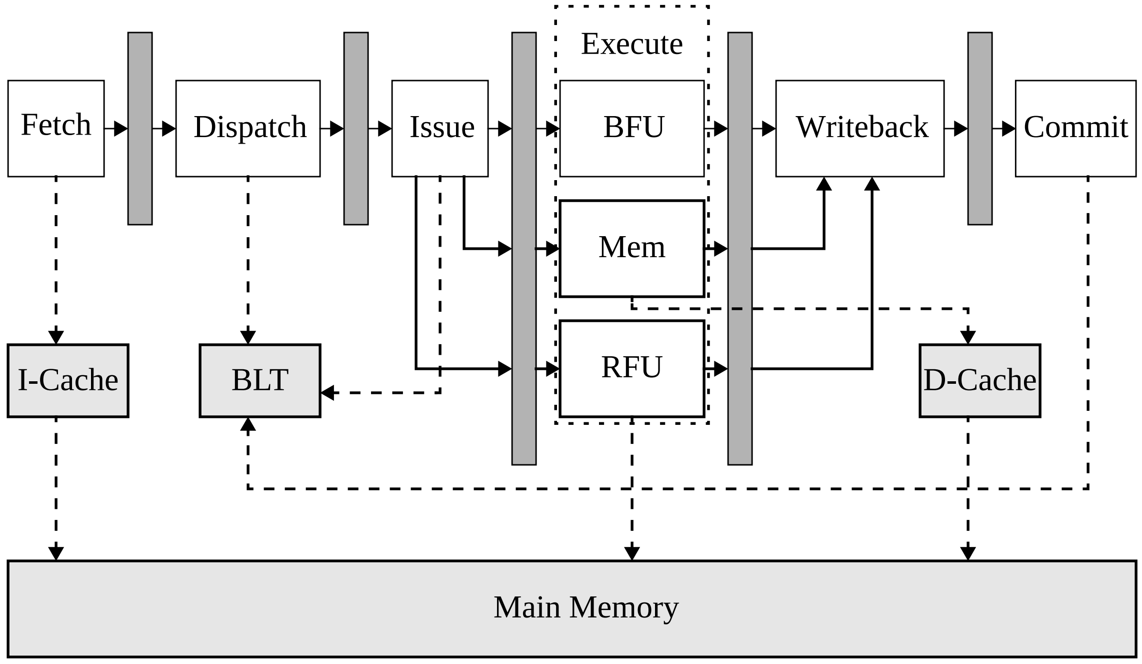

Figure 4-2. Sim-OneChip's Pipeline

RFU instructions. This stage adds entries in the BLT to lock memory blocks when RFU instructions are dispatched.

ISSUE STAGE

The *issue* stage identifies ready instructions from the scheduler queues (RUU, LSQ and RFU) and allows them to proceed in the pipeline. This stage also checks the BLT to keep memory consistency and stalls the corresponding instructions.

EXECUTE STAGE

The *execute* stage is where instructions are executed in the corresponding functional units. Completed instructions are scheduled on the event queue as writeback events. This stage is divided into three parallel stages: *bfu* stage, *mem* stage and *rfu* stage. The *bfu* stage is where all operations that require basic functional units, such as integer and floating point are executed; the *mem* stage is where all memory access operations are executed and has access to the D-cache, and; the *rfu* stage is where RFU instructions are executed.

WRITEBACK STAGE

The *writeback* stage remained unmodified and moves completed operation results from the functional units to the RUU. Dependency chains of completing instructions are also scanned to wake up any dependent instructions.

COMMIT STAGE

The *commit* stage retires instructions in-order and frees up the resources used by the instructions. It commits the results of completed instructions in the RUU to the register file and stores in the LSQ will commit their result data to the data cache. This stage clears BLT entries to remove memory locks once the corresponding RFU instruction is committed.

The BLT is accessed by the dispatch, issue and commit stages. The memory consistency scheme requires that instructions are entered in the BLT and removed from it in program order. In the pipeline, the issue, execute and writeback stages do not necessarily follow program order since out-of order issue, execution and completion is allowed. Hence, memory block locks and the corresponding entries in the BLT need to be entered when an RFU instruction is dispatched, since dispatching is done in program order. Likewise, entries from the BLT need to be removed when RFU instructions commit, since committing is also performed in program order.

All actions in the memory consistency scheme are taken in the issue stage. The issue stage is allowed to probe the BLT for memory locks. Instructions that conflict with locked memory blocks are prevented from issuing at this point. All others are allowed to proceed provided there are no dependencies.

4.5 Addition of instructions

As explained previously, annotations in SimpleScalar are useful for creating new instructions. Annotations are attached to the opcode in assembly files for the assembler to translate them and append them in the annotation field of assembled instructions as shown in Figure 4-1. Taking advantage of this feature, new instructions can be created without the need to modify the assembler. OneChip's RFU instructions will be disguised as already existing, but annotated instructions that the simulator will recognize as an RFU instruction and model the corresponding operation. Without the annotation, instructions are treated as regular ones; with the annotation they become instructions that target the reconfigurable unit.

SimpleScalar supports two types of annotations: bit and field annotations. Bit annotations tell the assembler to set any particular bit in the annotation field to 1. Annotation */a* will set bit number 0, while annotation */b* will set bit number 1 and successively all the way to */p* that will set bit number 15. Two examples of bit annotations are shown in Table 4-1. The first one with an annotation */c*, sets bit number 2 to the value of 1, while the second example */b/d/p* sets bits number 1, 4 and 15 to the value of 1.

Field annotations will tell the assembler to assign to the specified bits a specified value. Annotation */s:e(v)* will assign bits *s (start)* through *e (end)* the value of *v*. Two examples of field annotations are shown in Table 4-1. On the third row, the annotation */7:5(7)* will assign

Annotation Type	Annotated Instruction	Instruction Annotation Field
bit	add/c $r1,$r2,$r3	0000 0000 0000 0**1**00
bit	add/b/d/p $r4,$r2,$r3	**1**000 0000 0000 **1**0**1**0
field	add/7:5(7) $r1,$r5,$r2	0000 0000 **111**0 0000
field	add/11:4(9) $r5,$r3,$r1	0000 **0000 1001** 0000

Table 4-1. SimpleScalar Instruction Annotations

bits 7 through 5 the value of 7, or 111 binary. The last row shows how bits 11 through 4 are assigned the value of 9, or 00001001 binary, with the annotation */11:4(9)*.

The four instructions defined for OneChip (i.e. two RFU operation and two configuration instructions described in the previous chapter) were created for Sim-OneChip. Macros are used to translate from a C specification to the corresponding annotated assembly instruction. OneChip's RFU instruction macros are defined in `oc-lib.h`, which will be described in detail in Section 4.6.1.

4.6 Programming model

Currently, the programming model for OneChip is the use of circuit libraries. Programming for Sim-OneChip is done in C. To use the reconfigurable unit, the user needs to include the `oc-lib.h` library in the program. This library is not a circuit library, it is only the user interface for the RFU. It includes macros for defining configuration addresses, using and pre-loading configurations. These macros will generate the corresponding annotated instructions for the simulator.

RFU configurations are defined in the file `fpga.conf`. The user may use existing configurations from a library of configurations, or create custom ones. Configurations are defined in C and several macros are available for accessing memory or instruction fields. Details on how to define configurations are presented in Section 4.6.2.

The complete simulation process is shown in Figure 4-3. A C program that includes calls to reconfigurable instructions is compiled by the simplescalar gcc compiler *ssgcc* along with the OneChip Library `oc-lib.h`. This will produce a binary file that can be executed by the simulator *sim-onechip*. All the program configurations specified in `fpga.conf` must be previously compiled by gcc along with the simulator source code to produce the simulator.

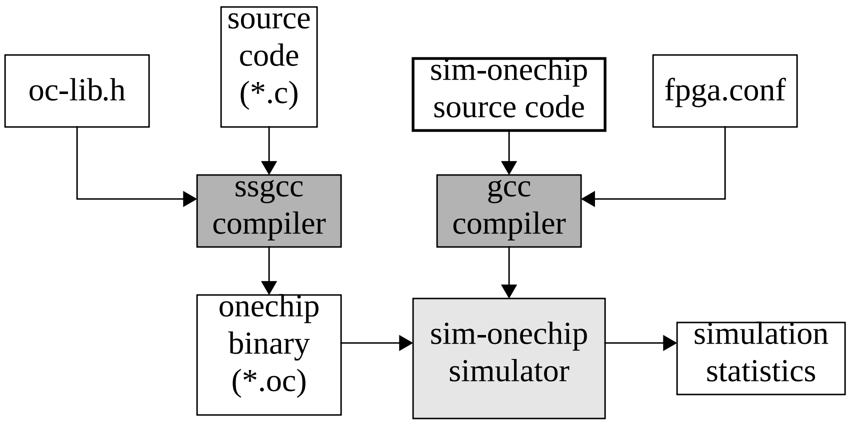

Figure 4-3. Sim-OneChip's Simulation Process

Once both binaries are ready, the simulator can simulate the execution of the binary and produce the corresponding statistics.

Sim-OneChip's processor specification can be defined as command-line arguments. One can specify the processor core parameters, such as fetch and decode bandwidth, internal queues sizes and number of execution units. The memory hierarchy and the branch predictor can also be modified.

4.6.1 OneChip library

This section is intended to be a guide for using the OneChip library. To be able to use RFU instructions, the following header must be included in the code.

```
#include “oc-lib.h”
```

The library defines the following five macros:

```
oc_configAddress(func, addr)
oc_preLoad(func)
rec_2addr(func, src_addr, dst_addr, src_size, dst_size)
```

```
rec_3addr(func, src1_addr, src2_addr, dst_addr, blk_size)
oc_encodeSize(size)
```

`oc_configAddress(func, addr)` is used for specifying the configuration address for a specified function. It will associate the function “func” with the address “addr” where the FPGA configuration bits will be taken from and will enter the corresponding entry in the BLT. For example,

```
oc_configAddress(1, 0x7FFFC000);
```

will associate the configuration located in memory at address 0x7FFFC000 to function number 1. The corresponding entry will be updated in the RBT and the *Address* field for *FPGA function* 1 will have the value 0x7FFFC000.

`oc_preLoad(func)` is used for pre-loading the configuration associated with the specified function “func” into an available FPGA context. If there are no currently available contexts, the least recently used configuration will be replaced. For example,

```
oc_preLoad(3);
```

will start loading the configuration located in the memory address currently associated with the *FPGA function* 3 in the RBT.

`rec_2addr(func, src_addr, dst_addr, src_size, dst_size)` is the two-operand reconfigurable instruction. The instruction parameters are the following:

`func`	The FPGA function number
`src_addr`	Source block address
`dst_addr`	Destination block address
`src_size`	Source block size encoded
`dst_size`	Destination block size encoded

This instruction will perform the context switch to activate function *func* and will execute the corresponding operation associated with it. It will also lock one source and one destination block of memory (*src_addr* and *dst_addr* of encoded sizes *src_size* and *dst_size* respectively) by entering the corresponding fields in the BLT for as long as the function takes to execute. When finished, the BLT entries corresponding to the instruction will be cleared. For example,

```
rec_2addr(0,&a,&b,4,4);
```

where `a` and `b` are defined as

```
unsigned char a[32], b[32];
```

will activate function 0 and perform the operation with the array `a` as source data and array `b` as destination data.

`rec_3addr(func, src1_addr, src2_addr, dst_addr, blk_size)` is the three-operand reconfigurable instruction. The instruction parameters are the following:

`func`	The FPGA function number
`src1_addr`	Source-1 block address
`src2_addr`	Source-2 block address
`dst_addr`	Destination block address
`blk_size`	Block size encoded

As with the previous instruction, this one will perform the context switch to activate function *func* and will execute the corresponding operation associated with it. It will also lock two source blocks (*src1_addr* and *src2_addr*) and one destination block of memory (*dst_addr)* by entering the corresponding fields in the BLT. The blocks here are all the same size (*blk_size*). For example,

```
rec_3addr(2,&a,&b,&c,3);
```

where `a`, `b` and `c` are defined as

```
unsigned char a[16], b[16], c[16];
```

will activate function 2 and perform the operation with arrays `a` and `b` as source data and array `c` as destination data.

`oc_encodeSize(size)` is a macro used for encoding the size of memory blocks. It obtains the encoded value from a table that is defined by the function `ln(size)/ln(2)-1`. This macro should be used to encode block sizes in reconfigurable instructions. For example,

```
rec_3addr(2,&a,&b,&c,oc_encodeSize(16));
```

where `a`, `b` and `c` are defined as

```
unsigned char a[16], b[16], c[16];
```

The encoded size passed to the reconfigurable instruction will be `ln(16)/ln(2)-1 =` **`3`**.

The details of these macros can be found in `oc-lib.h`, which is included in APPENDIX A.

4.6.2 Configuration definition

This section is intended to be a guide for creating configurations for OneChip. The behavior of the RFU is modeled with a high-level functional simulation. It is given some inputs, and a function produces the corresponding outputs without performing a detailed micro-architecture simulation of the programmable logic.

FPGA configurations are defined in the file `fpga.conf`. The makefile will search for this file when compiling the simulator in the simulator directory. Configurations are defined as follows,

```
DEFCONF(<addr>, <oplat>, <issuelat>,
        {
          <EXPR>
        }
        )
```

where

`<addr>`	Configuration address (i.e. the location of the configuration bits in memory).
`<oplat>`	Operation latency (i.e. the number of cycles until result is ready for use).
`<issuelat>`	Issue latency (i.e. the number of cycles before another operation can be issued on the same resource).
`<EXPR>`	Expression.

The separation of the instruction latency into operation and issue latencies, allows the specification of pipelined configurations. For example, assume one configuration takes 20 cycles to complete one instruction, but the configuration is pipelined and one instruction can be started every 4 cycles. In this case, the operation latency will be 20 and the issue latency will be 4. Hence, the configuration can have 20/4 = 5 executing instructions at a time and the throughput for the configuration is implied as 20/5 = 4 cycles per instruction.

The *expression* field is where the semantics of the configuration will be specified. It is a C expression that implements the configuration being defined, the expression must modify all architected state affected by the instruction's execution, by default, the next PC (NPC) value defaults to the current PC (CPC) plus SS_INST_SIZE, as a result, only taken branches need to set NPC.

All memory accesses in the DEFCONF() expression must be done through the memory interface. The available macros for memory reads and writes are the following.

READ_WORD(SRC)
READ_UNSIGNED_HALF(SRC)
READ_SIGNED_HALF(SRC)

```
READ_UNSIGNED_BYTE(SRC)
READ_SIGNED_BYTE(SRC)
WRITE_WORD(SRC, DST)
WRITE_HALF(SRC, DST)
WRITE_BYTE(SRC, DST)
```

For accessing general purpose registers, the following macros are available. The first macro will access register N and the second will set the register N to the value of EXPR.

```
GPR(N)
SET_GPR(N, EXPR)
```

In the same way, the following macros are provided to access floating point and miscellaneous registers.

```
FPR_L(N)
SET_FPR_L(N, EXPR)
FPR_F(N)
SET_FPR_F(N, EXPR)
FPR_D(N)
SET_FPR_D(N, EXPR)
HI
SET_HI(EXPR)
LO
SET_LO(EXPR)
FCC
SET_FCC(EXPR)
```

The following predefined macros are available for use in DEFCONF() expressions to access the value of the RFU instruction operand field values. The field size is shown in parentheses.

OC_FUNC	FPGA function (4 bits)
OC_SR	Source register (8 bits)
OC_DR	Destination register (8 bits)
OC_SBS	Encoded source block size (6 bits)
OC_DBS	Encoded destination block size (6 bits)

For three-operand instructions, the available macros are the following.

OC_3A_FUNC	FPGA function (4 bits)
OC_3A_S1R	Source1 register (8 bits)
OC_3A_S2R	Source2 register (8 bits)
OC_3A_DR	Destination register (8 bits)
OC_3A_BS	Encoded block size (6 bits)

The following two macros are defined to handle block sizes. One of them creates a block mask an the other decodes the block size. Both of them need to be passed an encoded block_size as parameter (OC_SBS, OC_DBS or OC_3A_BS).

OC_MASK(block_size)

OC_BLOCKSIZE(block_size)

The following data types are available for use in configurations.

OC_MASK_TYPE

OC_BLOCKSIZE_TYPE

Some configuration examples are included in the file `fpga.conf` in APPENDIX B.

4.7 A short example

This section will present an example of how to port an application so that it uses the RFU in OneChip to get speedup. The application to be implemented is an 8-tap FIR filter.

Consider you have this C code in a file called `fir.c`.

```
 1: /* FILE: fir.c */
 2:
 3: #include <stdio.h>
 4:
 5: #define TAPS 8
 6: #define MAX_INPUTS 1024
 7:
 8: int coef[TAPS] = {1,2,3,4,5,6,7,8};
 9: int inputs[MAX_INPUTS];
10:
11: void main(){
12:   int i, j;
13:   int *x;
14:   int y[MAX_INPUTS];
15:
16:   /* Set the inputs to some random numbers */
17:   for (i = 0; i < MAX_INPUTS; i++){
18:     inputs[i] = 3 * (i % 1000) - MAX_INPUTS % 123;
19:     y[i] = 0;
20:   }
21:   x = inputs;
22:
23:   /* FIR Filter kernel */
24:   for (i = 0; i < MAX_INPUTS; i++){
25:     for (j = 0; j < TAPS; j++){
26:       y[i] += coef[j] * x[j];
27:     }
28:     x++;
29:   }
30:
31:   printf("\n8-tap FIR filter done!\n");
32: }
```

The inner loop in the FIR filter kernel on line 25 on `fir.c` can be ported to be executed entirely on the OneChip RFU. For that, we need to do some modifications to the C code. The file `fir.oc.c` that reflects this changes is shown below.

```
 1: /* FILE: fir.oc.c */
 2:
 3: #include <stdio.h>
 4: #include “oc-lib.h”
 5:
 6: #define TAPS 8
 7: #define MAX_INPUTS 1024
 8:
 9: int coef[TAPS] = {1,2,3,4,5,6,7,8};
10: int inputs[MAX_INPUTS];
11:
12: void main(){
13:   int i, j;
14:   int *x;
15:   int y[MAX_INPUTS];
16:
17:   oc_configAddress(0, 0x7FFFC002);
18:   oc_preLoad(0);
19:
20:   /* Set the inputs to some random numbers */
21:   for (i = 0; i < MAX_INPUTS; i++){
22:     inputs[i] = 3 * (i % 1000) - MAX_INPUTS % 123;
23:     y[i] = 0;
24:   }
25:   x = inputs;
26:
27:   /* FIR Filter kernel */
28:   for (i = 0; i < MAX_INPUTS; i++){
29:     rec_3addr(0, x, coef, &y[i], oc_encodeSize(8));
30:     x++;
31:   }
32:
33:   printf(“\n8-tap FIR filter done!\n”);
34: }
```

The first step was including the OneChip library in the code as explained in section 4.6.1 and shown on line 4 in `fir.oc.c`. The second step was defining the address of the configuration bitstream for the FIR filter. In this case, we are using configuration #0 and the memory address is 0x7FFFC002, as shown on line 17. As a third step, notice that lines 25-27 on `fir.c` have been removed and replaced by a 3-operand RFU instruction in line 29 on `fir.oc.c`. This instruction is using configuration #0 and is passing the address of the two source memory blocks, `x` and `coef` which are pointers, as well as the address of destination memory block, which for each iteration will be `&y[i]`. The block size 8 is passed using the function oc_encodeSize as explained in section 4.6.1.

The previous three changes are necessary. Furthermore, if we want to reduce configuration overhead, we would introduce a pre-load instruction as in line 18. This instruction tells the processor that configuration #0 will be used soon. This way, by the time it gets to execute the RFU instruction, the configuration is already loaded and no time is spent waiting for the configuration to be loaded. This instruction is not necessary, because if the configuration is not loaded in the FPGA, the processor will automatically load it.

Now that the C code has been modified to use the RFU, we need to define the FPGA configuration that will perform the FIR filter. As explained in section 4.6.2, configurations are defined in a file called `fpga.conf`. The fir filter definition used is shown below.

```
/* This configuration is for a 3-operand instruction. It is used
   for a fir filter program. */

DEFCONF(0x7FFFC002, 24, 24,
        {
          int oc_index;            /* temp for indexing */
          unsigned int oc_word;    /* temp for storing words */
         unsigned int oc_result;  /* temp for storing result */

          oc_result = 0;
          for(oc_index = 0;
              oc_index <= OC_MASK(OC_3A_BS);
              oc_index++)
            {
              oc_word = READ_WORD(GPR(OC_3A_S1R)+(4*oc_index));
              oc_word*= READ_WORD(GPR(OC_3A_S2R)+(4*oc_index));
              oc_result += oc_word;
            }
          WRITE_WORD(oc_result, GPR(OC_3A_DR));
        }
)
```

This configuration is the equivalent of the inner loop in the FIR filter kernel on lines 25-27 in `fir.c`. Note that in the configuration, each memory access is done through the memory interface as explained in section 4.6.2. Lines 11-13 define the iteration loop for the FIR filter. Line 15 reads a word from the memory location defined by the address stored in the general purpose register that contains one source address plus the corresponding memory offset[1]. In the same way, line 16 reads a word from the other source block and multiplies it with the data previously read and stored in the `oc_word` variable. Line 17 simply accumulates the multiplied values across loop iterations. When the loop is finished, line 19 writes the result into the memory location defined by the address stored in the general purpose register that has the destination block address.

The configurations file must be compiled and linked with the simulator. It must be located in the same directory as the simulator source files. To compile the simulator, on the directory where the source is located, type

```
make sim-onechip
```

To compile the original FIR filter application that can be executed on *sim-onechip* or any other of the SimpleScalar's simulators, type

```
ssgcc -o fir.ss fir.c
```

To compile the FIR filter application that uses the OneChip's RFU, type

```
ssgcc -o fir.oc fir.oc.c
```

This will produce two binaries `fir.ss` and `fir.oc`. After compiling the simulator and the applications, run the normal FIR filter application on sim-onechip by typing

```
sim-onechip fir.ss
```

1. The offset is obtained by multiplying the index by 4, since words are 4 bytes.

And for the RFU version, type

```
sim-onechip fir.oc
```

The simulator will generate statistics for the number of instructions executed in each program. The speedup obtained with Sim-OneChip can be verified.

4.8 Simulator testing

Several test programs were written to test the simulator. The six hazard situations were forced with six different programs. A C program with an RFU instruction was compiled and the corresponding assembly language generated. Memory reads and writes were inserted in the program at the assembly level to force each of the six hazards. Each of these programs was executed on the simulator and the correct operation was verified. The assembly programs used for testing are `sit0[1-6].s`, which hold situations 1-6 respectively. These files, which were assembled with the *ssas* assembler, are included in APPENDIX C.

4.9 Summary

This chapter describes an architecture simulator of the OneChip reconfigurable processor called Sim-OneChip. It is based on the *sim-outorder* simulator from the SimpleScalar tool set. The description is focused on the modifications and enhancements done to *sim-outorder* to support OneChip's features. Several applications were executed on the system to simulate its performance. Details on this experiments and the results obtained will be the topic of the next chapter.

CHAPTER 5 Applications and Results

In the last chapter, Sim-OneChip, the architecture simulator of the OneChip reconfigurable processor was described. This chapter will present the benchmark applications that were used to evaluate the system's performance. The results of the modified applications for execution in Sim-OneChip will be presented and analyzed.

5.1 Experimental setup

Benchmark applications are used to measure different aspects of processor performance. While no single numerical measurement can completely describe the performance of a complex device like a microprocessor, benchmarks are useful tools for testing and comparing different systems.

To evaluate the performance of the OneChip architecture, several applications were compiled and executed on Sim-OneChip. To do the experiments, four steps were performed for each application. Step one is the identification of which parts of each application are suitable for implementation in hardware. Step two is modeling the hardware implementation of the identified parts of the code. Step three is the replacement of the identified code in the application, with the corresponding hardware function call. And step four is the execution and verification of both the original and the ported versions of the application.

The first step in the experiments is to identify the parts of an application that should be ported to the RFU. For OneChip, these parts include loops with regular memory access and operations behavior, such as memory streaming applications. For a piece of code to be a candidate, it must take a relatively high execution time and perform memory accesses in a regular manner, as in applications suitable for vector processors. In general, any application that can be sped up by a vector processor, will be also suitable for OneChip. This identification is done by profiling the applications and by analyzing the results to determine which pieces of code spend the most time executing with respect to the others. Although it is not a rule, a piece of code that spends a lot of time executing is a good candidate for being ported to custom hardware. By analyzing these pieces of code, one can identify loops to port to the RFU.

The second step is modeling the implementation in hardware of the identified parts of the code. Although this is straightforward for OneChip since the identified code will be almost the same as the one to be put in the configuration definition expression, as described in Section 4.6.2, it is a bit tricky because all memory accesses need to be done through the memory interface to have a reliable simulation. Recall the example in the previous chapter in Section 4.7, where the code of the FIR filter

```
for (j = 0; j < TAPS; j++){
  y[i] += coef[j] * x[j];
}
```

was ported to hardware as the expression

```
oc_result = 0;
for(oc_index = 0;
    oc_index <= OC_MASK(OC_3A_BS);
    oc_index++)
  {
   oc_word = READ_WORD(GPR(OC_3A_S1R)+(4*oc_index));
   oc_word *= READ_WORD(GPR(OC_3A_S2R)+(4*oc_index));
   oc_result += oc_word;
```

```
          }
          WRITE_WORD(oc_result, GPR(OC_3A_DR));
        }
```

where all the processor state affected by the instruction is specified through the available macros. The operation latency for each of the expressions will be described later for each application. The simulator needs to be recompiled each time new configurations are added or modified.

The third step is the replacement of the identified code in the application with the corresponding hardware function call. This RFU function call will use the specified configuration to perform whatever the software code did, but in the customized hardware. As in the example in Section 4.7, the code for the FIR filter

```
        for (j = 0; j < TAPS; j++){
          y[i] += coef[j] * x[j];
        }
```

was replaced with

```
        rec_3addr(0, x, coef, &y[i], oc_encodeSize(8));
```

which is a function call to a reconfigurable instruction that uses the configuration implementing the FIR filter in hardware. The simulator will execute the configuration expression, updating the architected state of the processor and will simulate the operation and issue latencies of the configuration, stalling instructions according to the memory coherence scheme.

Finally, the fourth step is the execution and verification of the applications. The original application is compiled and executed on *sim-outorder,* which has the same architecture as OneChip, except for the RFU. The ported application is compiled and executed on *sim-onechip*. The speedup reported is obtained from comparing the results of both applications.

The pipeline configuration used for both simulations was the default used in SimpleScalar. Among the most relevant characteristics are an instruction fetch queue size of 4 instructions; instruction decode, issue and commit bandwidths of 4 instructions per cycle; a 16-entry register update unit (RUU) and an 8-entry load/store queue (LSQ). The number of execution units available in the pipeline are 4 integer ALU's, 1 integer multiplier/divider, 2 memory system ports available (to CPU), 4 floating-point ALU's, 1 floating-point multiplier/divider. Also, in the case of *sim-onechip*, 1 reconfigurable functional unit (RFU), an 8-entry RBT and a 32-entry BLT were used. The branch predictor and cache configuration remained unmodified as well.

5.2 Benchmark applications

There is currently no standard benchmark suite for reconfigurable processors. S. Kumar et al. [50] from Honeywell Technology Center, have proposed a suite of stressmarks, which are benchmarks that focus on specific points of interest, such as versatility, capacity, interface, scalability, timing sensitivity and CAD performance. The applications include algorithms such as image compression, Huffman encoding, two-dimensional vector rotation (CORDIC), Fast Fourier Transform (FFT), Constant False Alarm Rate (CFAR) and boolean satisfiability (SAT). Unfortunately, this suite is not publicly available.

C. Lee et al. [51] from the University of California at Los Angeles have proposed a set of benchmarks for evaluating multimedia and communication systems, which is called MediaBench. Since current reconfigurable processors available are used mostly for communications applications [42][43], MediaBench was taken as the suite for evaluating OneChip. Not all of the applications were used for the evaluation. Some of them could not be ported to SimpleScalar, due to the complexity of the makefiles or due to some missing libraries. However, the rest of the applications can provide good feedback on the architecture's performance.

The applications chosen from MediaBench to evaluate OneChip's performance are

- JPEG: Image compression and decompression.
- MPEG: Digital video encoding and decoding.
- GSM: Full-rate speech encoding.
- G.721: Voice compression and decompression.
- PEGWIT: Public key encryption and authentication.
- Mesa: 3-D graphics library (texture mapping and standard rendering).
- RASTA: Speech recognition.
- EPIC: Image compression and decompression.
- ADPCM: Audio encoding and decoding.

5.3 Profiles

Profiling the execution of an application helps to identify the parts of the application where the processor spends its time and which functions are called with highest frequency. This information tells us which parts of the application take a lot of time to execute and hence, being candidates for rewriting to make it execute faster. Profiling of the applications was performed using GNU's profiler *gprof* included in GNU's *binutils 2.9.1* package [52].

Profiling has several steps:

- The application must be compiled and linked with profiling options enabled. (i.e. the options `-g` and `-pg` must be specified when compiling the program).
- The application must be executed several times to generate a reliable profile data file. This is to acquire more samples in each program and account for any random data or short running functions. Our tests were done by executing each application 100 times.
- The profiler must be executed to analyze the profile data and generate a readable report.

Several forms of output are available from the report. The most important for our work, since

	% time	cumulative seconds	self seconds	calls	self us/call	total us/call	name
ADPCM decode	100.00	2.31	2.31	14800	156.08	156.08	adpcm_decoder
	0.00	2.31	0.00	100	0.00	23100.00	main
ADPCM encode	100.00	2.27	2.27	14800	153.38	153.38	adpcm_coder
	0.00	2.27	0.00	100	0.00	22700.00	main
EPIC encode	67.99	15.25	15.25	2400	6354.17	6395.83	internal_filter
	6.51	16.71	1.46	1300	1123.08	1123.08	quantize_image
	4.37	17.69	0.98	400	2450.00	2450.00	internal_transpose
	3.03	18.37	0.68	100	6800.00	6800.00	ReadMatrixFromPGMStream
	2.67	18.97	0.60				_sbrk_unlocked
EPIC decode	21.43	1.20	1.20				_write
	18.21	2.22	1.02	100	10200.00	35000.00	main
	17.14	3.18	0.96	400	2400.00	2400.00	internal_int_transpose
	12.32	3.87	0.69	100	6900.00	16500.00	collapse_pyr
	8.93	4.37	0.50				_libc_open
G721 decode	28.92	84.10	84.10				internal_mcount
	28.80	167.87	83.77	236032000	0.00	0.00	fmult
	15.53	213.02	45.15	29504000	0.00	0.00	update
	4.35	225.68	12.66	29504000	0.00	0.00	tandem_adjust_ulaw
	2.49	232.93	7.25	29504000	0.00	0.01	g721_decoder
G721 encode	30.89	83.98	83.98	236032000	0.00	0.00	fmult
	28.71	162.02	78.04				internal_mcount
	16.67	207.33	45.31	29504000	0.00	0.00	update
	4.35	219.16	11.83	29504000	0.00	0.00	quantize
	2.93	227.12	7.96	29504000	0.00	0.01	g721_encoder
GSM toast	55.71	171.27	171.27	1014800	0.17	0.17	Calculation_of_the_LTP
	19.00	229.69	58.42	1014800	0.06	0.06	Short_term_analysis_fi
	6.07	248.34	18.65	253700	0.07	0.07	Autocorrelation
	4.03	260.72	12.38	1014800	0.01	0.01	Weighting_filter
	2.94	269.76	9.04	253700	0.04	0.04	Gsm_Preprocess
GSM untoast	63.72	60.18	60.18	1014800	59.30	59.30	Short_term_synthesis_f
	8.79	68.48	8.30				_libc_write
	5.52	73.69	5.21	1014800	5.13	5.13	Gsm_Long_Term_Synthesis
	4.07	77.53	3.84				internal_mcount
	3.74	81.06	3.53	253700	13.91	13.91	Postprocessing
JPEG encode	20.87	4.30	4.30	572000	7.52	7.52	jpeg_fdct_islow
	15.87	7.57	3.27	384800	8.50	19.67	forward_DCT
	14.08	10.47	2.90	577200	5.02	5.02	encode_one_block
	12.43	13.03	2.56	41600	61.54	61.54	rgb_ycc_convert
	7.86	14.65	1.62				_libc_read

Table 5-1. **Profiling results**

	%	cumulative	self		self	total	
	time	seconds	seconds	calls	us/call	us/call	name
JPEG decode	47.49	11.63	11.63				_libc_write
	16.54	15.68	4.05	572000	7.08	7.08	jpeg_idct_islow
	9.88	18.10	2.42	41600	58.17	58.17	ycc_rgb_convert
	6.94	19.80	1.70				_libc_open
	5.68	21.19	1.39	41600	33.41	33.41	h2v2_fancy_upsample
MESA mipmap	35.31	17.73	17.73				_libc_write
	13.24	24.38	6.65	200	33250.00	64500.00	lambda_textured_triangl
	8.15	28.47	4.09				putc_unlocked
	7.29	32.13	3.66				__ieee754_log
	6.81	35.55	3.42	7761800	0.44	0.44	sample_2d_nearest
MESA osdemo	56.14	13.62	13.62				_write
	8.20	15.61	1.99				putc_unlocked
	5.61	16.97	1.36	100	13600.00	34400.00	main
	5.15	18.22	1.25				fputc
	5.11	19.46	1.24	102000	12.16	12.16	smooth_color_z_triangle
MESA texgen	24.25	16.00	16.00				_write
	12.70	24.38	8.38	1344000	6.24	6.24	de_casteljau_surf
	9.46	30.62	6.24	621600	10.04	28.15	general_textured_triang
	8.73	36.38	5.76	5954600	0.97	0.97	sample_1d_linear
	6.26	40.51	4.13				internal_mcount
MPEG2 decode	30.20	5.46	5.46	115200	47.40	47.40	Reference_IDCT
	18.81	8.86	3.40				_libc_close
	16.59	11.86	3.00				_open
	10.51	13.76	1.90				_write
	7.25	15.07	1.31				internal_mcount
MPEG2 encode	57.01	35.02	35.02	12401400	2.82	2.82	dist1
	12.78	42.87	7.85	115200	68.14	68.14	fdct
	6.07	46.60	3.73				internal_mcount
	3.61	48.82	2.22	96000	23.12	387.92	fullsearch
	2.67	50.46	1.64	62400	26.28	26.28	quant_non_intra
PEGWIT decrypt	35.51	2.67	2.67	2028600	1.32	1.32	gfAddMul
	24.34	4.50	1.83	63400	28.86	28.86	gfMultiply
	6.78	5.01	0.51				internal_mcount
	5.05	5.39	0.38	100	3800.00	3800.00	gfInit
	4.92	5.76	0.37	31700	11.67	100.30	gfInvert
PEGWIT encrypt	39.07	5.38	5.38	4074186	1.32	1.32	gfAddMul
	26.87	9.08	3.70	127232	29.08	29.08	gfMultiply
	6.03	9.91	0.83				internal_mcount
	4.79	10.57	0.66	63666	10.37	98.48	gfInvert
	3.70	11.08	0.51	136666	3.73	3.73	gfSquare
RASTA	16.63	11.93	11.93	1014000	11.77	11.77	FR4TR
	14.69	22.47	10.54				__ieee754_pow
	9.46	29.26	6.79	253500	26.79	28.53	audspec
	4.68	32.62	3.36				__kernel_cos
	4.64	35.95	3.33	253500	13.14	13.18	fill_frame

Table 5-1. Profiling results

we are concerned about which functions take most of the cycles, is the flat profile. This profile shows how much time the processor spent in each function of the application, and how many times that function was called. More information on *gprof* can be found in the on-line manual [53].

For the nine applications chosen, the profiling information is shown in Table 5-1. The first five lines of the flat profile table is shown for each application. A sample was taken every 0.01 seconds. In the table, *% time* is the percentage of the total execution time the program spent in a particular function. *Cumulative seconds* is the cumulative total number of seconds the computer spent executing the function, plus the time spent in all the functions above this one in this table. *Self seconds* is the number of seconds accounted for by this function alone. Calls is the total number of times the function was called. *Self μs/call* represents the average number of microseconds spent in the function per call. *Total μs/call* represents the average number of microseconds spent in the function and its descendants per call. And *name* is the name of the function. In the table, some fields are left blank because the function was not compiled with profiling enabled, as is the case for libraries.

The table shows the profile data of the encoder and decoder for ADPCM, EPIC, G21, JPEG and MPEG2; toast and untoast for GSM; encrypt and decrypt for PEGWIT; RASTA filtering; and three different applications of MESA, which are texture mapping (texgen), standard rendering (osdemo) and fast texture mapping (mipmap).

5.4 Analysis and modifications

From the profiling information presented in Section 5.3, we can identify specific functions in each application that are worth improving by executing them in specialized hardware implemented in the OneChip reconfigurable unit. One could just select the most time consuming function and port it to the reconfigurable unit, however we must further analyze the code and see what kind of operations are done in the function to verify if a hardware version is feasible. The profiling information provides only a pointer to the pieces of code that are candidates to be ported.

Some of the applications were discarded for modification due to several reasons. The EPIC encoder, even though the `internal_filter` function executes for 67.99% of the time, was discarded because it acts a function manager that calls other smaller functions which do not appear near the top of the flat profile and do not execute for a considerable time. The EPIC decoder was discarded because it spends most of the time in system calls (i.e. `_write`), which can’t be ported to the reconfigurable unit.

Both, the G721 encoder and decoder have the `internal_mcount` function as time consuming. This function is introduced by the profiler to keep track of function calls. Also in both, the fmult function is time consuming. This is a software implementation of a floating point operation that can be optimized with floating point units rather than a reconfigurable unit, hence we discard G721.

GSM was also discarded due to the complexity of the application. The makefile first compiles the library `libgsm.a` and then the applications use the library. Due to the nature of the sim-onechip environment, it becomes very complicated to port the library into hardware. Likewise, the MESA applications use several libraries `libMesa*.a` which besides being libraries, they spend a lot of time in system calls, so it was discarded as well.

In RASTA, the function `FR4TR` uses up to four arrays of input data and OneChip supports up to two input arrays. To port the application, it will be necessary to redesign the application and rewrite a lot of pieces of code, hence it was also discarded.

Four types of applications were ported to OneChip. JPEG Image compression, ADPCM Audio coding, PEGWIT Data encryption and MPEG2 Video encoding. The encoder and the decoder for each one was ported. The modifications to the applications are done by hand (i.e. no compiler technologies are used).

For the RFU timing in each of the applications, we assume that memory accesses dominate the computational logic and that our bottleneck is the memory bandwidth. If we also assume that one memory access is perfomed in one cycle, the latency of an operation will be obtained from counting the total number of memory accesses performed by the operation.

5.4.1 ADPCM

The Adaptive Differential Pulse Code Modulation (ADPCM) is a very simple algorithm for audio coding. It is a family of speech compression and decompression algorithms. A common implementation takes 16-bit linear PCM samples and converts them to 4-bit samples, yielding a compression rate of 4:1.

From the ADPCM encoder application profiles, we denote that all the time is spent in the routine `adpcm_encoder`. This is because the application only takes data from standard input, compresses the data and writes to standard output. There is no data validation or other operation done in the application, hence it spends most of the time in the application kernel. The `adpcm_encoder` routine reads 1000 half words[1] and writes 500 bytes. Since the compression uses combinational logic and two tables, which can be implemented in a RAM, we will port the whole routine to hardware. Assuming the combinational logic is fast enough and that the bottleneck for the computations is the memory bandwidth, we will define the RFU configuration operation and issue latencies as 1500 for the total memory accesses. The `adpcm_encoder` function was replaced in the original application with a call to the reconfigurable unit.

1. A word is 4 bytes wide and a half-word is 2 bytes wide.

The ADPCM decoder is similar to the encoder, but does the inverse function. An RFU configuration for it was also written and the `adpcm_decoder` function was replaced by its corresponding reconfigurable function call in the original application.

5.4.2 JPEG

The Joint Photographic Experts Group committee [54] wrote JPEG, a standardized image compression mechanism for full-color or gray-scale images. JPEG is a lossy algorithm, meaning that the decompressed image isn't quite the same as the original one, and takes advantage of known limitations of the human eye.

From the JPEG encoder application profiles, we denote that 20.87% of the time is spent in the routine `jpeg_fdct_islow`, which is an implementation of a Forward Discrete Cosine Transform (FDCT). Previous research and experimentation in our group [3] has demonstrated that a 2-dimensional FDCT can be efficiently implemented in an FPGA and computed in 0.2 μs. If the FDCT reads 64 words, performs the computation and writes 64 words back, the memory bandwidth will be again our bottleneck. An RFU configuration will be defined to calculate the FDCT and will have operation and issue latencies of 128 for the total memory accesses.

From the JPEG decoder application profiles, we denote that 16.54% of the time is spent in the routine `jpeg_idct_islow`, which is an implementation of an Inverse Discrete Cosine Transform (IDCT). Even though the application spends more time in system calls (i.e. 47.49% in `_libc_write`), the IDCT can be easily ported to hardware, as it performs the inverse operation as the FDCT does. As with the FDCT in the JPEG encoder, an RFU configuration was created for the IDCT with operation and issue latencies of 128.

5.4.3 MPEG

The Moving Picture Experts Group committee [55] developed MPEG, a standard application used for coding audio-visual information in a digital compressed format. The MPEG family of standards includes MPEG-1, MPEG-2, MPEG-4, MPEG-7 and MPEG-21. MPEG-2 is the standard on which such products as Digital Television set top boxes and DVD are based. Video Compression relies on the eye’s inability to resolve high frequency color changes and the redundancy within and between frames.

From the MPEG2 decoder application profiles, we denote that 30.20% of the time is spent in the routine `Reference_IDCT`, which is also an implementation of an Inverse Discrete Cosine Transform (IDCT). The IDCT reads 64 half-words, performs the computation and writes 64 half-words back. An RFU configuration was defined to calculate the IDCT, which has operation and issue latencies of 64. If the data is properly aligned in memory, 128 half-word memory accesses can be done as 64 word memory accesses.

From the MPEG2 encoder application profiles, we denote that 12.72% of the time is spent in the routine `fdct`, which is again an implementation of an Forward Discrete Cosine Transform (FDCT). As with the IDCT in the MPEG decoder, an RFU configuration was created for the FDCT with operation and issue latencies of 64.

5.4.4 PEGWIT

PEGWIT is a program for performing public key file encryption and authentication. It uses an elliptic curve over the Galois Field of order 2^{255}, and the symmetric block cipher square.

From the PEGWIT encrypt and decrypt profiles, we denote that 39.07% and 35.51% of the time in each case respectively, are spent in the routine `gfAddMul`, which is a Galois Field Addition and Multiplication. The function takes an input data vector, performs the corresponding operation and outputs the result data vector. Vector sizes vary through function calls up to a maximum of 17 half-word data elements plus the vector size, making 18 reads and 18 writes for a total of 36 half-word accesses. If data is aligned in memory, data can be accessed in 18 word accesses. Therefore, considering again that a hardware configuration can be fast enough and the memory bandwidth is the bottleneck, an RFU configuration was created to perform the operation with an operation and issue latencies of 18.

5.5 Results

The original and the modified versions of the eight chosen applications were executed on their respective simulator. Each application was tested with three different sizes of data, one small, one medium and one large. Four experiments were done for each application. Our first experiment was executing the original applications with *in-order* issue to verify how many cycles each one takes to execute. As a second experiment, we executed the OneChip version of each application also with *in-order* issue. This way, we could verify the speedup obtained by simply using the reconfigurable unit in the OneChip pipeline. The third experiment was executing the original version of the applications with *out-of-order* issue to verify the speedup obtained by simply using the *out-of-order* issue feature. And the fourth and last experiment was executing again the OneChip version, but now with *out-of-order* issue. This way we could verify the speedup obtained by using both features, the reconfigurable unit and the *out-of order* issue, in the OneChip pipeline at the same time. The execution time of the simulations are shown in Table 5-2. The table shows the results of the four experiments done for each application.

The input data used for JPEG are a 227x149 Raw PPM image with 2645 colors (57476 bytes), a 580x416 Raw PPM image with 1546 colors (723855 bytes) and 1024x768 Raw PPM image with 7371 colors (2359312 bytes). The three images were first converted to the JPEG format with the encoder and converted back to PPM with the decoder. For ADPCM, the three input

Application	Data size	Original version inorder (A)	OneChip version inorder (B)	Original version outorder (C)	OneChip version outorder (D)
JPEG encode	Small	7887987	5765222	3442853	2564950
	Medium	53518581	39313836	23338135	17534301
	Large	166768434	121070511	71471848	52895500
JPEG decode	Small	5431229	4207685	2196403	1834218
	Medium	36772217	28584747	14587409	12226740
	Large	112817890	90311725	44523923	38477007
ADPCM encode	Small	2771009	123791	1794829	105326
	Medium	7397678	281834	4563939	255657
	Large	37852407	1264984	24320754	1182330
ADPCM decode	Small	2262430	123497	1414301	104992
	Medium	6128925	281281	3779778	255159
	Large	30851417	1262665	19208935	1180646
PEGWIT encrypt	Small	35378451	24311382	16900856	11810028
	Medium	43647207	32868895	19825740	14554560
	Large	72913465	62889749	29455311	23717766
PEGWIT decrypt	Small	18462721	13191721	8868985	6265859
	Medium	24361054	19083603	10732235	8120484
	Large	45481145	40197063	17390436	14773347
MPEG2 decode	Small	32606894	6951738	16123835	2965464
	Medium	214296738	42287665	103433032	18135180
	Large	582809167	111521087	279533226	47267445
MPEG2 encode	Small	176798350	152831105	93182732	81755330
	Medium	723308492	557412170	388121565	308590949
	Large	2084474531	1629443773	1116596443	898911309

Table 5-2. **Execution time (# of cycles)**

data are a 0:06.6 sec. 16-bit linear, 8kHz, mono, Raw PCM audio file (104876 bytes); a 0:18.4 sec. 16-bit linear, 8kHz, mono, Raw PCM audio file (295040 bytes); and a 1:30.9 sec. 16-bit linear, 8kHz, mono, Raw PCM audio file (1454582 bytes). These audio files were converted to ADPCM with the encoder and converted back to PCM with the decoder. For PEGWIT, the input data are a 8268-byte text file, a 91503-byte text file and a 389718-byte text file. The three text files were encrypted and decrypted back with the encrypter and decrypter. And for MPEG, the three input data are a 128x128, 3 frames, 0.48 sec. M2V video file (6796 bytes); a 352x240, 4 frames, 1.14 sec. M2V video file (34906 bytes); and a 352x240, 11 frames, 3.05 sec. M2V video file (113203 bytes) which were decoded with the MPEG2 decoder and encoded back with the MPEG2 encoder. All files were verified to have the correct data after being encoded and decoded with the simulator.

The speedup obtained from the experiments is shown in Table 5-3. The first column **(A/B)** shows the speedup obtained by only using the reconfigurable unit. The second column **(C/D)** shows the speedup obtained by introducing a reconfigurable unit to an out-of-order issue pipeline. The third column **(A/C)** shows the speedup obtained by only using out-of-order issue. The fourth column **(B/D)** shows the speedup obtained by introducing out-of-order issue to a pipeline with a reconfigurable unit as OneChip. The fifth column **(A/D)** shows the total speedup obtained by using the reconfigurable unit and out-of order issue at the same time.

Further analyzing the simulation statistics, we note that there are no BLT instruction stalls (i.e. instructions stalled due to to memory locks) in the applications, except for JPEG. This means that either the RFU is fast enough to keep up with the program execution or there are no memory accesses performed in the proximity of the RFU instruction execution. The second one is the actual case. It is important not to confuse BLT stalls, which prevent data hazards, with stalls due to unavailable resources, which are structural hazards. If there are two consecutive RFU instructions with no reads or writes in between, there will most likely be a structural hazard. Since there is only one RFU, the trailing RFU instruction will be stalled until the RFU is available. This is not considered a BLT stall. A discussion of these results is presented in the following section.

Application	Data size	Onechip inorder (A/B)	OneChip outorder (C/D)	Outorder original (A/C)	Outorder OneChip (B/D)	Total (A/D)
JPEG encode	Small	1.37X	1.34X	2.29X	2.25X	3.08X
	Medium	1.36X	1.33X	2.29X	2.24X	3.05X
	Large	1.38X	1.35X	2.33X	2.29X	3.15X
JPEG decode	Small	1.29X	1.20X	2.47X	2.29X	2.96X
	Medium	1.29X	1.19X	2.52X	2.34X	3.01X
	Large	1.25X	1.16X	2.53X	2.35X	2.93X
ADPCM encode	Small	22.38X	17.04X	1.54X	1.18X	26.31X
	Medium	26.25X	17.85X	1.62X	1.10X	28.94X
	Large	29.92X	20.57X	1.56X	1.07X	32.02X
ADPCM decode	Small	18.32X	13.47X	1.60X	1.18X	21.55X
	Medium	21.79X	14.81X	1.62X	1.10X	24.02X
	Large	24.43X	16.27X	1.61X	1.07X	26.13X
PEGWIT encrypt	Small	1.46X	1.43X	2.09X	2.06X	3.00X
	Medium	1.33X	1.36X	2.20X	2.26X	3.00X
	Large	1.16X	1.24X	2.48X	2.65X	3.07X
PEGWIT decrypt	Small	1.40X	1.42X	2.08X	2.11X	2.95X
	Medium	1.28X	1.32X	2.27X	2.35X	3.00X
	Large	1.13X	1.18X	2.62X	2.72X	3.08X
MPEG2 decode	Small	4.69X	5.44X	2.02X	2.34X	11.00X
	Medium	5.07X	5.70X	2.07X	2.33X	11.82X
	Large	5.23X	5.91X	2.08X	2.36X	12.33X
MPEG2 encode	Small	1.16X	1.14X	1.90X	1.87X	2.16X
	Medium	1.30X	1.26X	1.86X	1.81X	2.34X
	Large	1.28X	1.24X	1.87X	1.81X	2.32X

Table 5-3. Speedup

5.6 Discussion

Based on results presented in the previous section, this section will present a discussion on the behavior of each of the benchmarks used for the evaluation of OneChip.

In the case of JPEG, there are CPU reads and writes performed in the proximity of RFU instructions. These are shown in Table 5-4. *RFU instructions* shows the total dynamic count of

Application	Data size	RFU instructions (X)	BLT instruction stalls (Y)	Stalls per RFU instruction (Y/X)	RFU Overlapping (128 - Y/X)
JPEG encode	Small	851	99531	116.96	11.04
	Medium	5720	669204	116.99	11.01
	Large	18432	2156508	117.00	11.00
JPEG decode	Small	851	104422	122.71	5.29
	Medium	5720	702466	122.81	5.19
	Large	18432	2264375	122.85	5.15

Table 5-4. **JPEG RFU instructions**

RFU instructions in the program, *BLT instruction stalls* is the number of CPU reads and writes stalled after an RFU write is executing (this was the only type of hazard present). The next column shows the *Stalls per RFU instruction* and the last one shows the average *RFU instruction overlap* with CPU execution. Note that 128 is the operation latency for JPEG. We can see that for the JPEG encoder there is an overlap of approximately 11 instructions, and for the JPEG decoder an overlap of approximately 5 instructions.

From the previous analysis, we note that the data size does not affect the overall performance very much. Nevertheless, we can make an interesting analysis from Table 5-4. We can see that there is an approximate overlap of 11 instructions for the decoder. This means that when an RFU instruction is issued, 11 following instructions are also allowed to issue out-of-order because there are no data dependencies. Then, even if the RFU issue and operation latencies are improved (i.e. reduced) by new hardware technologies, the maximum improvement for this application will be observed if the configuration has a latency of 11 cycles. That is, any latency lower than 11 will not improve performance because the other 11 overlapping instructions will still need to be executed and the RFU will need to wait for them. The same will be observed for the JPEG decoder with a latency of 5 cycles. For the rest of the applications there is no overlap, so any improvement in the RFU latency will be reflected in the overall performance.

ADPCM shows a fairly large speedup from OneChip. This is because the application does not perform any data validation or other operations besides calling the encoder kernel. The data is simply read from standard input and written to standard output. The data is encoded in blocks of 1000 bytes at a time. It was expected for ADPCM to be the application with the most speedup due to Amdahl's Law [6], which states that the performance improvement to be gained from using a faster mode of execution is limited by the fraction of time the faster mode can be used. ADPCM's performance clearly depends on the size of the data. The larger the data, the less time the applications reads and writes data, and the most time the RFU executes instructions. The number of RFU instructions executed for the encoder and decoder are the

Application	Data size	RFU instructions (X)
ADPCM encode	Small	53
	Medium	148
	Large	728
ADPCM decode	Small	53
	Medium	148
	Large	728

Table 5-5. ADPCM RFU instructions

same for each data size. This is shown in Table 5-5. Since there are no BLT instruction stalls, this column is not shown in the table.

PEGWIT's performance also shows a dependence on the data size. However, different behavior is observed for the RFU and the out-of-order issue features. The RFU shows better performance with small data, while out-of-order issue shows a better performance improvement with larger data. The overall speedup with both features is greater as the input data is larger. This is because for the decoder, the application makes a number of RFU instruction calls independent from the data size, and even if the data size for each call is different, the latency is the same for every call. With the encoder, almost the same thing happens. The number of RFU instructions in PEGWIT is shown in Table 5-6. The BLT

Application	Data size	RFU instructions (X)
PEGWIT encode	Small	40060
	Medium	40572
	Large	41212
PEGWIT decode	Small	20288
	Medium	20288
	Large	20288

Table 5-6. PEGWIT RFU instructions

instruction stalls column is not shown in the table as there are none for this application.

MPEG2's performance is also data size dependent. In the case of the decoder, the larger the data, the greater the performance improvement. This applies for both, the RFU feature and the out-of-order issue feature. In the case of the encoder, the performance improvement is shown to be larger, as the frame sizes get larger. There is a a higher performance improvement between the tests with small and medium input data, which have a different frame size and almost the same number of frames, than between the tests with medium and large data, which have the same frame size and different number of frames. In the application, there are no BLT instruction stalls, hence this column is not shown in Table 5-7. The reason the simulator

Application	Data size	RFU instructions (X)
MPEG2 encode	Small	1152
	Medium	7920
	Large	21780
MPEG2 decode	Small	1152
	Medium	7920
	Large	21780

Table 5-7. MPEG2 RFU instructions

reports a much higher performance improvement on the decoder is because the most time consuming routines after the DCT are system calls, which are not simulated as such and are only passed to the underlying system for execution. Hence, in the simulated programs, the DCT takes much more execution time than the one obtained from the profiler, which does report the system calls execution time.

For all the applications, we can see that out-of-order issue by itself produces a big gain. Using an RFU still adds more speedup to the application. Speedup obtained from dynamic scheduling ranges from 1.60 up to 2.53. Speedup obtained from an RFU ranges from 1.13 up to 29.92. When using both at the same time, even when each technique limits the potential gain that the other can produce, the overall speedup is increased. Dynamic scheduling seems to be more effective with the applications, except for ADPCM, where the biggest gain comes from using the RFU. This leads us to think that for kernel oriented applications, it is better to use an RFU first and for the other applications it is better to use dynamic scheduling.

5.7 Summary

This chapter describes the benchmark applications that were used to evaluate the system's performance. The profiling information for each of the applications was presented, analyzed and discussed. Selected applications were modified for execution in Sim-OneChip and the results of the experiments were also presented and analyzed.

CHAPTER 6

Conclusion and Future Work

This chapter contains a conclusion of the work presented in this thesis. It also presents research directions and starting points for future work in the area.

6.1 Conclusion

In this work, the behavior of the OneChip architecture model was studied. Its performance was measured by executing several off-the-shelf software applications on a software model of the system. As part of the work, a simulator for the OneChip system that is capable of executing real applications, was developed. A software environment, which includes a C compiler and the required libraries to support the architecture, were also developed for programming the system.

Also as part of the work, a set of multimedia-type benchmarks suitable for the OneChip model was gathered. The applications were profiled, analyzed and modified for the OneChip architecture. The architecture was tested with the benchmark applications using different input data sizes. The results obtained confirm the performance improvement by the architecture on DSP-type applications.

From the work, a question arises whether the additional hardware cost of a complex structure, such as the Block Lock Table, is really necessary in reconfigurable processors. It has been proved that the concept of the BLT does accomplish its purpose, which is maintaining memory consistency when closely linking reconfigurable logic with memory and when parallel execution is desired between the CPU and the RFU. However, considering that only one of the four applications (i.e. JPEG) used in this research actually uses the BLT and takes advantage of it, we conclude that by removing it and simply make the CPU stall while the RFU is executing will not degrade performance significantly. In JPEG there is an average of 11 overlapping instructions, which is only 8.6% of the configuration operation latency slot of 128. If the RFU is used approximately 20% of the time in the JPEG encoder, the performance improvement by the overlapping is only 1.72%. This is a small amount compared to the performance improvement of dynamic scheduling, which is approximately 56% (i.e. 2.29 speedup). Hence, dynamic scheduling improves performance significantly only when used with relatively short operation delay instructions, as opposed to OneChip's RFU instructions, which have large operation delays.

Based on the four applications in this work, it appears to be that the number of contexts does not need to be large to achieve good performance improvement with an RFU. In these applications, only one context was used for each application and a considerable speedup was obtained. For some applications, a second context could have provided an increase in this improvement, but not as much as for the first context. This is because, based on our profiles, we could implement a different routine in a second context in the same way the first one was, but it would not be so frequently used. The question of how many contexts is an optimal number is still unanswered. From this work, it seems that no more than two contexts will provide a good speedup/hardware size ratio. In case an application used more than two, a configuration allocation algorithm implemented in the compiler could be used to reduce the number of context reconfigurations. This algorithm could be derived from an allocation algorithm such as graph coloring [26], and a configuration pre-fetch algorithm as described by Hauck [28].

Another question that arises from this work is whether the configurations are small enough to fit on today’s reconfigurable hardware, or if they can be even implemented. Hardware implementations of DSP structures done by other groups [56][57][58], and which have even been shown to outperform digital signal processors, have been proven to fit on Xilinx XC4036 devices[59] which have a maximum of 36,000 logic gates. Today’s FPGAs have more than 1 million system gates available.

We also conclude that dynamic scheduling is important to achieve good performance. By itself it produces a big gain for a number of applications. With kernel oriented applications, the gain obtained by an RFU is bigger, but with complete applications, the biggest gain is obtained from out-of-order issue and execution.

It becomes difficult to compare OneChip’s performance with other current reconfigurable systems. This is because there are no standard application benchmarks for reconfigurable processors. So far, performance reports by other groups [34][35][37][38][39][40][41] are done using applications kernels such as DCT, FIR filters, or even more complicated algorithms. In this work, we are focused on the architecture’s performance with complete applications. These results have not been reported yet for other systems.

As a final conclusion, it was previously demonstrated that the OneChip architecture can provide speedup for application kernels. It is now known that it can also provide an overall speedup for a wide range of already existing software applications.

6.2 Future work

There is still a big gap between the hardware and the software. To close this gap, further investigation is necessary in the area of compilers for reconfigurable processors. Specifically, a compiler designed for the OneChip architecture is needed to fully exploit it and to better estimate the advantages and disadvantages of the architecture’s features.

It is necessary to modify the simulator to properly simulate configuration loading latency. As for now, it is assumed that configurations are loaded in a single cycle or that they are simply pre-loaded. Since the configuration sizes are unknown, it becomes hard to estimate the loading latency. A proper way of reflecting in the simulations the configuration loading latency according to the configuration size is essential.

In this thesis, the porting of the applications was done by hand, based on profiling information. Developing a compilation system that allows automatic detection of structures suitable for the OneChip RFU, as well as generating the corresponding configuration and replacing the structure in the program, will allow further investigation of the optimal number of contexts for the RFU. It should also make an optimal use of the BLT by scheduling as many instructions in the RFU delay slot. With such a compiler, a wider range of applications can be tested and the architecture’s features further explored.

Based on the experience from this work, it might be advantageous to add BLT instructions to the processor. These instructions will be responsible for adding and removing entries to and from the BLT, instead of having the RFU instructions do it. This will allow the specification of block sizes that are not known at compile time, as they can be stored in registers, as opposed to being an immediate value in the instruction opcode. The compiler will be in charge of generating the correct instruction sequence for locking blocks before an RFU instruction is issued and unlocking it after it is done. This is a slightly different approach as the one presented in this thesis that requires more development of the compilation system.

It is also desirable to study the behavior of the architecture in a multiprocess operating system. Reconfigurable processors seem to be suitable for embedded applications. If this type of architecture is to be used on desktop or supercomputers, it must be tested with an operating system designed for it. Extra architectural features could be included to support the needs of operating systems.

As a final note, investigating the previously mentioned topics will lead to the development of a high performance reconfigurable system. After a complete study of the interactions between architecture, compiler, and operating systems for reconfigurable processors, one would be able to determine the best track to follow in the reconfigurable world.

References

[1] Wittig, R. D. *Onechip: An FPGA Processor with Reconfigurable Logic*, M. A. Sc. Thesis, Department of Electrical and Computer Engineering, University of Toronto, 1995.

[2] Wittig, R. D., and P. Chow. “OneChip: An FPGA Processor with Reconfigurable Logic”, *Proceedings of the IEEE Symposium on Field-Programmable Custom Computing Machines (FCCM'96)*, March 1996, pp. 126-135.

[3] Jacob, J. A. *Memory Interfacing for the OneChip Reconfigurable Processor*, M. A. Sc. Thesis, Department of Electrical and Computer Engineering, University of Toronto, 1998.

[4] Jacob, J. A., and P. Chow. “Memory Interfacing and Instruction Specification for Reconfigurable Processors”, *Proceedings of the 1999 ACM/SIGDA International Symposium on Field Programmable Gate Arrays (FPGA'99)*, February 1999, pp. 145-154.

[5] Patterson, D. A., and J. L. Hennessy. *Computer Organization and Design: The Hardware/Software Interface*, Second Edition, Morgan Kaufmann, San Francisco, CA, 1998.

[6] Hennessy, J. L., and D. A. Patterson. *Computer Architecture: A Quantitative Approach*, Second Edition, Morgan Kaufmann, San Francisco, CA, 1996.

[7] Hwang, K. *Advanced Computer Architecture: Parallelism, Scalability, Programmability.* McGraw-Hill, USA, 1993.

[8] Flynn, M. J. *Computer Architecture: Pipelined and Parallel Processor Design.* Jones and Bartlett Publishers, USA, 1995.

[9] Sima, D., T. Fountain and P. Kacsuk. Advanced Computer Architectures: A Design Space Approach. Addison-Wesley Publishing Company, USA, 1997.

[10] Brown, S. D., R. J. Francis, J. Rose and Z. G. Vranesic. *Field-Programmable Gate Arrays*, Kluwer Academic Publishers, Norwell, MA, 1992.

[11] Trimberger, S. M. *Field-Programmable Gate Array Technology*, Kluwer Academic Publishers, Norwell, MA, 1994.

[12] Chow, P., S. Seo, J. Rose, K. Chung, G. Páez-Monzón and I. Rahardja. “The Design of an SRAM-Based Field-Programmable Gate Array, Part I: Architecture”, IEEE Transactions on Very Large Scale Integration (VLSI) Systems, Vol. 7 No. 2, June 1999, pp. 191-197.

[13] Chow, P., S. Seo, J. Rose, K. Chung, G. Páez-Monzón and I. Rahardja. “The Design of an SRAM-Based Field-Programmable Gate Array, Part I: Circuit Design and Layout”, IEEE Transactions on Very Large Scale Integration (VLSI) Systems, Vol. 7 No. 3, Sept. 1999, pp. 321-330.

[14] DeHon, A. “DPGA-Coupled Microprocessors: Commodity ICs for the Early 21st Century”, *Proceedings of the IEEE Workshop on Field-Programmable Custom Computing Machines (FCCM’94)*, April 1994, pp. 31-39.

[15] Trimberger, S., D. Carberry, A. Johnson and J. Wong. “A Time-Multiplexed FPGA”, *Proceedings of the IEEE Symposium on Field-Programmable Custom Computing Machines (FCCM’97)*, April 1997, pp. 34-40.

[16] Scalera, S. M., and J. R. Vázquez “The Design and Implementation of a Context Switching FPGA”, *Proceedings of the IEEE Symposium on Field-Programmable Custom Computing Machines (FCCM’98)*, April 1998.

[17] Compton, K., and S. Hauck. “Configurable Computing: A Survey of Systems and Software”. Technical Report, Dept. of ECE, Northwestern University, 1999.

[18] Galloway, D. “The Transmogrifier C Hardware Description Language and Compiler for FPGAs”, *Proceedings of the IEEE Symposium on Field-Programmable Custom Computing Machines (FCCM'95)*, April 1995, pp. 136-144.

[19] Gokhale, M., and J. Stone.”NAPA C: Compiling for a Hybrid RISC/FPGA Architecture”, *Proceedings of the IEEE Symposium on Field-Programmable Custom Computing Machines (FCCM'95)*, April 1998, pp. 126-135.

[20] Cadambi, S., and S. Goldstein. “CPR: A Configuration Profiling Tool”, *Proceedings of the IEEE Symposium on Field-Programmable Custom Computing Machines (FCCM’99)*, April 1999, pp. 104-113.

[21] Callahan, T., and J. Wawrzyneck. “Instruction Level Parallelism for Reconfigurable Computing”, *Field-Programmable Logic and Applications, 8th International Workshop (FPL'98)*, Tallinn Estonia, September 1998.

[22] Wang, Q. *Automatic Compilation of Field-Programmable Custom Compute Accelerators*, Ph. D. Thesis, Department of Electrical and Computer Engineering, University of Toronto, 1999.

[23] Weinhardt, M., and W. Luk. “Pipeline Vectorization for Reconfigurable Systems”, *Proceedings of the IEEE Symposium on Field-Programmable Custom Computing Machines (FCCM’99)*, April 1999, pp. 52-62.

[24] Asanovic, K. *Vector Microprocessors*, Ph. D. Thesis, Computer Science Division, University of California at Berkeley, May 1998.

[25] Aho, A. V., R. Sethi and J. D. Ullman. *Compilers: Principles, Techniques and Tools*. Addison-Wesley Publishing Company, 1986.

[26] Muchnick, S. *Advanced Compiler Design and Implementation*, First Edition, Morgan Kaufmann, San Francisco, CA, 1997.

[27] Li, Z., K. Compton and S. Hauck. “Configuration Caching Management Techniques for Reconfigurable Computing”, *Proceedings of the IEEE Symposium on Field-Programmable Custom Computing Machines (FCCM’00)*, April 2000.

[28] Hauck, S. “Configuration Prefetch for Single Context Reconfigurable Coprocessors”, *Proceedings of the 1999 ACM/SIGDA International Symposium on Field Programmable Gate Arrays (FPGA'98)*, February 1998, pp. 65-74.

[29] Hauck, S., Z. Li and E. Schwabe. “Configuration Compression for the Xilinx XC6200 FPGA”, *Proceedings of the IEEE Symposium on Field-Programmable Custom Computing Machines (FCCM’98)*, April 1998, pp. 123-130.

[30] Xilinx, Inc. “XC6200 Field Programmable Gate Arrays”, Data sheet, October 1996.

[31] Li, Z., and S. Hauck. “Don’t Care Discovery for FPGA Configuration Compression”, *Proceedings of the 1999 ACM/SIGDA International Symposium on Field Programmable Gate Arrays (FPGA'99)*, February 1999, pp. 91-98.

[32] Laufer, R., R. R. Taylor and H. Schmit. “PCI-PipeRennch and the SwordAPI: A system for Stream-based Reconfigurable Computing”, *Proceedings of the IEEE Symposium on Field-Programmable Custom Computing Machines (FCCM’99)*, April 1999, pp. 200-208.

[33] Goldstein, S. C., H. Schmit, M. Moe, M. Budiu, S. Cadambi, R. R. Taylor and R. Laufer. “PipeRench: A Coprocessor for Streaming Multimedia Acceleration”, *Proceedings of the 26th. Annual International Symposium on Computer Architecture (ISCA’99)*, Los Alamitos, CA, 1999, pp. 28-39.

[34] Goldstein, S. C., H. Schmit, M. Budiu, S. Cadambi, M. Moe and R. R. Taylor. “Piperench: A reconfigurable Architecture and Compiler”, *Computer: Innovative Technology for Computer Professionals*, Volume 33, Number 4, IEEE Computer Society, April 2000, pp. 70-77.

[35] Hauck, S., T. W. Fry, M. M. Hosler, and J. P. Kao. “The Chimaera Reconfigurable Functional Unit”, *Proceedings of the IEEE Symposium on Field-Programmable Custom Computing Machines (FCCM’97)*, April 1997, pp. 87-96.

[36] Hauser, J. R., and J. Wawrzynek. “GARP: A MIPS Processor with a Reconfigurable Coprocessor”, *Proceedings of the IEEE Symposium on Field-Programmable Custom Computing Machines (FCCM’97)*, April 1997, pp. 12-21.

[37] Callahan, T. J., J. R. Hauser and J. Wawrzyneck. “The Garp Architecture and C Compiler”, *Computer: Innovative Technology for Computer Professionals*, Volume 33, Number 4, IEEE Computer Society, April 2000, pp. 62-69.

[38] Lu, G., H. Singh, M. Lee, N. Bagherzadeh, F. Kurdahi and E. M. C. Filho. “The MorphoSys Parallel Reconfigurable System”, *Proceedings of Euro-Par 99*, Toulouse, France, Sep 99.

[39] Miyamori, T., and K. Olukotun. “A Quantitative Analysis of Reconfigurable Coprocessors for Multimedia Applications”, *Proceedings of the IEEE Symposium on Field-Programmable Custom Computing Machines (FCCM’98)*, April 1998, pp. 2-11.

[40] Razdan, R., and M. D. Smith. “A High-Performance Microarchitecture with Hardware-Programmable Functional Units”, *International Symposium on Microarchitecture (MICRO 27)*, November 1994, pp.172-180.

[41] Haynes, S. D., J. Stone, P. Y. K. Cheung, W. Luk. “Video Image Processing with the Sonic Architecture”, Computer: Innovative Technology for Computer Professionals, Volume 33, Number 4, IEEE Computer Society, April 2000, pp. 50-57.

[42] Chameleon Systems, Inc. *CS2000 Reconfigurable Communications Processor Family Product Brief*, San Jose, CA, 2000.

[43] Triscend Corporation. *Triscend E5 Configurable System-on-Chip Family Product Description*, Mountain View, CA, 2000.

[44] Rupp, C. R., M. Languth, T. Garverick, E. Gomersall, H. Holt, J. M. Arnold and M. Gokhale. “The NAPA Adaptive Processing Architecture”, *Proceedings of the IEEE Symposium on Field-Programmable Custom Computing Machines (FCCM’98)*, April 1998, pp. 28-37.

[45] Annapolis Micro Systems, Inc. *Wildfire Reference Manual*, Annapolis, MD, 1998.

[46] Silberschatz, A., and P. B. Galvin. Operating System Comcepts, Fifth Edition, Addison-Wesley, USA, February 1998.

[47] Pai, V. S., P. Ranganathan and S. V. Adve. “RSIM: An Execution-Driven Simulator for ILP-Based Shared-Memory Multiproecessors and Uniprocessors”, *Proceedings of the Third Workshop on Computer Architecture Education*, February 1997.

[48] Burger, D., and T. M. Austin. “The SimpeScalar Tool Set, Version 2.0”, Technical report #1342, Computer Sciences Department, University of Wisconsin-Madison, June 1997.

[49] Austin, T. M. “A User’s and Hacker’s Guide to the SimpleScalar Architectural Research Tool Set”, Intel MicroComputer Research Labs, January, 1997.

[50] Kumar, S., L. Pires, S. Ponnuswamy, C. Nanavati, J. Golusky, M. Vojta, S. Wadi, D. Pandalai and H. Spaanenberg. “A Benchmark Suite for Evaluating Configurable Computing Systems - Status, Reflections, and Future Directions”, *Proceedings of the 2000 ACM/SIGDA International Symposium on Field Programmable Gate Arrays (FPGA’00)*, February 2000, pp. 126-134.

[51] Lee, C., M. Potkonjak and W. H. Mangione-Smith, “MediaBench: A Tool for Evaluating and synthesizing multimedia and communications systems”, *Procedings of the IEEE/ACM International Symposium on Microarchitecture (MICRO 30)*, December 1997, pp. 330-335.

[52] GNU’s Not Unix! - the GNU Project and the Free Software Foundation (FSF). On-line: http://www.gnu.org

[53] GNU gprof Manual- the GNU Project and the Free Software Foundation (FSF). On-line: http://www.gnu.org/manual/gprof-2.9.1/gprof.html

[54] The JPEG Committee Website. On-line: http://www.jpeg.org

[55] The MPEG Committee Website. On-line: http://www.mpeg.org

[56] Bergmann, N., Y. Chung and B. Gunther. “Efficient Implementation of the DCT on Custom Computers”, *Proceedings of the IEEE Symposium on Field-Programmable Custom Computing Machines (FCCM’97)*, April 1997, pp. 244-245.

[57] Walters, A., and P. Athanas. “A Scalable FIR Filter Using 32-bit Floating-Point Complex Arithmetic on a Configurable Computing Machine”, *Proceedings of the IEEE Symposium on Field-Programmable Custom Computing Machines (FCCM’98)*, April 1998, pp. 333-334.

[58] Lau, D., A. Schneider, M. Ercegovac and J. Villasenor. “FPGA-Based Structures for On-line FFT and DCT”, *Proceedings of the IEEE Symposium on Field-Programmable Custom Computing Machines (FCCM’99)*, April 1999, pp. 310-311.

[59] Xilinx, Inc. *The Programmable Logic Data Book*, San Jose CA, 1999.

APPENDIX A

oc-lib.h

```
/*
 *      oc-lib.h - OneChip library
 *
 * To use the reconfigurable unit in sim-onechip, the user needs to
 * include this library in the program. This library is not a circuit
 * library, it is only the user interface for the RFU. It includes
 * macros for defining configuration addresses, using and pre-loading
 * configurations. These macros will generate the corresponding
 * annotated instructions for the simulator.
 *
 * Instructions that target the reconfigurable unit are disguised as
 * annotated instructions. Without the annotation, they are treated as
 * regular instructions; with the annotation they become instructions
 * that target the reconfigurable unit. This way, there is no need to
 * modify the compiler.
 *
 */

#ifndef OC_LIB_H
#define OC_LIB_H

/* rec_2addr(func, src_addr, dst_addr, src_size, dst_size)
 *
 *      func     - FPGA function
 *      src_addr - source address
 *      dst_addr - destination address
 *      src_size - source block size
 *      dst_size - destination block size
 *
 * This macro is used to build an instruction that will be assembled
 * by gas as the reconfigurable instruction.
 *
 * The reconfigurable instruction is disguised as an annotated andi
 * instruction. Without the annotation, it is a regular andi; with the
 * annotation it becomes the reconfigurable instruction. This way,
 * there is no need to modify the compiler.
 *
 * Here, the compiler will use the registers that contain the source
 * and destination addresses (src_reg and dst_reg) to build the
 * annotated instruction that will later be assembled.
 *
 * The function and block sizes are combined and put into the "imm"
 * field of the annotated and instruction. These three need to be
 * immediate values (i.e. NO variables).
 *
 * The instruction format for the reconfigurable instruction should be:
 * +------------+-----------+-------+-------+----+--------+--------+
 * | Annotation |   Opcode   |src_reg|dst_reg|func|src_size|dst_size|
 * |    (16)    |    (16)    |  (8)  |  (8)  |(4) |  (6)   |  (6)   |
 * +------------+-----------+-------+-------+----+--------+--------+
 *                                          <--------IMM(16)------->
 *
 */

#define rec_2addr(func, src_addr, dst_addr, src_size, dst_size)\
__asm__ volatile("andi/a  %0,%1,%2# Reconfigurable 2-addr inst"\
 :\
 : "r"(dst_addr)\
 , "r"(src_addr)\
 , "g"((func<<12)|(src_size<<6)|(dst_size))\
);

/* rec_3addr(func, src1_addr, src2_addr, dst_addr, blk_size)
 *
 *      func      - FPGA function
 *      src1_addr - source-1 address
 *      src2_addr - source-2 address
 *      dst_addr  - destination address
 *      blk_size  - block size
 *
 * This macro is used to build an instruction that will be assembled
 * by gas as the reconfigurable 3-address instruction.
 *
 * The reconfigurable instruction is disguised as an annotated and
 * instruction. Without the annotation, it is a regular and; with
 * the annotation it becomes the reconfigurable instruction. This way,
 * there is no need to modify the compiler.
 *
 * Here, the compiler will use the registers that contain the source
 * and destination addresses to build the annotated instruction that
 * will later be assembled.
 *
 * The function and block size are combined and put into the 10 msb
 * (most signficant bits) of the “annotation” field of the annotated
 * and instruction. This is done because there is no instruction that
 * uses three addresses and an immediate value that we could use to
 * put the block size and fucntion into. These two need to be immediate
 * values (i.e. NO variables).
 *
 * The instruction will be assembled as this:
 * +----+--------+-----+-----+--------+--------+--------+----------+
 * |func|blk_size|Annot|Opcode|src1_reg|dest_reg|src2_reg| NOT USED |
 * | (4)|   (6)  | (6) | (16) |   (8)  |   (8)  |   (8)  |   (8)    |
 * +----+--------+-----+-----+--------+--------+--------+----------+
 * <----ANNOTATION----->
 *
 */

#define rec_3addr(func, src1_addr, src2_addr, dest_addr, blk_size)\
__asm__ volatile(“and/a/15:6(%3) %2,%1,%0# Reconfigurable 3-addr inst”\
 : /* no outputs */\
 : “r”(dest_addr)\
 , “r”(src1_addr)\
 , “r”(src2_addr)\
 , “g”((func<<6)|(blk_size))\
                );

/* oc_configAddress(func, addr)
 *
 *      func      - FPGA function
 *      addr      - memory address where FPGA configuration is located
 *
 * This macro is used to build an instruction that will be assembled
 * by gas as the “configuration address” instruction for the
 * reconfigurable unit. The instruction will associate the function
 * “func” with the address “addr” where the FPGA configuration should
 * be located.
 *
 * The “configuration address” instruction is disguised as an
 * annotated andi instruction. For it we use the annotation “/b”.
 *
 * Here, the compiler will use the register that contains the address
 * to be assigned to the function in order to build the annotated
 * instruction that will later be assembled.
 *
 * The function is put into the “imm” field of the annotated and
 * instruction, as it is an immediate value.
 *
 * The instruction format is:
 * +------------+------------+--------+--------+----+--------------+
 * | Annotation |   Opcode   |addr_reg|NOT USED|func|   NOT USED    |
 * |    (16)    |    (16)    |   (8)  |   (8)  |(4) |     (12)      |
 * +------------+------------+--------+--------+----+---------------+
 *                            <-src_reg>        <-------IMM(16)------>
 *
 */

#define oc_configAddress(func, addr)\
__asm__ volatile("andi/b  $0,%0,%1# configAddress instruction"\
 : /* no outputs */\
 : "r"(addr)\
 , "g"(func<<12)\
 );

/* oc_preLoad(func)
 *
 *      func      - FPGA function
 *
 * This macro is used to biuld an instruction that will be assembled
 * by gas as the "pre-load configuration" instruction for the
 * reconfigurable unit. The instruction will pre-load the
 * configuration located in the memory address currently associated
 * with the function "func".
 *
 * The "pre-load configuration" instruction is disguised as an annotated
 * andi instruction. For it we use the annotation "/a/b".
 *
 * The function and block sizes are combined and put into the "imm"
 * field of the annotated and instruction. These three need to be
 * immediate values (i.e. NO variables).
 *
 * The instruction format is:
 * +------------+------------+-----------------+----+---------------+
 * | Annotation |   Opcode   |     NOT USED    |func|    NOT USED   |
 * |    (16)    |    (16)    |       (16)      |(4) |      (12)     |
 * +------------+------------+-----------------+----+---------------+
 *                                              <-------IMM(16)------>
 *
 */

#define oc_preLoad(func)      \
__asm__ volatile("andi/a/b $0,$0,%0# preLoad instruction"        \
 : /* no outputs */      \
 : "g"(func<<12)      \
 );

/* oc_encodeSize(size)
 *
 * DESCRIPTION HERE!
 * Can this be changed for a table?????!!!!!!!!!!!
 *
 *
 */

#define oc_encodeSize(size)\
                     (size == 2 ? (unsigned char) 0 :\
      size == 4 ? (unsigned char) 1 :\
      size == 8 ? (unsigned char) 2 :\
      size == 16 ? (unsigned char) 3 :\
      size == 32 ? (unsigned char) 4 :\
      size == 64 ? (unsigned char) 5 :\
      size == 128 ? (unsigned char) 6 :\
      size == 256 ? (unsigned char) 7 :\
      size == 512 ? (unsigned char) 8 :\
      size == 1024 ? (unsigned char) 9 :\
      size == 2048 ? (unsigned char) 10 :\
      size == 4096 ? (unsigned char) 11 :\
      size == 8192 ? (unsigned char) 12 :\
      size == 16384 ? (unsigned char) 13 :\
      size == 32762 ? (unsigned char) 14 :\
      size == 65536 ? (unsigned char) 15 :\
      printf(“oc-lib: ERROR in block size!!!”)\
     )

#endif /* OC_LIB_H */
```

APPENDIX B

fpga.conf

```
/*
 *      fpga.conf - fpga configuration
 *
 * FPGA configurations for sim-onechip are defined in this file. The
 * makefile will search for this file in the simulator directory when
 * compiling the simulator.
 *
 * To define a new configuration you should use:
 *
 *      DEFCONF(<addr>, <oplat>, <issuelat>,
 *              {
 *                <EXPR>
 *              }
 *      )
 *
 *      <addr>     - Configuration address: Location of the configuration
 *                   bits in memory.
 *      <oplat>    - Operation latency: Cycles until result is ready for
 *                   use.
 *      <issuelat> - Issue latency: Cycles before another operation can be
 *                   issued on this resource.
 *      <EXPR>     - Expression.
 *
 * NOTE: See "onechip.def" for more details on how to define an expression.
 *
 * The following predefined macros are available for use in DEFCONF()
 * expressions to access the value of RFU instruction operand field
 * values:
 *
 *       OC_FUNC    FPGA function (4 bits)
 *       OC_SR      Source register (8 bits)
 *       OC_DR      Destination register (8 bits)
 *       OC_SBS     Source block size (6 bits)
 *       OC_DBS     Destination block size (6 bits)
 *
 * For 3-operand instructions, the available macros are the following:
 *
 *       OC_3A_FUNC FPGA function (4 bits)
 *       OC_3A_S1R  Source1 register (8 bits)
 *       OC_3A_S2R  Source2 register (8 bits)
 *       OC_3A_DR   Destination register (8 bits)
 *       OC_3A_BS   Block size (6 bits)
 *
 * Some configuration examples are shown here.
 *
 */

/* configuration #0 */
/* This configuration is for a 2-operand instruction. It doubles the
   values of the source array into the destination array
  ( dst[i] = 2 * src[i] ). */
DEFCONF(0x7FFFC000, 12,12,
{
  int oc_index;/* temp for indexing */
  unsigned char oc_byte;/* temp for storing bytes */

  for (oc_index = 0; oc_index <= OC_MASK(OC_SBS); oc_index++)
    {
      oc_byte = READ_UNSIGNED_BYTE(GPR(OC_SR) + oc_index);
      WRITE_BYTE(oc_byte << 1, GPR(OC_DR) + oc_index);
    }
}
)

/* configuration #1 */
/* This configuration is for a 2-operand instruction. It adds the
   values of the source and destination arrays and stores them into
   the destination array ( dst[i] = dst[i] * src[i] ). */
DEFCONF(0x7FFFC001, 12, 12,
{
  int oc_index;/* temp for indexing */
  unsigned char oc_byte;/* temp for storing bytes */

  for (oc_index = 0; oc_index <= OC_MASK(OC_SBS); oc_index++)
    {
      oc_byte = READ_UNSIGNED_BYTE(GPR(OC_SR) + oc_index);
      oc_byte += READ_UNSIGNED_BYTE(GPR(OC_DR) + oc_index);
      WRITE_BYTE(oc_byte, GPR(OC_DR) + oc_index);
    }
}
)
```

```
/* configuration #2 */
/* This configuration is for a 3-operand instruction. It is used
   for a fir filter program. */
DEFCONF(0x7FFFC002, 24, 24,
{
  int oc_index;/* temp for indexing */
  unsigned int oc_word;       /* temp for storing words */
  unsigned int oc_result;/* temp for storing the result */

  oc_result = 0;
  for (oc_index = 0; oc_index <= OC_MASK(OC_3A_BS); oc_index++)
    {
      oc_word = READ_WORD(GPR(OC_3A_S1R) + (4 * oc_index));
      oc_word *= READ_WORD(GPR(OC_3A_S2R) + (4 * oc_index));
      oc_result += oc_word;
    }
  WRITE_WORD(oc_result, GPR(OC_3A_DR));
}
)

#ifdef OC_HELPFUNC
/* This section is used to write any C code functions that may be used in
   the configurations */


#endif /* OC_HELPFUNC */
```

APPENDIX C

testing files

```
.file1 “sit01.s”

 # GNU C 2.6.3 [AL 1.1, MM 40, tma 0.1] SimpleScalar running sstrix compiled
            by GNU C

 # Cc1 defaults:
 # -mgas -mgpOPT

 # Cc1 arguments (-G value = 8, Cpu = default, ISA = 1):
 # -quiet -dumpbase -o

gcc2_compiled.:
__gnu_compiled_c:
.rdata
.align2
$LC0:
.ascii“a[1] = %d\n\000”
.text
.align2
.globlmain

.text

.loc1 5
.entmain
main:
.frame$fp,88,$31# vars= 64, regs= 2/0, args= 16, extra= 0
.mask0xc0000000,-4
.fmask0x00000000,0
subu$sp,$sp,88
sw$31,84($sp)
sw$fp,80($sp)
move$fp,$sp
jal__main
```

```
li$2,0x00000005# 5
sb$2,54($fp)
lbu$2,54($fp)
sb$2,17($fp)
addu$2,$fp,48
addu$3,$fp,16

 sb$8,20($fp)# Store to SRC_BLOCK

 #APP
andi/a  $2,$3,260# Reconfigurable instruction
 #NO_APP

nop
nop
nop
        nop
        nop
        nop
        nop
        nop
nop
nop
nop
nop

lbu$2,54($fp)
sb$2,20($fp)
lbu$2,25($fp)
sb$2,54($fp)
lbu$2,17($fp)
la$4,$LC0
move$5,$2
jalprintf
$L1:
move$sp,$fp# sp not trusted here
lw$31,84($sp)
lw$fp,80($sp)
addu$sp,$sp,88
j$31
.endmain
```

```
.file1 “sit02.s”

 # GNU C 2.6.3 [AL 1.1, MM 40, tma 0.1] SimpleScalar running sstrix compiled
            by GNU C

 # Cc1 defaults:
 # -mgas -mgpOPT

 # Cc1 arguments (-G value = 8, Cpu = default, ISA = 1):
 # -quiet -dumpbase -o

gcc2_compiled.:
__gnu_compiled_c:
.rdata
.align2
$LC0:
.ascii“a[1] = %d\n\000”
.text
.align2
.globlmain

.text

.loc1 5
.entmain
main:
.frame$fp,88,$31# vars= 64, regs= 2/0, args= 16, extra= 0
.mask0xc0000000,-4
.fmask0x00000000,0
subu$sp,$sp,88
sw$31,84($sp)
sw$fp,80($sp)
move$fp,$sp
jal__main
li$2,0x00000005# 5
sb$2,54($fp)
lbu$2,54($fp)
sb$2,17($fp)
addu$2,$fp,48
addu$3,$fp,16
 #APP
andi/a  $2,$3,260# Reconfigurable instruction
 #NO_APP
 lbu$8,50($fp)# Load from DST_BLOCK

        nop
        nop
        nop
        nop
        nop
        nop
        nop
        nop
        nop
        nop
        nop
        nop

lbu$2,54($fp)
sb$2,20($fp)
lbu$2,25($fp)
sb$2,54($fp)
lbu$2,17($fp)
la$4,$LC0
move$5,$2
jalprintf
$L1:
move$sp,$fp# sp not trusted here
lw$31,84($sp)
lw$fp,80($sp)
addu$sp,$sp,88
j$31
.endmain
```

```
.file1 “sit03.s”

 # GNU C 2.6.3 [AL 1.1, MM 40, tma 0.1] SimpleScalar running sstrix compiled
            by GNU C

 # Cc1 defaults:
 # -mgas -mgpOPT

 # Cc1 arguments (-G value = 8, Cpu = default, ISA = 1):
 # -quiet -dumpbase -o

gcc2_compiled.:
__gnu_compiled_c:
.rdata
.align2
$LC0:
.ascii“a[1] = %d\n\000”
.text
.align2
.globlmain

.text

.loc1 5
.entmain
main:
.frame$fp,88,$31# vars= 64, regs= 2/0, args= 16, extra= 0
.mask0xc0000000,-4
.fmask0x00000000,0
subu$sp,$sp,88
sw$31,84($sp)
sw$fp,80($sp)
move$fp,$sp
jal__main
li$2,0x00000005# 5
sb$2,54($fp)
lbu$2,54($fp)
sb$2,17($fp)
addu$2,$fp,48
addu$3,$fp,16

 lbu$8,54($fp)# Load from DST_BLOCK

 #APP
andi/a  $2,$3,260# Reconfigurable instruction
 #NO_APP

        nop
        nop
        nop
        nop
        nop
        nop
        nop
        nop
        nop
        nop
        nop
        nop

lbu$2,54($fp)
sb$2,20($fp)
lbu$2,25($fp)
sb$2,54($fp)
lbu$2,17($fp)
la$4,$LC0
move$5,$2
jalprintf
$L1:
move$sp,$fp# sp not trusted here
lw$31,84($sp)
lw$fp,80($sp)
addu$sp,$sp,88
j$31
.endmain
```

```
.file1 “sit04.s”

 # GNU C 2.6.3 [AL 1.1, MM 40, tma 0.1] SimpleScalar running sstrix compiled
            by GNU C

 # Cc1 defaults:
 # -mgas -mgpOPT

 # Cc1 arguments (-G value = 8, Cpu = default, ISA = 1):
 # -quiet -dumpbase -o

gcc2_compiled.:
__gnu_compiled_c:
.rdata
.align2
$LC0:
.ascii“a[1] = %d\n\000”
.text
.align2
.globlmain

.text

.loc1 5
.entmain
main:
.frame$fp,88,$31# vars= 64, regs= 2/0, args= 16, extra= 0
.mask0xc0000000,-4
.fmask0x00000000,0
subu$sp,$sp,88
sw$31,84($sp)
sw$fp,80($sp)
move$fp,$sp
jal__main
li$2,0x00000005# 5
sb$2,54($fp)
lbu$2,54($fp)
sb$2,17($fp)
addu$2,$fp,48
addu$3,$fp,16
 #APP
andi/a  $2,$3,260# Reconfigurable instruction
 #NO_APP

 sb$8,20($fp)# Store to SRC_BLOCK

        nop
        nop
        nop
        nop
        nop
        nop
        nop
        nop
        nop
        nop
        nop
        nop

lbu$2,54($fp)
sb$2,20($fp)
lbu$2,25($fp)
sb$2,54($fp)
lbu$2,17($fp)
la$4,$LC0
move$5,$2
jalprintf
$L1:
move$sp,$fp# sp not trusted here
lw$31,84($sp)
lw$fp,80($sp)
addu$sp,$sp,88
j$31
.endmain
```

```
.file1 “sit05.s”

 # GNU C 2.6.3 [AL 1.1, MM 40, tma 0.1] SimpleScalar running sstrix compiled
            by GNU C

 # Cc1 defaults:
 # -mgas -mgpOPT

 # Cc1 arguments (-G value = 8, Cpu = default, ISA = 1):
 # -quiet -dumpbase -o

gcc2_compiled.:
__gnu_compiled_c:
.rdata
.align2
$LC0:
.ascii“a[1] = %d\n\000”
.text
.align2
.globlmain

.text

.loc1 5
.entmain
main:
.frame$fp,88,$31# vars= 64, regs= 2/0, args= 16, extra= 0
.mask0xc0000000,-4
.fmask0x00000000,0
subu$sp,$sp,88
sw$31,84($sp)
sw$fp,80($sp)
move$fp,$sp
jal__main
li$2,0x00000005# 5
sb$2,54($fp)
lbu$2,54($fp)
sb$2,17($fp)
addu$2,$fp,48
addu$3,$fp,16

 sb$8,50($fp)# Store to DST_BLOCK

 #APP
andi/a  $2,$3,260# Reconfigurable instruction
 #NO_APP

        nop
        nop
        nop
        nop
        nop
        nop
        nop
        nop
        nop
        nop
        nop
        nop

lbu$2,54($fp)
sb$2,20($fp)
lbu$2,25($fp)
sb$2,54($fp)
lbu$2,17($fp)
la$4,$LC0
move$5,$2
jalprintf
$L1:
move$sp,$fp# sp not trusted here
lw$31,84($sp)
lw$fp,80($sp)
addu$sp,$sp,88
j$31
.endmain
```

```
.file1 “sit06.s”

 # GNU C 2.6.3 [AL 1.1, MM 40, tma 0.1] SimpleScalar running sstrix compiled
            by GNU C

 # Cc1 defaults:
 # -mgas -mgpOPT

 # Cc1 arguments (-G value = 8, Cpu = default, ISA = 1):
 # -quiet -dumpbase -o

gcc2_compiled.:
__gnu_compiled_c:
.rdata
.align2
$LC0:
.ascii“a[1] = %d\n\000”
.text
.align2
.globlmain

.text

.loc1 5
.entmain
main:
.frame$fp,88,$31# vars= 64, regs= 2/0, args= 16, extra= 0
.mask0xc0000000,-4
.fmask0x00000000,0
subu$sp,$sp,88
sw$31,84($sp)
sw$fp,80($sp)
move$fp,$sp
jal__main
li$2,0x00000005# 5
sb$2,54($fp)
lbu$2,54($fp)
sb$2,17($fp)
addu$2,$fp,48
addu$3,$fp,16
 #APP
andi/a  $2,$3,260# Reconfigurable instruction
 #NO_APP

 sb$8,50($fp)# Store to DST_BLOCK

        nop
        nop
        nop
        nop
        nop
        nop
        nop
        nop
        nop
        nop
        nop
        nop

lbu$2,54($fp)
sb$2,20($fp)
lbu$2,25($fp)
sb$2,54($fp)
lbu$2,17($fp)
la$4,$LC0
move$5,$2
jalprintf
$L1:
move$sp,$fp# sp not trusted here
lw$31,84($sp)
lw$fp,80($sp)
addu$sp,$sp,88
j$31
.endmain
```