Abstract

ŞAHİN, İBRAHİM. A Compilation Tool for Automated Mapping of Algorithms onto FPGA-Based Custom Computing Machines. (Under the directions of Dr. Clay S. Gloster and Dr. Winser E. Alexander).

Adaptive computing, also known as Reconfigurable Computing (RC), is a field that combines hardware and software data processing platforms. RC systems combine the flexibility of General Purpose Processors (GPP) with the speed of application specific processors [1, 2]. In a typical reconfigurable computer, computationally intensive portions of algorithms are executed on Field Programmable Gate Arrays (FPGA) for enhanced performance.

Although RC systems offer significant performance advantages over GPPs, they have a few disadvantages. RC systems require more application development time than GPPs. Also, RC system designers need to be knowledgeable in the areas of hardware and software system design. Since each application is different in terms of data inputs, outputs, and the method of processing data, designers are required to design a specific RC implementation for each specific problem.

Our major contribution in this research is the development of a design automation tool called the Reconfigurable Computing Compilation Tool (RCCT) to address the problems mentioned above. In addition, this tool was designed to automate the process of mapping applications onto RC systems, and to provide the potential performance benefits of RC systems to typical software programmers. The final version of the tool contains four components: The RC Compiler, the Module Library, the Loader and the

Simulator. **Our contributions also includes a novel assembly language instruction set for the modules and a session file format (a new assembly language program format for RC systems).**

The tool was tested on several applications to demonstrate its effectiveness. Among the selected applications were matrix multiplication, and some image processing algorithms such as 3-D Image correlation. We compared the execution times of the applications when they were run on different GPPs to different RC configurations to demonstrate the tool's effectiveness.

Our results showed that the tool is able to enhance the performance of the applications by mapping portions of them to the RC systems. Simulations with the tool showed that when the user applications are mapped to the RC systems, significant speedups (around 10 times to 100 times) can be attained for the mapped sections of the applications. We also noticed that the design and implementation time of the RC versions of the applications were reduced significantly. With the tool, the RC versions of the applications were developed, in a matter of a few hours. No special skills are needed to map applications to the RC systems using RCCT if the required hardware modules are readily available.

A Compilation Tool for Automated Mapping of Algorithms onto FPGA-Based Custom Computing Machines

by
İbrahim Şahin

A dissertation submitted to the Graduate Faculty of
North Carolina State University
in partial fulfillment of the
requirements for the Degree of
Doctor of Philosophy

**Department of
Electrical and Computer Engineering**

Raleigh, NC

August 2002

Electrical Engineering

Approved By:

Co-chair, Dr. Clay S. Gloster	Co-chair, Dr. Winser E. Alexander
Member, Dr. Paul D. Franzon	Member, Dr. Gregory T. Byrd

Dedication

This thesis is dedicated to my wife,

Şeyma Şahin

and my son,

Mehmet S. Şahin

Biography

İbrahim Şahin was born on December 14, 1970 in Ankara, Türkiye. He attended Gazi University in Ankara, Türkiye where he obtained the degree of Bachelor of Science in Electronics and Computer Education in June 1993. Immediately after graduation, he started working at a high school as a teacher. After teaching for six months at Isparta Technical and Vocational High School, he passed a nation wide competitive qualification examination organized by The Higher Educational Council of Turkey (YÖK) and received a university sponsored grant to pursue on M.S. and Ph.D. in an overseas country. After the examination he became a research assistant at Abant Izzet Baysal University in Bolu, Türkiye. In 1995, he moved to Norfolk, Virginia and started his M.S. study at Old Dominion University. He received his M.S degree from Old Dominion University in December 1997. He was admitted to the Ph.D. program in Electrical and Computer Engineering at North Carolina State University in the Fall of 1997. His hobbies include traveling, hiking, and skydiving.

Acknowledgments

I would like to express my sincere gratitude to my advisors, Dr Clay S. Gloster and Dr. Winser E. Alexander for their invaluable guidance, encouragement, and support during this lengthly work. This dissertation would not have been possible without their knowledge, wisdom and directions.

I would like to express my appreciation to the other members of my Ph.D. committee, Dr. Gregory T. Byrd, Dr. Paul F. Franzon and Dr. Albert J. Shih for their suggestions, comments and beneficial discussions.

In addition, I would like to thank the BDFA group members particularly Christopher C. Doss and An-Te Deng for their help with knowledge and suggestions.

I also thank NASA for their financial support in this research project*. My special thanks goes to Abant Izzet Baysal University of Türkiye for their support through my M.S. and Ph.D. study in the U.S.A.

* This research was supported by NASA under contract AIST-0016-0044.

Contents

List of Tables

List of Figures

Chapter 1

Introduction

Adaptive computing, also known as Reconfigurable Computing (RC), is a field that combines hardware and software data processing platforms. RC systems typically include a general purpose processor and one or more Field Programmable Gate Array (FPGA) devices. These systems combine the flexibility of general purpose processors with the speed of application specific processors [1, 2]. In a typical reconfigurable computer, computationally intensive portions of algorithms are executed on FPGA devices for enhanced performance. A well-designed and utilized adaptive computer could yield 10 times to 100 times improvement in execution time over conventional "software only" computers based on a general purpose processor based.

Several applications have been mapped to reconfigurable computers to demonstrate the viability of RC systems. Applications mapped to these systems include image processing algorithms [4, 5, 6, 7], genetic optimization algorithms [8], and pattern recognition [9]. In most cases, the reconfigurable computing system provided the smallest published execution time for these applications.

Each FPGA device contains a finite set of hardware resources. Therefore, not all applications can be efficiently mapped to these systems. This is especially true for applications for which floating-point (FP) arithmetic operations are needed due to the large amount of resources required by floating-point units. As a result, application developers typically either avoided implementing these applications in RC systems or converted the floating-point operations to fixed-point operations to reduce the amount of hardware resources required [10].

Recent advances in FPGA technology have opened new doors for developers. Both size and clock speed of FPGA devices have increased significantly. With today's technology, more than a million logic gates can be implemented on a single FPGA device, and can be clocked at speeds greater than 100 MHz. These improvements give us the opportunity to implement more complex applications, including those that require floating-point arithmetic.

Although RC systems offer significant performance advantages over general purpose processors, they have a few disadvantages. RC systems require more application development time than general purpose processors, but significantly less than developing an application specific integrated circuit. Also, RC system designers need to be knowledgeable in the areas of hardware and software system design. Since each application is different in terms of data inputs, outputs, and the method of processing data, designers are required to design a specific RC implementation for each specific problem.

These disadvantages of RC systems suggest that there is a need for a tool to automate the design and implementation process of RC applications. Such a tool could reduce both the design and implementation time of the applications greatly and it could eliminate the need for the system designer. With such a tool, an ordinary software programmer with little or no hardware knowledge could easily develop RC applications. Our primary contribution in this research is to develop this RC design automation tool that can speed up applications by automatically mapping them to RC systems.

1.1 Goals of the Proposed Research

The goals of this research project are to:

- **Automate the RC application development process:** In this research, a compilation tool that maps C/C++ programs onto RC systems is presented.

- **Reduce the application development time:** RC systems require more application development time than general purpose processors. Our goal is to reduce application development time significantly. We anticipate that the development time can be reduced from several weeks to a few hours when the tool is used.

- **Increase the performance of the application:** It is well known that applications can achieve significant speedup when they are mapped to RC systems. The goal of this research is to achieve at least an order of magnitude speedup when the applications are mapped to the RC system using the tool.

- **Provide the potential performance benefits of RC systems to typical software programmers:** As mentioned in the previous section, developing an RC application requires designers who are knowledgeable in the areas of hardware and software system design. An ordinary software programmer with little or no hardware knowledge could be able to develop RC applications by utilizing the tool.

Our main contribution in this research is development of a design automation tool called Reconfigurable Computing Compilation Tool (RCCT) to accomplish our goals listed above. The tool consists of four components: The Compiler, the Loader, the Simulator, and the Hardware Module Library. Additionally, a novel assembly language instruction set for the hardware modules and a session file format (a new assembly language program format) for RC systems were developed.

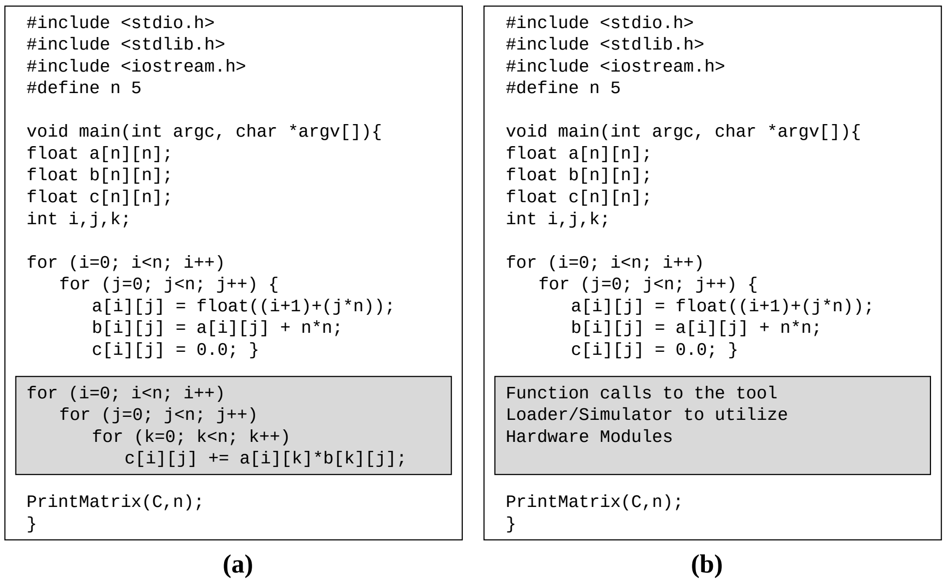
The tool was tested on several applications to demonstrate its effectiveness. Among the selected applications are matrix multiplication, and a few image processing algorithms such as 3-D image correlation. We compared the execution times of the applications when they were run on different GPPs and when they were mapped to different RC configurations to demonstrate the tool's effectiveness.

Our results showed that the tool is able to enhance the performance of the applications by mapping portions of them to the RC systems. The tool's Simulator showed that when the user applications are mapped to the RC systems, significant speedups (around 10 times to 100 times) can be attained for the mapped sections of the applications. We also noticed that the design and implementation time of the RC versions of the applications were reduced significantly. With the tool, in a matter of a few hours, RC versions of the applications were created. It is also observed that with RCCT, no special skills are needed to map applications to RC systems if the required hardware modules are available. The following section presents an overview of the tool.

1.2 The Tool: An Overview

The main contribution of this research is the development of a tool that automates the application design and implementation process for reconfigurable systems. The tool targets sections of the applications that have the greatest potential to reduce execution time on an RC system. Basically, it targets single or nested `for` loops due to the fact that `for` loops are the most frequently used loops to perform computationally complex vector operations. The final version of the tool contains four components: The RCCT Compiler, the Hardware Module Library, the Loader and the Simulator. Additionally, an assembly language instruction set for the modules and a session file format have been developed.

The RCCT Compiler performs several important tasks. As shown in Figure 1.1, it takes the original source code and produces a modified source code that is compiled by a traditional programming language compiler.



```
#include <stdio.h>
#include <stdlib.h>
#include <iostream.h>
#define n 5

void main(int argc, char *argv[]){
float a[n][n];
float b[n][n];
float c[n][n];
int i,j,k;

for (i=0; i<n; i++)
   for (j=0; j<n; j++) {
      a[i][j] = float((i+1)+(j*n));
      b[i][j] = a[i][j] + n*n;
      c[i][j] = 0.0; }

for (i=0; i<n; i++)
   for (j=0; j<n; j++)
      for (k=0; k<n; k++)
         c[i][j] += a[i][k]*b[k][j];

PrintMatrix(C,n);
}
```

(a)

```
#include <stdio.h>
#include <stdlib.h>
#include <iostream.h>
#define n 5

void main(int argc, char *argv[]){
float a[n][n];
float b[n][n];
float c[n][n];
int i,j,k;

for (i=0; i<n; i++)
   for (j=0; j<n; j++) {
      a[i][j] = float((i+1)+(j*n));
      b[i][j] = a[i][j] + n*n;
      c[i][j] = 0.0; }

Function calls to the tool
Loader/Simulator to utilize
Hardware Modules

PrintMatrix(C,n);
}
```

(b)

Figure 1.1: Sample source code: a) Before compilation with the RCCT Compiler b) After compilation with the RCCT Compiler.

The Module Library includes a set of hardware modules that perform basic mathematical operations on vectors. Each module is a previously placed and routed configuration file for a specific FPGA device. These modules are optimized for both area and speed.

The Loader works as an interface between the application and the RC system. It executes the candidate portions of the application on the RC system. It allows the application to utilize the hardware modules on one or more FPGA devices when the execution flow reaches the modified sections of the application.

Before executing a modified section of the user application on an RC system, the Loader checks the availability of an RC system and an appropriate hardware module. It checks for the existence of an RC system via the system's Application programming Interface (API) and attempts to locate a configuration file for the hardware module. When an appropriate hardware module or an RC system is not available, the Loader activates the Simulator to execute the same section on the host computer as if it were being executed on an FPGA device. The Simulator returns the estimated module execution time.

The tool works in two phases, the compilation phase and the execution phase. Figure 1.2 demonstrates the compilation phase. In this phase, the RCCT Compiler takes the original source code (nested loops) and modifies it. It detects the computationally complex portions of the source code and replaces them with function calls to the Loader. The sections that do not require high performance are not modified and are executed on the general purpose processor. While compiling the code, the RCCT Compiler also generates one or more session files for each modified section. Session files include an assembly language program for the hardware modules to execute and commands for the Loader to initialize the FPGA/memory.

After the application code is compiled by the RCCT Compiler, it is recompiled by a traditional programming language compiler and the executable code for the application that includes function calls to the Loader is produced.

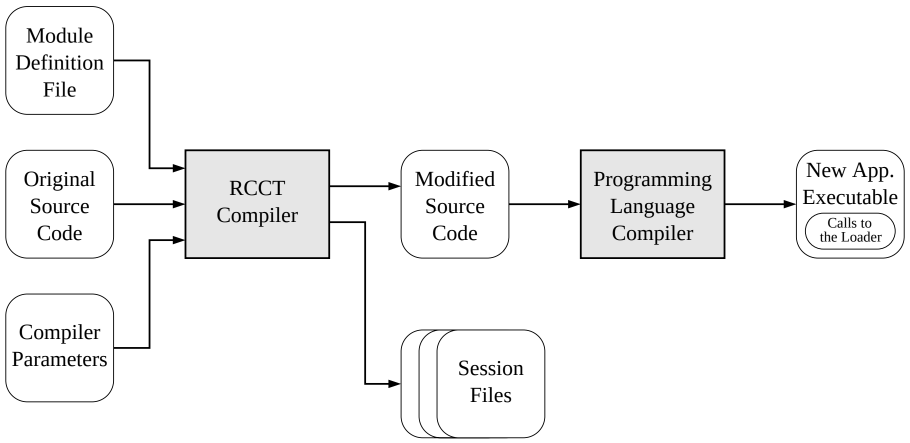


Figure 1.2: Compilation phase.

Figure 1.3 illustrates the execution phase of the mapping process. In this phase, sections of the application executable code that do not require high performance run on the general purpose processor. When the execution reaches the sections produced by the RCCT Compiler, the application utilizes the RC system via several function calls to the Loader. The Loader includes four types of functions, that can be called by the application. These are: `IsModuleAvailable`, `Store_`*`Data`*, `Load_`*`Data`* and `Run` functions. The `Store_`*`Data`* and `Load_`*`Data`* functions move data between FPGA devices and the host processor. By calling the `Run` function with a session file name as a parameter to it, it instructs the Loader to execute a given assembly language program on a particular FPGA device with a specific hardware module.

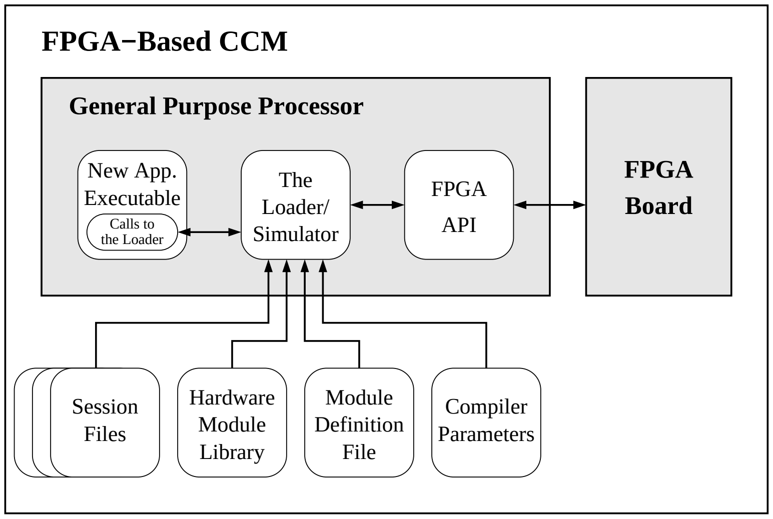


Figure 1.3: Execution phase.

1.3 Distinguishing Features of RCCT

Recently, several tools have been presented in the literature for compilation of high-level programming languages, such as Java and C++, to hardware designs. These tools typically take the user application developed in a high level programming language and produce a circuit design implemented in a Hardware Description Language (HDL). This HDL code is synthesized, and FPGA placement and routing tools are executed to map the design to a specific RC system.

The major problem with these tools is that the time required for placement and routing of the HDL source code is significant and has to be repeated each time a change is made to the source code. On the other hand, RCCT uses pre-compiled, placed and routed hardware modules which are loaded into the FPGA as needed. Thus, it eliminates these time consuming steps from the application mapping process. Figure 1.4 shows the design flow for RCCT and the traditional design flow approach.

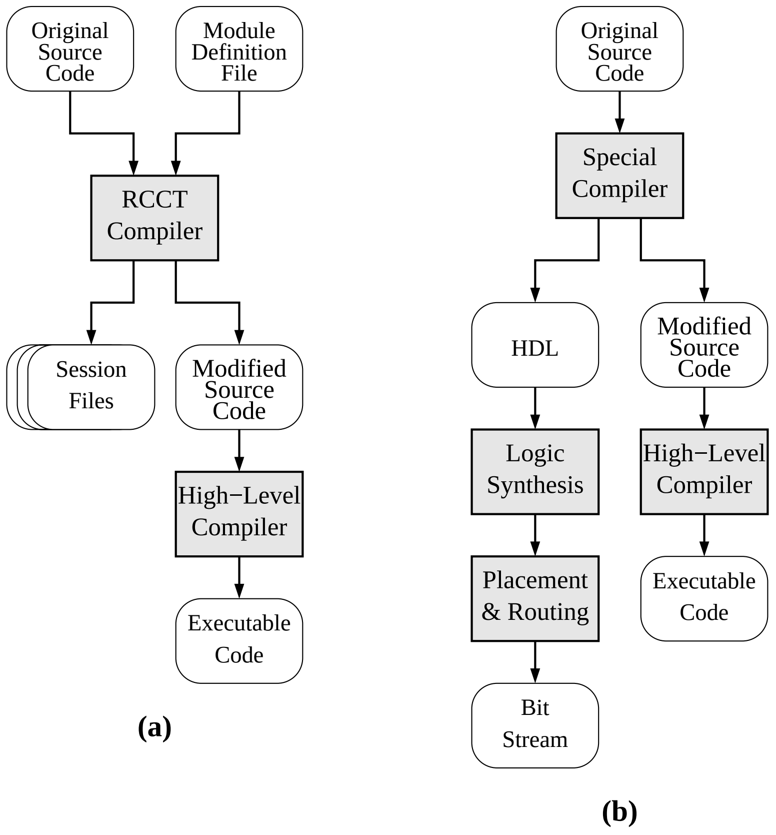


Figure 1.4: Design flows. **a)** RCTT, **b)** Traditional approach.

In addition, the Hardware Module Library of RCCT is dynamic. That is, as new modules are designed and implemented, the user of the tool can easily add them to the module definition file. The Compiler can then recognize and make use of these new modules without any modification on it. Therefore, as specifications for the new module implementations are added to the module definition file, the tool has a better chance to improve the performance of the user applications when they are mapped to the RC systems.

1.4 Organization of the Dissertation

This thesis is organized as follows. Chapter one gives an overview of RCCT and its components. It also presents the distinguishing features of the tool. Background information and related studies presented in Chapter 2. Through the course of this research, our understanding of the module design and implementation improved greatly. As a result, several versions of the hardware modules have been designed and implemented. Details of the module design and the core units used in these modules are introduced in Chapter 3. The Compiler is the most important part of RCCT. It consists of a total of 10467 lines of C++ code: 2482 produced using compiler generation tools, and 7985 generated manually. Several recursive algorithms were also developed/implemented for the Compiler and are presented next. Details of the Compiler are presented in Chapter 4. The Loader is another major component of RCCT. Two versions: the stand-alone version and the Dynamic Link Library (DLL) version of the Loader are discussed in Chapter 5. We implemented an RC resource simulator by adding several functions to the DLL version of the Loader. The Simulator is also presented in Chapter 5. Several applications were mapped using the tool and were simulated for different RC environments to demonstrate the effectiveness of RCCT. These applications and the results of mapping them to an RC system are introduced in Chapter 6. Chapter 7 presents conclusions and suggestions for future research.

Chapter 2

Background

A Reconfigurable Computing (RC) system can be defined as a computer system that contains one or more general purpose processors and one or more configurable hardware components that are designed for their functionality to be configured by the user of the system for different applications. Usually the general purpose processor acts as the host processor and the reconfigurable hardware components are used as a coprocessor. The users of the RC system typically partition their applications and execute the computationally complex sections on the reconfigurable hardware to potentially increase performance.

It has been shown that executing computationally complex sections of applications on RC systems significantly reduces the execution time of the applications compared to the general purpose processor only systems [11]. However, applications must be mapped to FPGA devices before they can be executed on these systems. The mapping processes can be performed either manually or automatically using software tools. Several applications were mapped to RC systems manually including image processing algorithms [4, 5, 6, 7], genetic optimization algorithms [8], and pattern recognition [9].

Mapping applications to FPGA devices manually is a time consuming process. Some of the steps taken during the manual mapping are designing the hardware, coding it in a Hardware Description Language (HDL), compiling the HDL code to bit streams for the FPGA devices (which could take several hours) and developing an interface to utilize the new design on the FPGA devices. A mistake done at the design stage requires repetition of the steps and results in a increase in the development time.

Several tools were developed to automate the application mapping process and to reduce the Reconfigurable Computing application development time. With these tools, in a matter of hours or days, RC implementations of the software applications can be developed.

The tradeoff between the manual and automatic mapping is the quality of the resulting RC implementations. In the manual mapping, one can develop a specific hardware implementation and an interface resulting in a performance improvement for the target applications. On the other hand, in the automatic mapping, improvement in the application performance is limited by the tool's ability to map the applications to the RC systems. Several studies have been conducted at research centers to improve the quality of the automatic mapping process. In this chapter, after a brief introduction to the building blocks of the RC system (FPGA devices) we will introduce some of the selected automatic mapping tools.

2.1 FPGA Technology

Field Programmable Gate Arrays (FPGAs) are a type of chip that are completely prefabricated and contain special features for customization [12, 13, 14]. The user of these chips can implement digital circuit designs by configuring them. The biggest advantage of these chips is their configuration time. Since the configuration time of these chips is very small (for some chips the configuration time is less then a millisecond), circuit designs can be realized very quickly compared to Application Specific Integrated Circuits (ASIC) implementations. A typical circuit development cycle for an FPGA device includes four steps. These steps are designing the circuit, coding the design in a Hardware Description Language, compiling the HDL code to a configuration file and loading the configuration to the chip.

FPGA chips can be classified into three categories according to the structure of their configurable parts [15]. These categories are Static Random Access Memory (SRAM)-based FPGAs, Electrically Erasable and Programmable Read Only Memory (EEPROM)-based FPGAs, and antifuse-based FPGAs. Because the configuration of an antifuse is permanent, antifuse-based FPGA's are one-time programmable devices. Thus, the user of the chip should carefully verify his design before loading the configuration file onto the chip. The configuration of an EEPROM-based FPGA can be changed electrically. Since loading and erasing a configuration to/from a chip is done using high-voltage electrical signals, they have to be configured with special equipment before they are placed into the target systems. SRAM-based FPGA chips can be reconfigured very easily by loading the bits in the configuration file into the SRAM memory cells. These chips can be reconfigured at run time by loading a new configuration to the SRAM cells. Since they use the same technology as computer memories, they have to be configured each time the system is powered on. Because the ease of configuration, SRAM-based FPGA chips are the most widely used FPGA chips. Also, theoretically these chips can be reprogrammed an infinite number of times. The flexibility of FPGA devices comes at a cost in efficiency relative to the ASIC circuits. Villasenor et. al. stated that "an FPGA device never achieves the power, clock rate, or die size that could be realized in a full custom chip optimized for a particular task".

A typical FPGA device contains three configurable parts [15, 16]. These parts are an array of logic cells called Configurable Logic Blocks (CLBs), a programmable interconnection network and programmable input/output blocks. Figure 2.1 shows the structure of the Xilinx 4000 series FPGA devices. Each I/O block includes a number of I/O cells. These cells provide the interface between the package pins and internal signal lines of the FPGA chip. Each cell can be configured as an input, output, or bidirectional port. The interconnection network consists of switch boxes and metal wires. The CLBs are connected together by configuring the switch boxes in the interconnection network. Two most commonly used interconnection network types are *island style* and *cellular style*. In *island style* networks, point-to-point communications between the CLB are possible. On the other hand, the *cellular style* network provides only local communication between the CLBs.

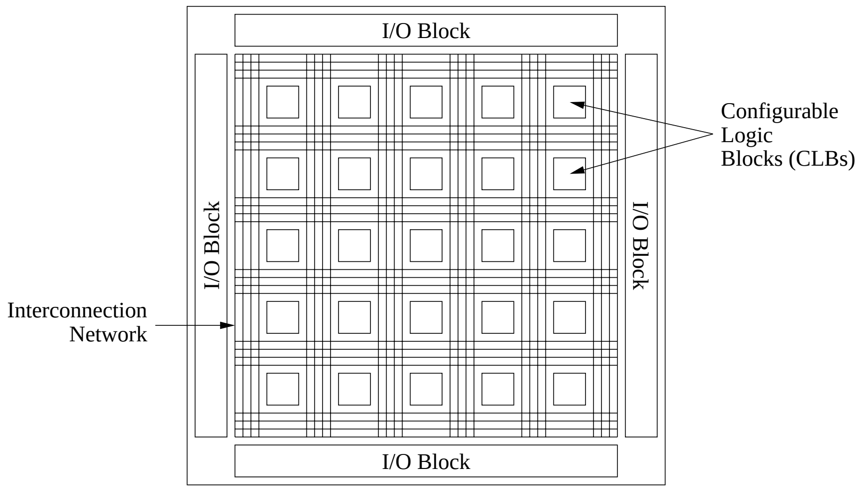


Figure 2.1: Structure of the Xilinx 4000 series FPGA chips [3].

CLBs are the most important parts of the FPGA device. Each FPGA manufacturer implements a different type of CLB. In this section, we briefly introduce the structure of CLBs for the Xilinx series FPGA chips. Figure 2.2 shows the block diagram of the CLB used in Xilinx 4000 series FPGA chips [3, 17]. This CLB includes three Lookup tables (LUT), two programmable flip-flops and several programmable multiplexers. The LUTs are function generators, capable of implementing any combinational logic function of their inputs. The LUTs in Figure 2.2 can perform any function of up to five inputs when they are combined. SRAM controlled multiplexers are used to route signals within the CLB. The flip-flops are used to register output signals when required.

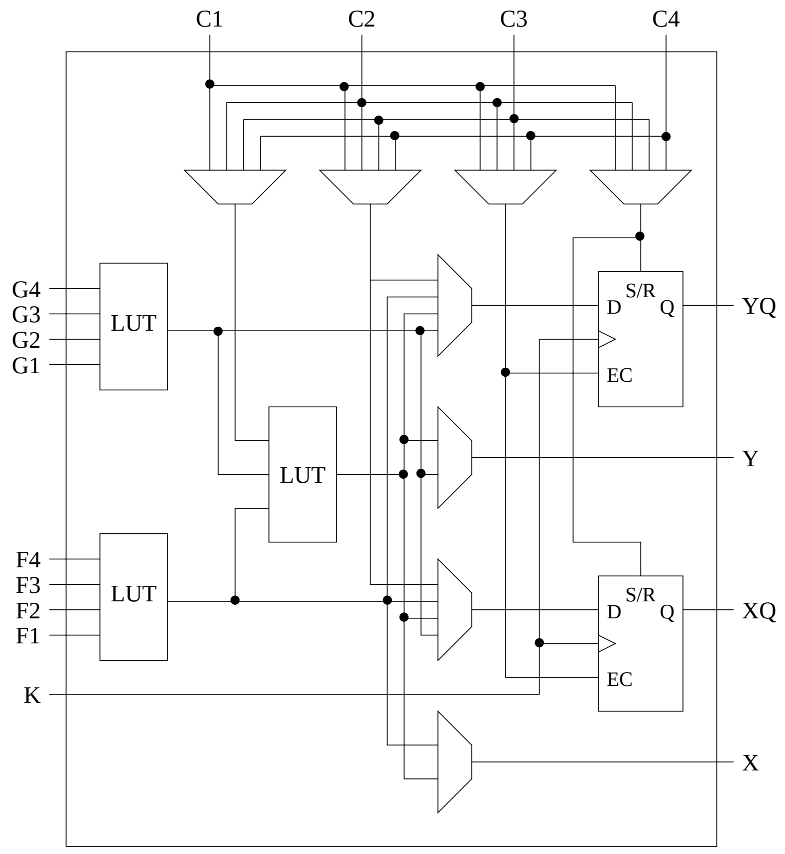


Figure 2.2: CLB block diagram of Xilinx 4000 series FPGA chips [3].

In Xilinx's latest FPGA chips, (the Virtex-II Pro), each CLB comprises four similar slices cite [18]. The slices are connected together with a local feedback bus. The Four slices in the CLB are split into two columns. Each slide includes two four-input function generators, arithmetic logic gates, carry logic, function multiplexers and data storage elements.

In a reconfigurable computer, one or more FPGA chips are organized on a printed circuit board (PCB) and they are attached to a host computer as a coprocessor. Some of the most famous FPGA boards are SPLASH-2 [19, 20, 21] and DECPeRLe [22, 23, 24]. The SPLASH-2 board includes a linear array of Xilinx 4010 FPGA chips. Sixteen FPGA chips are used on the board and they are organized in a linear systolic array. One additional FPGA is used for control purposes. Each FPGA has a limited 36-bit connection to its two nearest neighbor chips. A 512 KByte local memory is also attached to each FPGA. Several SPLASH boards can be connected to form a chain and up to 16 boards can be connected together to form a 256-element linear systolic array. The DECPeRLe-I board includes 23 Xilinx 3090 FPGAs. Sixteen FPGAs were used to form a 4 x 4 array and the remaining chips were used for interfacing with the RAM and the host computer.

2.2 An overview of the automatic mapping tools for RC Systems

With a promise to deliver high-performance and flexibility, FPGA-based Custom Computing Machines attract great attention in the scientific community. Several studies have been conducted to automate the process of designing and mapping applications to these machines [25, 26, 27, 28, 29, 30, 31, 32].

Some of these studies are similar in terms of the methodology and environment employed. Hauck et. al. classified the automatic mapping tools according to the tool suites' input application language [33]. According to his classification, the input application language tool classes are: those that support C [34, 35, 36, 37, 38, 39, 40, 41, 42], C++ [43], Ada [44], Occam [45, 46], Data Parallel C [47, 48], Smalltalk [49], Assembly [50], and special hardware description languages [51, 52]. The Reconfigurable Computing Compilation Tool (RCCT) developed in this thesis supports C and C++. Thus, according to Hauck's classification, our tool can be classified in the first two groups. In fact RCCT is very flexible. By reconstructing the front end of the RCCT Compiler, some other high-level programming languages can also be supported.

A better and more up-to-date classification was done by Radunovic et. al. They classified these studies according to their granularity (fine, medium and coarse), integration (closely coupled and loosely coupled), and reconfigurability of the external interconnection network (fixed network, reconfigurable network) [53], [54]. The granularity reflects the smallest block that a RC device is made of, and the integration is the way the RC device is connected to a general purpose processor. The following is a partial list of the classes related to our study that were reported by Radunovic et. al. with some example studies.

1. Coarse grained, Fixed external interconnection network: RAW project conducted at MIT [26].

2. Medium grained, Fixed external interconnection network, Loosely coupled: MATRIX project by Mirsky et. al. at MIT [32] and Xputer project studied by Rainer Kress et. al. at University of Trier Germany [55, 56].

3. Fine grained, Fixed external interconnection network, Loosely coupled: SPLASH machine developed by Gokhale et. al at Supercomputing Research Center [57], PRISM-I developed by Athanas et. al at Virginia Polytechnic University [34], and RENCO studied by Haenni et. al. at Swiss Federal Institute of Technology [58].

4. Fine grained, Fixed external interconnection network, Closely coupled: SPYDER project developed by Iseli et. al. at Swiss Federal Institute of Technology [43], and GARP project developed by Hauzer et. al. at University of California, Berkeley [59].

5. Fine grained, Reconfigurable external interconnection network, Loosely coupled: SPLASH-II developed by Buel et. al. at Supercomputing Research Center [60], and SOP (Sea Of Processors) developed by Yamauchi et. al at NEC Laboratory [42].

According to Radunovic's classification, our tool, RCCT, falls in to the 4^{th} class because we used fine grained Processing Elements (PE), a general purpose processor has to communicate with PEs through a PCI bus and the current version of the tool does not support communication between the PEs.

The general aim of the automatic mapping tools is to take a software application implemented in a high-level programming language and translate it into a hardware configuration suitable for a specific RC system. Different tool suites have been developed for applications implemented in different high-level programming languages. Usually, each approach introduces a new reconfigurable architecture and a set of associated compilation/simulation tools. Since C/C++ is the most widely used high-level programming language, most of the mapping tools accept applications implemented in C/C++. In the following sections some of the selected studies are briefly presented.

2.2.1 Using an High-Level Programming Language (HLPL) as an HDL

High level programming languages offer several advantages over the traditional hardware description languages in digital circuit development. Some of the advantages are: fast circuit synthesis, easy simulation and debugging, and a more flexible environment for circuit specification. Several studies have been conducted to develop digital circuit design automation tools that uses a HLPL as an input.

A Java-based FPGA application development and debugging tool, JHDL, has been developed by Hutchings et. al. at Brigham Young University [25, 61, 62]. JHDL is a combination of several design and development tools including a circuit library, a simulator, a schematic generator and a hardware debugger.

JHDL helps RC application developers in two ways. First, it provides several tools to develop, simulate and debug an application. Second, it provides tools to generate interface programs that can be used to effectively utilize RC systems.

As a hardware description language, Java was selected. The reason for selecting Java that it is a common programming language and its object-oriented nature is useful for capturing all the details of hierarchical circuit designs. In JHDL, gates, circuits and wires are represented as Java objects. The users specify their designs in JHDL by extending existing classes from the JHDL circuit library. Although JHDL provides great benefits for developing, debugging and running applications, it does not automate the RC application development process completely. Users still have to design and hand code the application.

Another Java-based development tool has been developed by Chu et. al. at the University of California at Berkeley [63]. This tool is a generator framework which can be utilized in large development tools for RC systems. The user of the tool can describe his design using the base class provided by the tool. A block-based hierarchical design methodology is supported. The tool also provides several methods that can access any subcomponent and perform various functions on the subcomponents. The tool is able to generate an XNF netlist and simulate the design. This tool, used in isolation, is not sufficient to automate the RC application development process but can be utilized in an integrated design automation environment.

One major advantage of the HLPL based development tools is that the user can specify the design using a high-level programming language. On the other hand, the initial design of the application has to be completely manually coded in a specific HLPL. These tools are not capable of converting an arbitrary application written in Java or C++ into a magic datapath. Another drawback of these development tools is that the user must have experience in hardware design and must specify his design in the HLPL similar to any other hardware description language. Lastly, changes in the HLPL description require the user to run placement and routing tools repeatedly. Whereas, each run could take several hours to complete.

2.2.2 Raw Machines

In a research project called Reconfigurable Architecture Workstation (RAW), Waingold et. al. at Massachusetts Institute of Technology, a new reconfigurable system was developed [26, 64, 65]. Their system has two components, a new RAW

Processor architecture and several compilation tools that help to map applications to this architecture.

The RAW microprocessor is based on highly interconnected replicated tiles. Tiles in the processor work as separate processing elements (PE). Each tile contains an instruction memory, and a data memory, an ALU, and registers. The processor has two configurable parts, a configurable logic part that allows the users to implement custom instructions, and a programmable switch that supports both dynamic and compiler-orchestrated static routing between tiles. The RAW microprocessor can be viewed as a gigantic FPGA with coarse grained tiles in which software-orchestrates communication over the static interconnections [66].

The compiler for the RAW processor is able to map serial or parallel programs written in C. The RAW compiler views the tiles in a RAW processor as a collection of function units. While compiling a program, the compiler tries to utilize as many tiles as possible to maximize instruction level parallelism. The compiler is also responsible for selecting an application specific configuration for loading into the configurable logic in each tile. Additionally, the compiler also calculates the switch configurations for a given application to regulate the communications between the tiles.

Several algorithms have been mapped to the RAW processor and the results demonstrate that the RAW processor can achieve 10 to 1000 speedup over the Sparc station 20/71. The main advantage of the RAW processor comes from its parallel structure and its reconfigurable parts (configurable logic inside each tile and configurable switches between the tiles). The compiler can effectively select configurations for these parts for a given application to maximize the speed. The

RAW processor is a kind of MIMD processor. The PEs are simplified in order to reduce hardware complexity. On the other hand, due to this simplification, implementing floating-point applications seems infeasible.

This project differs from our research in terms of execution of the applications. It tries to execute the entire applications on PEs. On the other hand, our system executes only the computationally complex parts of the applications on the RC system using the modules in our module library.

2.2.3 Pipeline Vectorization for RC Systems

Weinhart et. al. developed a tool for automatically producing pipelined circuits from high-level programs [27, 67]. They used software vectorization techniques to produce circuits for reconfigurable devices.

This research mainly focused on vectorization of the loop structures in high-level programs and creating circuits for RC systems. First, their tool normalizes the target loops in the application program using several loop transformation techniques. After that, it checks the data dependencies and generates a Dataflow Graph (DG) for each loop using the predefined macro cells. In the next step, the tool inserts pipeline registers between the combinational logic units to reduce the critical path delay. Next, candidate DGs are selected according to their potential in terms of speedup. Finally, the selected DGs are synthesized and, using vendors' tools, bit streams for specific FPGA device are created. They claimed that their tool is not architecture specific and can be used as an front end for some other design suites such as RaPid-B.

The main advantage of this tool is that it automatically generates the circuits necessary to calculate arithmetic operations in a loop. However, the tool is not able to create the controller and interface program for the datapaths. The output of the tool is a circuit design implemented in a HDL. This HDL implementation has to go through the placement and routing processes to be usable. The placement and routing step influences the application development of the tool negatively (requiring several hours to days). Another disadvantage of the tool is that it is not able to handle nested loops. It only generates datapaths for the innermost loops, resulting in poor performance in execution of nested loops. This tool, used in isolation, is not able to automate the entire process of mapping applications to RC systems. This project differs from our research in terms of creating the hardware components and level of abstractions. They build datapaths for each specific loop. In our case, hardware modules are predefined, generalized to accommodate different cases, optimized for speed, and able to handle nested loops. Our tool also includes a generalized interface for different applications.

2.2.4 Handel-C

Ian Page and his research group at Oxford University, developed a new behavioral programming language called Handel-C and a compiler for mapping applications to reconfigurable systems [68, 69]. Later, Page founded a company called Celoxica and commercialized the Handel-C system by adding several new tools to the suite.

The Handel-C language was developed for specifying high-level algorithms and compiling them directly into gate-level hardware for FPGA implementation. The language is a variant of the C language with additional constructs to accommodate parallel execution [70]. The parallel constructs were added to exploit inherent parallelism in the algorithm being mapped to hardware. The language also supports variable bit-with data types.

The commercialized Handel-C environment consists of a simulator, a compiler and a user interface. The simulator can display content and status of the variables declared in the given program or design. It can also visualize actual timing behavior of the design for a number of clock cycles. The compiler processes the design and generates either EDIF or VHDL for FPGA implementation [71].

The hardware development cycle using the Handel-C environment includes several steps. First, the algorithm has to be coded in the Handel-C language. While coding, the user has to identify potential parallel execution opportunities. Then, the hardware description must be simulated. The design is then compiled to VHDL code using the Handel-C compiler. From this point on, the user has to use FPGA vendors design tools to place, route and create the final bit stream for the FPGA chips from the VHDL code generated by Handel-C compiler.

Handel-C has a few weaknesses. First of all, it only supports single clock designs. Modularity in the language is not strong enough [70]. Compared to RCCT, Handel-C also has some other weak points. The user has to be knowledgeable in hardware design. Applications already implemented in the C language can not be used directly. The user has to re-implement his/her application fully or partially in Handel-C language. Also, the Handel-C compiler produces VHDL and this code has to go through the placement and routing process which are very time consuming steps. RCCT eliminates all of these weak points of using Handel-C.

2.2.5 Garp

Garp is another reconfigurable architecture and compiler set for automatic mapping of applications implemented in a high-level language [59, 28]. The Garp architecture includes a MIPS processor and a reconfigurable array of logic blocks. It combines the processor and the reconfigurable hardware on the same die.

The reconfigurable hardware includes a two dimensional array of logic blocks. Logic blocks are similar to CLB's of the Xilinx 4000 series FPGA chips. Four memory busses are placed vertically through the rows. These buses are used to transfer data to/from the logic block array. They are also used for loading configurations to the array and saving/restoring the array state. Another wire network is used for the communication between the blocks. The main processor is responsible for the loading and execution of the configurations. Several new instructions were added to the MIPS instructions set for these purposes.

The software environment for the Garp architecture includes a compiler and a simulator. Actual processor chip has not been implemented yet; therefore, a simulator is used instead of the processor. The compiler accepts applications implemented in C language. It uses SUIF as a front end [72]. It performs several tasks on the user source code before producing the final executable code for the processor. The compiler takes the source code written in C and generates a dataflow graph. Then, the module mapping and the placement tasks are performed. In the module mapping task, modules are synthesized for the nodes in the dataflow graph. In the placement step, the synthesized modules are placed for the reconfigurable array. Next, the routing task is applied to the placed module and a bit stream is generated for the modules. In the final step, the compiler links the bit stream as constant data to the final program executable code.

While the Garp compiler works at the instruction level, the RCCT tool works at the statement level. Thus, it may not be able to exploit parallelism available in the user applications. Also, Reconfigurable resources available in the the Garp processor are limited. Hence, large tasks cannot be implemented. Moreover, floating-point operations cannot be implemented on the reconfigurable part of the Garp processor.

2.2.6 PipeRench

Goldstein et.al. at Carnegie Mellon University developed a reconfigurable system called PipeRench. Similar to the design suites mentioned above, PipeRench also includes a reconfigurable architecture and an associated compiler [73, 74, 75, 76].

Developers of PipeRench identified several problems associated with the current commercial FPGA devices. The granularity of the FPGA chips were not suitable for multimedia applications. The configuration times were found to be the limiting factor in the FPGA potential performance gain. Also, FPGAs typically were not forward compatible. That is, applications developed for the current FPGAs need to be redesigned and compiled for the future FPGA devices. The sizes of the FPGA chips were also found to be not large enough to implement a wide variety of applications. They claimed that their approach solved these problems.

The PipeRench processor includes a set of physical pipeline stages called stripes. Each stripe contains a set of PEs and an interconnection network. Each PE contains an ALU and a pass register file. PEs can access operands from registered outputs of the previous stripe through the interconnection network.

When an applications is compiled for PipeRench, the resulting hardware for the application is divided into small sequential pieces and a separate configuration file for each piece is generated. Let us assume that an application is divided into five sequential pieces and compiled for a three stage PipeRench. Also assume that the execution of each piece takes two cycles. During the execution of the example application, the first sequential configuration file is loaded into the first pipeline stage in the first clock cycle. In the second cycle, the second configuration is loaded into the second stage while the first stage is executing the first part of the applications. In the next cycle, the third configuration is loaded into the third stage. In this cycle, configuration one continues execution in the first stage, and the second configuration starts execution and it uses the result produced by the first stage. In the fourth cycle, the first stage of the pipeline becomes available and is configured with the fourth configuration file. The second configuration continues execution and the third starts execution. Sequential pieces of the application are loaded into the pipeline in order and executed repeatedly until the application completes. PipeRench uses a technique called pipeline configuration to rapidly configure each stage of the pipeline [77].

The compiler for PipeRench accepts applications implemented in a special language called the Dataflow Intermediate Language (DIL). After performing several steps on the applications implemented in DIL, the compiler produces configuration files for the PipeRench.

Goldstein et.al. compared PipeRench with the state-of-the-art general purpose processors. Results showed that PipeRench outperforms other processor by more than 10 times.

The most important advantage of PipeRench over the other reconfigurable systems is that it is able to reduce the configurable resources needed for a given application. On the other hand, it may not fully exploit the available parallelism because detecting parallelism at the instruction level is a more difficult task. Since it uses a special language, the user of the system needs to be familiar with the language and hardware design issues. RCCT eliminates these disadvantages of PipeRench.

2.2.7 COBRA-ABS

Duncan et. al. at the University of Aberdeen, developed another automatic mapping system called COBRA-ABS [78, 79]. COBRA-ABS system includes a parameterized compiler. The purpose of the compiler is to transform high-level applications implemented in C into VHDL code. The compiler reads three input files. These are the algorithm description specified in C, the definition of the target FPGA-based Custom Computing Machine and a library file that includes the definitions of the available RTL datapath library components. After the compilation process, the compiler produces VHDL code for the synthesized hardware design and a compiler script. The user of the tool must compile the VHDL code with the FPGA vendor's compilation tool to get the final bit stream.

One nice feature of the tool is that it is able to generate a compiler script. Its advantage over the other automatic mapping systems is that, to some degree, it is not architecture dependent. The user can specify different target RC environments. One drawback of the system is that it was designed to synthesize DSP algorithms implemented in, C rather than other generic applications.

Chapter 3

The Hardware Module Library

The Hardware Module Library is one of the major components of the Reconfigurable Computing Compilation Tool (RCCT). It includes several hardware modules designed to perform vector operations on the RC systems. In fact, each hardware module in the library is a pre-compiled, placed and routed configuration file to be used with a specific FPGA device. The modules were designed as vector processors, implemented in **V**ery High Speed Integrated Circuits **H**ardware **D**escription **L**anguage (VHDL), and compiled to configuration files using the FPGA vendor's compilation tools. Module generation is performed in isolation from application compilation. Hence, with our approach, long place and route times can be tolerated.

Through the course of this research, our understanding of the module design and implementation improved greatly. In addition, improvements in the FPGA technology enabled us to use more complex module designs. As a result, several versions of the modules have been designed and implemented. It is possible to classify these modules into three generations. In each generation, different controllers and datapaths are used with the same basic floating-point core units. Each generation is an improved version of its preceding generation.

In this chapter, basic floating-point core units are presented after a brief introduction to the hardware modules. Then, detailed information about module generations is given. The chapter concludes with the mathematical models of the modules and a comparison between the module execution and a general purpose processor execution.

3.1 Introduction to the Hardware Modules

A hardware module in the Module Library is a pre-compiled, placed and routed configuration file to be used with a specific FPGA device. The purpose of the hardware modules is to execute computationally complex sections (ie. mapped sections) of the user applications on FPGA devices with a significant speedup over the execution of the same code on a general purpose processor.

The modules are used in the execution phase of the application mapping process as shown in Figure 3.1. When the execution order comes to a mapped section of the user application, the Loader configures the FPGA devices with appropriate hardware modules and executes that section of the application on the RC system.

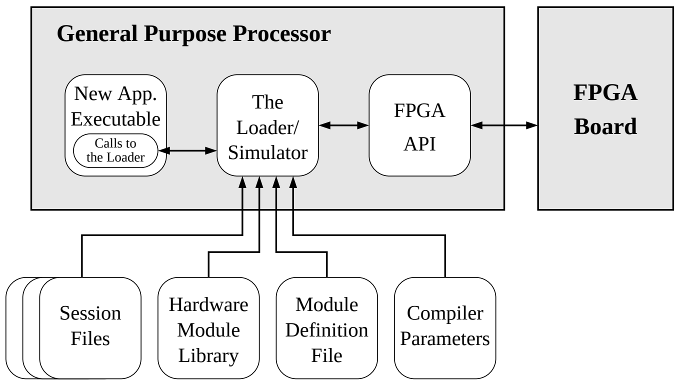


Figure 3.1: Execution phase.

Several modules have been designed and implemented to perform different vector operations on FPGA devices. In fact, each module was designed as a vector processor, capable of performing one or more mathematical operations on input vectors. We focused on vector operations due to the fact that general purpose processors perform single operations very well. There is no significant potential performance gain with the execution of a single operation on an FPGA. Actually, there is a performance degradation because of the data transfer issues between the host computer and the FPGA device. On the other hand, when the FGPA is configured with speed-optimized hardware modules, the FPGA device can easily outperform general purpose processors on vector operations.

Currently, all modules have been designed to perform IEEE single precision floating-point operations. We decided to implement floating-point modules instead of integer modules for the following reasons. First, most of the scientific applications that require an extensive amount of CPU time process floating-point data. Secondly, floating-point operations require more CPU time than integer operations. And finally, it is easier to debug hardware that was designed using floating-point when the original application uses floating-point. No floating-point to integer conversions are required.

We designed several floating-point modules to cover a wide variety of vector operations [80, 81, 82]. Some of the modules includes three and four operand addition, subtraction, multiplication and division modules, and accumulation module. Creating several different types of modules that are useful for various applications is a time consuming process. To reduce the design and implementation time, we took the following approach. First, we developed several standard components. These components are basic floating-point core units, module controllers and module datapaths. In order to facilitate module interconnection for complex operations, these components are standardized in terms of the number of inputs, number of outputs, and module latency. Second, several different types of modules have been created by combining basic core units with a few controllers and datapaths. Using this approach, the time required to design a new module is reduced significantly. When a new core unit is designed, one simply combines the new core with an off-the-shelf controller and datapath to form a new module. The modules were implemented using VHDL. Several thousand lines of VHDL code were written for these modules. After the VHDL coding, modules were compiled placed and routed for the Xilinx XC4044XL FPGA chip.

3.2 Basic Floating-Point Core Units

The most important components of the hardware modules are the basic floating-point core units instantiated in the datapath. For each floating-point operation, we developed a standard basic core unit. Each core unit is highly pipelined, has the same number of inputs and outputs, and has the same latency. We created several modules by instantiating one or more basic core units into a standard module structure.

Figure 3.2 shows the block diagram of the standardized basic core unit. Each core has two 32-bit inputs and one 32-bit output to accommodate single precision floating-point numbers. For addition, subtraction and multiplication, different floating-point core units were developed. There is a standard interface definition for the basic core units to reduce design time. Once a new core unit is designed, it is easy to create a new module by just instantiating the new core unit into one of the standard module structures.

As shown in Figure 3.2, all basic core units are divided into a standard number of pipeline stages (8) to improve the maximum clock speed that can be applied to the units. We used a standard number of pipeline stages to alleviate the need to develop a unique controller for each core. However, the main controller can handle cores with arbitrary latencies. While using pipelined units requires additional registers, resulting in an increase in FPGA CLB resources, it provides a significant benefit in terms of increased clock speed.

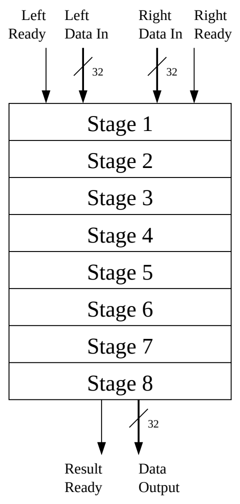


Figure 3.2: Block diagram of the standard core units.

The basic core units are designed as self-controlled units to reduce the hardware requirements and to make the module controller simpler. Once data is available at both inputs (`LEFT_READY` and `RIGHT_READY = 1`), the core unit starts processing. Results are available at the output of the unit 8 clock cycles later. This is accomplished with a standard floating-point core I/O interface. Each core has two input signals and one output signal for control and core interconnection. Each time that the module controller reads a floating-point number from the memory, it asserts either the `LEFT_READY` or `RIGHT_READY` signal corresponding to the core input that has valid data. When both inputs to the core have valid data and both ready signals are asserted, the core begins the floating-point operation. When the core finishes processing the data, it asserts the `RESULT_READY` signal. The main controller then stores the result in memory.

Use of the standard interface control signals serves two purposes. The main purpose is to reduce controller complexity. Hence, a single controller can handle future cores with arbitrary latencies. The controller does not send command signals to each stage of the core. Instead, it uses the interface signals to signal the core that the input data is ready. It also uses the `RESULT_READY` signal produced by the core to determine when the result is ready. This simplification in the controller saves control states, logic gates, and future application development time. The other purpose is to facilitate the addition of complex cores into the library. The use of the standard interface control signals makes it easy to form larger cores by simply linking existing cores together.

3.3 The First Generation Module

As a first generation module, we designed and implemented a vector adder. It was implemented as a test case to see the potential performance of our design before implementing the other modules in the same style. Figure 3.3 shows the top level block diagram of the vector adder.

The module architecture consists of a datapath and three separate controllers. The datapath is a standard datapath containing a floating-point core unit (adder core), registers, multiplexers and counters. As can be seen from Figure 3.3, to control the datapath and data movements between the datapath and the memory unit, a main controller, a read and a write controllers, were used. The main controller is responsible for starting and stopping given vector operations and controlling the read and write controllers. Read/Write controllers are used to fetch instructions and read and write data from/to local memory.

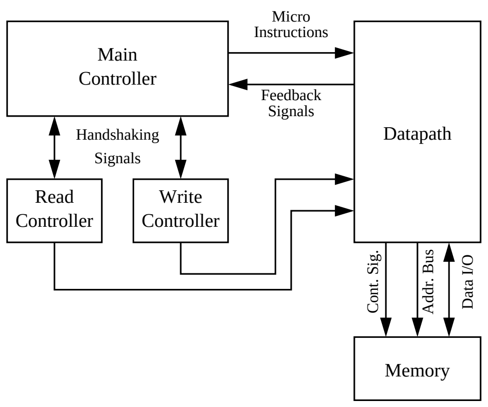


Figure 3.3: Block diagram of the first generation module.

To execute one vector instruction, the main controller first fetches the instruction by activating the read controller and initializes the vector address counters in the datapath. After the initialization, the main controller starts executing the instruction. Since it is a vector instruction, the main controller performs this operation in a loop. At each iteration of the loop, it reads the input data to the datapath by again activating the read controller. After the data is read, the main controller lets the datapath process it. When the data is processed and the result is available, the main controller writes the result back to the memory by activating the write controller. In this approach, the main controller is isolated from memory operations. Thus, every time the memory is accessed, the main controller activates either the read or write controller. Handshaking signals were used between the main controller and the read and write controllers to activate either controller and to check if these controllers had completed their tasks.

The main reason we use three separate controllers is that we want our design to be easily adaptable for different conditions. With little or no modification to the design, we can create new modules or recompile current module implementations for different types of RC systems. For instance, if we replace the core unit in the datapath with another core with a different number of pipeline stages, we only need to update the main controller. If we want this module to be available for another RC system, we only need to modify the read and write controllers.

After we implemented the first module, we tested it using large vectors. During the initial tests, we monitored both the execution time and the functionality of the module. Functionally, the module behaved as expected. On the other hand, the execution time results were not encouraging. We were expecting enhanced performance compared to general purpose processors, but actually observed a performance degradation.

When we closely examined the module, we discovered that the main controller spent too many clock cycles communicating with the read and write controllers, resulting in poor memory access performance. The controller was not able to utilize the memory address bus and memory data bus efficiently. Although the datapath is able to process one set of data at every clock cycle, due to the poor bus utilization, the datapath was waiting for data for more than half of the cycles. To solve this problem, we modified the controllers and created the second generation modules.

3.4 The Second Generation Modules

Figure 3.4 shows the generalized block diagram of the second generation floating-point modules. Each module consists of a standard controller and a standard datapath that interfaces with an external memory. For the second generation modules, four datapaths, four controllers, and three core units (adder, subtractor, multiplier and divider cores) were designed and implemented. Additionally, by combining the basic adder and multiplier cores, a multiply-accumulate core was also developed. As shown in Figure 3.4, a total of 9 floating-point modules were formed by combining these standard components.

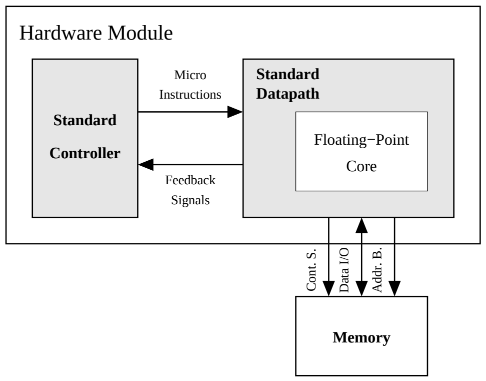


Figure 3.4: Top level block diagram of the second generation modules.

3.4.1 Module Datapaths

Four unique datapaths have been developed for the second generation modules. First, two datapaths were designed for the one and two input vector adder, subtractor and multiplier modules. A third datapath was designed for the accumulator and product modules and a fourth datapath was designed for the multiply-accumulate module. Since datapaths are similar in terms of structure, we only introduce two of them in this section.

Figure 3.5 shows the block diagram of the datapath for the two-input modules. As shown in the figure, the datapath is partitioned into two sections, the data processor and the fetch/decode unit. The data processor section includes a core function unit and two 32-bit data registers, `R0` and `R1`. These registers are used as temporary storage for incoming data from the local memory. By instantiating different core units into this datapath, the adder, subtractor, and multiplier modules are formed. Although currently only these core units are available, this datapath can be used in the design of any module that accepts two input vectors and produces an output vector, provided that all input and output vector sizes are equal.

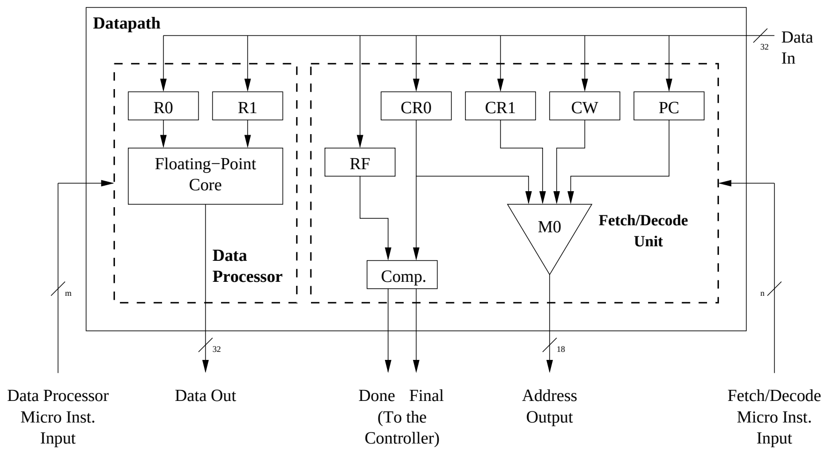


Figure 3.5: Block diagram of the standard datapath for four operand modules.

The fetch and decode unit includes four counters, one register, one specialized comparator, and a multiplexor. The `CR0`, `CR1` and `CW` counters are loadable counters and are used for addressing input and output vectors. `PC` is used as a program counter to keep track of the module instructions. The `RF` register is used to store the size of the input vectors. The specialized comparator produces two signals. The `DONE` signal is asserted when the module completes execution of the current vector instruction. The `FINAL` signal is asserted when all instructions have been processed, (i.e. when the `HALT` instruction is fetched). The `M0` multiplexor is used to select one of the address counters.

Figure 3.6 shows the block diagram of the datapath designed for the multiply-accumulate module. Similar to the datapath illustrated in Figure 3.5, the datapath is partitioned into the data processor and the fetch/decode sections. In the data processor section, two basic core units, the multiplier core and the adder core, were used. The incoming data sets are multiplied by the first core and the results produced by the first core are added up by the second core. To save the hardware resources, only one input register, `R0`, was used before the multiplier core. At each iteration, the first number read from memory is temporarily stored in `R0` and the second number is grabbed directly from the memory and fed to the core unit. `R1` and `R2` are used to store temporary results of the multiplier core. To be able to perform the accumulation process, it is necessary to connect the output of the second core back to the input of the same core. This connection is accomplished with `M0` and `M1` multiplexors.

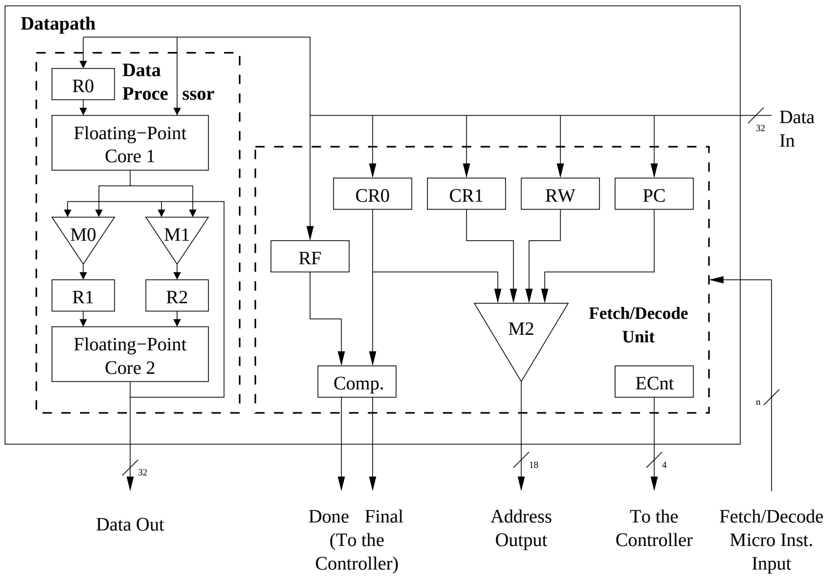


Figure 3.6: Block diagram of the standard datapath for the accumulator and the product modules.

The fetch and decode section is very similar to the previously explained datapath. The only difference is the `E` counter. This counter is utilized while emptying the pipelined core units. After the final input data is loaded from memory, the controller sets this counter equal to the number of cycles required to empty the pipeline. The controller waits until all the remaining data in the core is processed and the results are written back to the memory. The `E` counter is especially useful while emptying the accumulator, product, and multiply-accumulate cores.

3.4.2 Module Controllers

For the second generation modules, four unique main module controllers, one for each datapath, were implemented. The first controller assumes that elements of the input vector pair are interleaved or stored in consecutive memory locations. This controller is used for some of the three operand instruction modules. The second controller assumes that the input vectors are disjoint, and it is used for some of the four operand modules. The first and the second type of controllers were used to construct two different types of vector addition, subtraction, and multiplication modules. The third controller was developed for the accumulator and the product modules and the final controller was developed for the multiply-accumulate module. The controllers for the accumulator, the product and the multiply-accumulate modules are much more complicated than the others due to the required pipeline emptying process.

The first and the second type controllers perform vector operations in a single loop. At each iteration of the loop, they go though four steps: read first, wait idle, read second, and write back steps. The controllers perform vector operations by observing two conditions. The first condition is the availability of the result.

If the result is not available at the output of the core unit, the write back step becomes an idle step. The second condition is whether all the numbers are read from the memory and some results need to be written back to the memory. When this condition happens, read first and read second steps become idle steps. The first condition occurs at the beginning of the vector instruction when the pipelined core is not filled completely and the second conditions occurs while emptying the pipelined core unit. Figure 3.7 shows how the controller utilizes memory address and data buses and floating-point core unit. With this control scheme, a new result is available every 4 cycles.

	$R/\overline{W}$	1	X	1	X	1	X	1	X	1	X	1	X	1	0	1	X	1	0	1	X	X	0	X	X	X	0	X	X	X	0
Mem. Stages	MAB	A_1		A_2		B_1		B_2		C_1		C_2		D_1	A	D_2		E_1	B	E_2			C				D				E
	MDB			A_1		A_2		B_1		B_2		C_1		C_2	A	D_1		D_2	B	E_1		E_2	C				D				E
FP Core Unit Stages	S1						A				B				C				D				E								
	S2							A				B				C				D				E							
	S3								A				B				C				D				E						
	S4									A				B				C				D				E					
	S5										A				B				C				D				E				
	S6											A				B				C				D				E			
	S7												A				B				C				D				E		
	S8													A				B				C				D				E	

Figure 3.7: Memory access schedule for one and two input vector adder, subtractor and multiplier.

The multiply-accumulate controller performs the multiplication and accumulation process in three stages. In the first stage, the pipeline is initialized, and continuously reads numbers from the memory to enable the multiplier core to multiply these numbers. When the multiplication results are available, they are fed to the right input of the adder. At this stage, the multiplier results are forwarded through the adder core by adding 0 (zero). When the first number appears at the adder core output, the controller enters the second stage. In this stage, while still supplying new numbers to the multiplier from the memory and forwarding the result of multiplier results to the adder, the controller starts returning the adder results back to the adder's input and starts accumulation. During this stage, at any time, the pipelined adder core holds some intermediate results. When the last set of data is read from the memory and processed by the multiplier core, the module enters the last stage, pipeline emptying stage. In this stage, all the intermediate results in the core are added up and the final result is written back to the memory.

3.4.3 Module Instruction Format and Module Execution

All modules are designed to execute their specific machine language instruction. In fact, each module is able to execute only one module instruction and the `HALT` instruction. Each module instruction corresponds to a single floating-point vector operation. A standard module instruction includes three or four operands, depending on the type of module used. Figure 3.8 shows the instruction formats for each module type. For each three-operand module instruction, the first operand is the starting address of the input vector, the second is the starting address of the output vector, and the third is the size of the input vectors. For each four-operand module instruction, the first two operands are the starting addresses of the two input vectors, the third operand is the starting address of the output vector and the last operand is the size of the vectors. The floating-point accumulation and product modules use the instruction format of Figure 3.8a. However, these modules produce an output vector with length 1.

All modules were designed for a commercial FPGA board [83] that is readily available in our laboratory. This board includes five FPGA devices, or Processing Elements (PEs). Each PE has its own local memory (1M Byte). The host computer and the PE both have read and write access to the local memory. The memory space of each module is partitioned into two sections, instruction and data. The instruction memory always starts at memory address `$00000` and ends with the `HALT` instruction (`$FFFFFFFF`). The remaining memory that is not used for instructions is used for data.

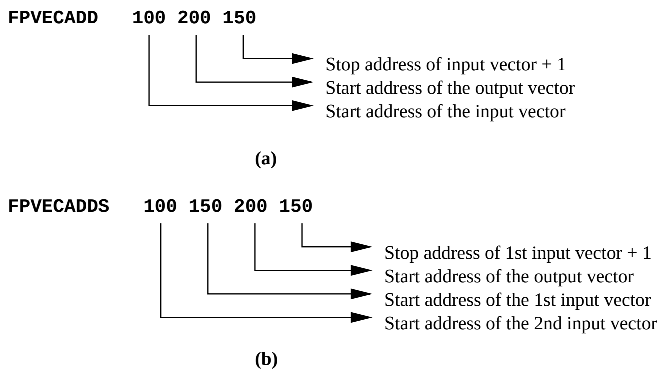

FPVECADD 100 200 150
Stop address of input vector + 1
Start address of the output vector
Start address of the input vector
(a)
FPVECADDS 100 150 200 150
Stop address of 1st input vector + 1
Start address of the output vector
Start address of the 1st input vector
Start address of the 2nd input vector
(b)

Figure 3.8: Modules instruction formats. (a) Module instruction for 3 operand vector modules. (b). Module instruction format for 4 operand vector module and the multiply-accumulate module.

Once a module configuration has been loaded into a PE, and the local memory has been initialized by the host computer, the module waits for the reset signal to be asserted. When this occurs, the module reads the first instruction from the memory location `$00000`. It then begins executing the instruction. When the current instruction is completed, the module reads the next instruction from the instruction memory. This process continues until the module reads a `HALT` instruction (`$FFFFFFFF`) from the instruction memory. When this value is read, the module stops and sends an interrupt signal to the host computer.

The modules' execution times can be evaluated given the number of cycles required to process one set of vectors. The memory unit we used has a two-clock cycle latency for read operations and a one clock cycle latency for write operations. The vector addition, subtraction and multiplication modules write results back to the memory between successive read operations. Hence, the optimal memory access schedule for these modules is two read cycles followed by one write cycle producing a result every 3 cycles. We achieved near optimal performance with our modules since we inserted only one idle state. Using this approach, an output is produced every 4 cycles.

We developed Equation 3.1 to approximate the total execution time T_{EX} of the modules. In this equation, N_F is the number of cycles required to fetch an instruction, N_P is the number of cycles required to process the given vectors, N_E is the number of cycles required to empty the pipelined core, F_M is the module clock rate, and C_{API} is the Application Programming Interface (API) overhead.

$$T_{EX} = \frac{N_F + N_P + N_E}{F_M} + C_{API} \tag{3.1}$$

$$T_{EX} = \frac{4N}{F_M} + C_{API} \tag{3.2}$$

For three and four operand modules developed for vector addition, subtraction, and multiplication, the instruction fetch takes 9 and 10 cycles respectively and pipeline emptying takes 8 cycles. Processing takes 4 cycles per pair of vector elements. The constant API overhead depends on the host computer's speed (in our case, it is 0.3 millisecond for each function call). For large vectors, instruction fetch and pipeline emptying times for addition subtraction, and multiplication are negligible and Equation 3.1 could be rewritten, as Equation 3.2 where N is the length of the vectors.

Since the accumulator and product modules do not write back to the memory until the end of the module instruction, both are able to read an element of the input vector from the memory every clock cycle. As a result, cores in the accumulator and the product modules are utilized 100% and run almost four times as fast as the other modules. Equation 3.1 also applies to these modules. Equation 3.3 shows the execution time for the accumulation and the product modules. The instruction fetch and emptying are negligible.

$$T_{EX} = \frac{N}{F_M} + C_{API} \tag{3.3}$$

$$T_{EX} = \frac{2N}{F_M} + C_{API} \tag{3.4}$$

The multiply-accumulate module is able to read two FP numbers, one number from each input vector, every two cycles. The core units in this module are 50% utilized. Equation 3.4, derived from Equation 3.1, could be used to estimate the execution time for the multiply-accumulate module.

3.4.4 Module Assembly Language Instruction Set

There are two types of instructions in the module assembly language, three-operand instructions and four operand instructions. Table 3.1 shows the three-operand instructions. In three operand instructions, the first operand is the starting address of the input vector and the second operand is the starting address of the output vector. In this instruction format, RF marks the size of the input vector. The output vector size changes depending on the type of the instruction. Three operand `FPVECADD`, `FPVECSUB` and `FPVECMULT` instructions assume that elements of the input vector pair are interleaved, stored in consecutive memory locations as follows: $A[0]$, $A[1]$, $A[2]$, $A[3]$, $\cdots$, $A[m-1]$, where $m = 2n$ and n is the length of each input vector. The even locations are the first input vector and the odd locations are the second input vector. For these instructions, the size of the output vector is equal to one half of the input vector size. Since FPVECACC and FPVECPROD instructions produce a single output, the size of the output vector is equal to one.

Table 3.1: Three-operand instructions.

Instruction	**Operands**	**Meaning**
FPVECADD	[CR] [CW] RF	$C[i] = A[2i] + A[2i+1]$, $i = 0, 1, 2, ..., n-1$ where $n = (size of input vector)/2$
FPVECSUB	[CR] [CW] RF	$C[i] = A[2i] - A[2i+1]$, $i = 0, 1, 2, ..., n-1$ where $n = (size of input vector)/2$
FPVECMULT	[CR] [CW] RF	$C[i] = A[2i] * A[2i+1]$, $i = 0, 1, 2, ..., n-1$ where $n = (size of input vector)/2$
FPVECACC	[CR] [CW] RF	$C = \sum_{i=0}^{n-1} A_i$, $i = 0, 1, 2, ..., n-1$ where n is the size of the input vector
FPVECPROD	[CR] [CW] RF	$C = \prod_{i=0}^{n-1} A_i$, $i = 0, 1, 2, ..., n-1$ where n is the size of the input vector

Table 3.2 shows four-operand instructions. All four-operand instructions assume that there are two equal size input vectors and these vectors are disjoint, not interleaved. In these instructions, CR0 and CR1 are the starting addresses of the input vectors and CW is the starting address of the output vector. Similar to the three-operand instructions, RF marks the size of the input vectors. With four operand instructions, the vectors are stored in two disjoint contiguous memory blocks as; $A[0]$, $A[1]$, $A[2]$, $\cdots$, $A[n-1]$ and $B[0]$, $B[1]$, $B[2]$, $\cdots$, $B[n-1]$

Table 3.2: Four-operand instructions.

Instruction	Operands	Meaning
`FPVECADDS`	`[CR0] [CR1] [CW] RF`	$C[i] = A[i] + B[i]$, $i = 0, 1, 2, ..., n-1$ where n is size of the input vectors
`FPVECSUBS`	`[CR0] [CR1] [CW] RF`	$C[i] = A[i] - B[i]$, $i = 0, 1, 2, ..., n-1$ where n is size of the input vectors
`FPVECMULTS`	`[CR0] [CR1] [CW] RF`	$C[i] = A[i] * B[i]$, $i = 0, 1, 2, ..., n-1$ where n is size of the input vectors
`FPVECMULTACC`	`[CR0] [CR1] [CW] RF`	$C = \prod_{i=0}^{n-1} A_i B_i$, $i = 0, 1, 2, ..., n-1$ where n is the size of the input vecs.

3.4.5 Module Statistics

Table 3.3 shows the resulting device utilization and maximum clock speed for each module. These values were collected after module placement and routing was completed for an XC4044XL FPGA device [84].

Table 3.3: Device utilization and maximum clock speeds.

Module Name	CLB Utilization	% Utilization	Clk. Speed (MHz)
Adder (Three Operand)	463	28	29.53
Adder (Four Operand)	473	29	30.44
Subtractor (Three Operand)	464	29	30.08
Subtractor (Four Operand)	476	29	30.64
Multiplier (Three Operand)	953	59	28.47
Multiplier (Four Operand)	984	61	27.23
Accumulation	432	27	31.43
Product	944	59	26.44
Multiply-Accumulate	1265	79	25.35

The adder and the subtractor modules use only 28% and 29% of the FPGA device, respectively. This means that three adder or subtractor modules can fit into one FPGA device. On the other hand, since the adder and subtractor cores require only 20% of the device, five cores can fit into one FPGA device. Since the board that we are using has 5 FPGA devices on it, a total of 25 adder or subtractor cores can be utilized on the board. The complete multiplier and product modules require around 60% of an FPGA device, and the multiply-accumulate module requires 79% of an FPGA device. Only one multiplier, product or multiply-accumulate module can fit into one FPGA. Therefore, a total of five multipliers, product or multiply-accumulate modules can be utilized on the board simultaneously.

3.5 The Third Generation Module

Usually, the floating-point core unit of a module is the one that requires more hardware resources than the other parts of the module do. As the size and the density of the FPGA devices increase, we realized that it is possible to place more core units (4 or more) into a single FPGA device at the same time. With some additions to the existing datapath and controller, we designed the third generation module that includes seven function core units. The third generation module brought several advantages over the previous generation. First, instead of using several different modules, we only need to use one module that can perform the same functionality. Second, reconfiguration time is reduced significantly. For example, during the execution of a program, if we need to use several different modules, we do not need to reconfigure the FPGA device every time we need a different module. All we need to do is to utilize the appropriate core unit in the module. Third, with this new module, it is possible to perform some other instructions such as load and store instructions. These features of the third generation module offer more flexibility in mapping applications to the reconfigurable systems.

3.5.1 Function Core Units

In the third generation module, we used function cores units. A function core is a unit that includes one or more of the basic floating-point core units and some other useful parts such as data delay buffers. Basic cores are used as building blocks for the function cores. Figure 3.9 shows two sample function cores. In the first function core, there is only one basic core unit used and it is similar to the cores used in the second generation modules. The second function core, Figure 3.9b, includes two multiplier cores and an adder core. To synchronize the core units, `ENABLE` and `DONE` signal are used. The main controller of the module first reads all the input data into the data registers and then it triggers the function core unit by asserting the `ENABLE` signal to the multipliers. The `DONE` signal of the multipliers are connected to the `ENABLE` input of the adder core through an `AND` gate. When both multiplier cores finish multiplying their input data, they trigger the adder core by asserting their `DONE` signals. The adder core grabs the multiplier outputs and adds them up. When the adder finishes its job it also asserts a `DONE` signal that goes to the module controller. Finally, when the controller receives this `DONE` signal, it writes the final result back to the memory. Using this method, more complicated function core units can be formed.

Using function core units in the module design offers great advantages over the previous generation. Firstly, we can perform complex arithmetic operations with more than 2 inputs. For example, the function core unit in Figure 3.9b can calculate the vector operation $A[i] = B[i] * C[i] + D[i] * E[i]$. Secondly, a single function core can be used pipelined fashion to perform several mathematical operation. Table 3.4 shows the list of expressions that can be calculated using the function core shown in Figure 3.9b. The first expression is the primary expression for this core and the rest of them are secondary expressions. Third, all the mathematical operations are performed in parallel in the function cores where as they are performed sequentially in a general purpose processor.

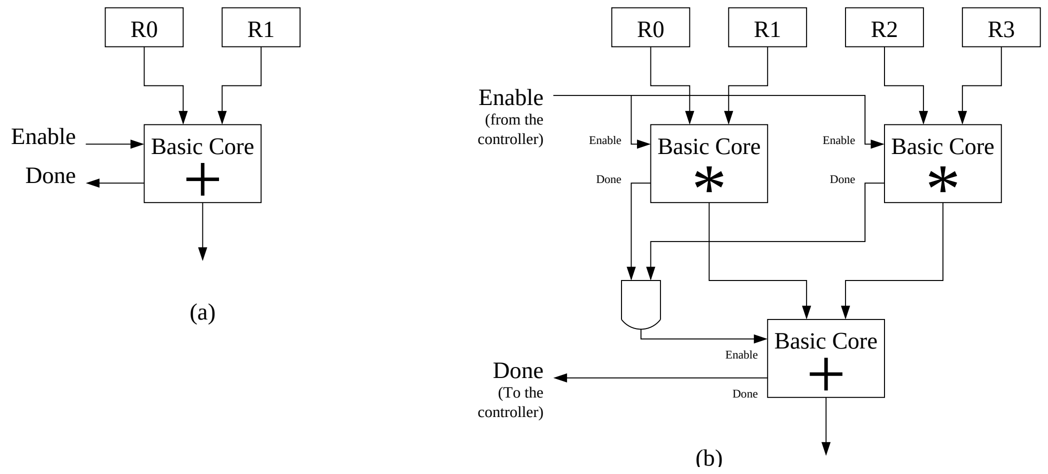


Figure 3.9: Sample function cores. (a) Single arithmetic unit function core (b) Multiple arithmetic unit function core.

Table 3.4: Expression that can be calculated by the function core in figure 3.9b.

Expression	**Comment**
$Y[i] = A[i] * B[i] + C[i] * D[i]$	Regular use
$Y[i] = A[i] * B[i] + C[i] * K$	Load `R3` with a constant K
$Y[i] = A[i] * B[i] + C[i]$	Load `R3` with value 1
$Y[i] = A[i] * B[i] + K$	Load `R2` with K and Load `R3` with value 1
$Y[i] = A[i] * B[i]$	Load `R2` and `R3` with value 0
$Y[i] = A[i] * K_1 + C[i] * K_2$	Load `R1` with K_1 and `R3` with K_2
$Y[i] = A[i] * K + C[i]$	Load `R1` with K and `R3` with value 1
$Y[i] = A[i] * K_1 + K_2$	Load `R1` with K_1, `R2` with K_2 and `R3` with value 1
$Y[i] = A[i] * K$	Load `R1` with K, and `R2` and `R3` with value 0

3.5.2 The Datapath

Figure 3.10 shows the block diagram of the data processor section of the third generation module's datapath. The datapath can include up to 7 function core units and each function core can include several basic cores. The size of the register file limits the number of inputs that a function core can have. Since the data register file has only 8 registers, the number of inputs to the function cores cannot exceed 8 but it can be less than 8. While executing a vector instruction all function cores read the input data from the register file, process it and produce a result, but only one result is written back to the memory. The main controller determines which result needs to be written back to the memory by looking at the instruction decoder output and asserting control signals to the `M0` multiplexer.

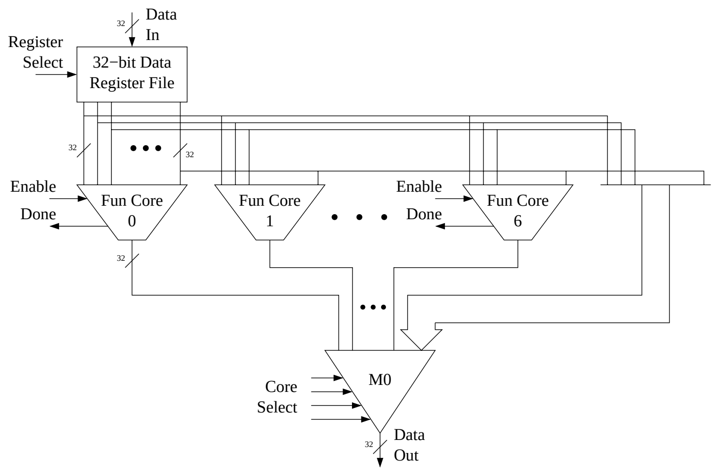


Figure 3.10: Data processor section of the datapath of the third generation module.

As seen in Figure 3.11, the Fetch/Decode section of the datapath includes an address counter file, a counter for the result vector, a program counter, a counter for the input vector size and an instruction decode unit. Each counter in the counter file is associated with a register in the register file. These counters are used to address input vectors. Different from the previous generation, the fetch/decode section has a special instruction decoder. The decoder is used because this module is able to execute not only vector instructions but also Load/Store instructions. These instructions are explained in detail in the following sections.

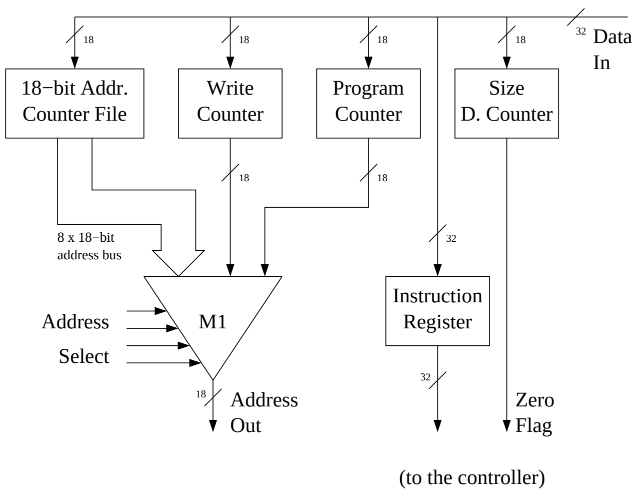


Figure 3.11: Fetch/decode section of the datapath of the third generation module.

3.5.3 The Controller

A special controller has been designed for the third generation module. The controller is able to utilize several function core units to execute a wide variety of vector instructions. During the execution of a vector instruction, first it initializes the register file with constant values, if needed. After that, similar to the second generation, the controller executes the vector instruction using the selected function core. A distinguishing feature of the controller is that, besides the vector instructions, the controller is able to execute Load/Store instructions and execution control instructions.

3.5.4 The Instruction Set

The third generation module is able to execute a wide variety of instructions. It is possible to classify module instructions into three classes, vector instructions, Load/Store instructions, and execution control instructions.

Since there are up to 7 function cores in the module and each core is able to execute a number of vector instructions, it is not possible to enumerate all vector instructions here; therefore, only an example instruction is presented in this section.

Similar to the second generation module, vector instructions start with the name of the instruction and include several operands identifying the start address and size of the input and output vectors. One additional operand, number of input vectors, is used in this instruction format. With the help of this additional operand, the controller is able to use the same function core to execute several different instructions written for several different mathematical expressions. Figure 3.12 shows how two different vector instructions can be executed on the same function core shown in Figure 3.9b. Figure 3.12a is the primary instruction and Figure 3.12b is one of the secondary instructions for this function core.

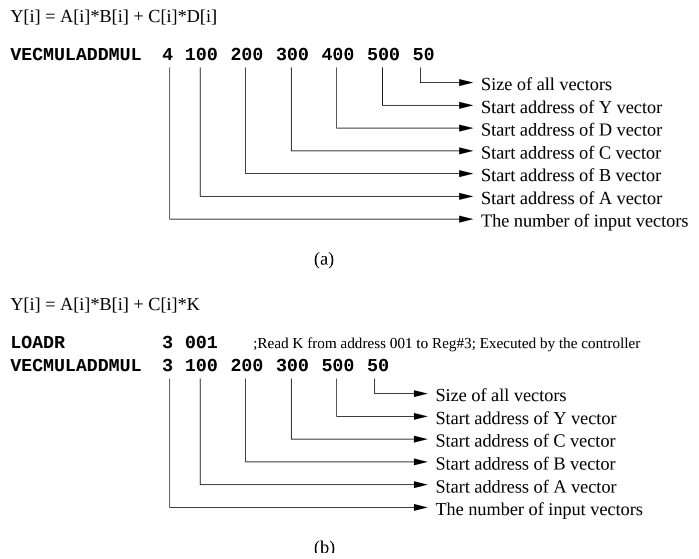

Y[i] = A[i]*B[i] + C[i]*D[i]
VECMULADDMUL 4 100 200 300 400 500 50
Size of all vectors
Start address of Y vector
Start address of D vector
Start address of C vector
Start address of B vector
Start address of A vector
The number of input vectors
(a)
Y[i] = A[i]*B[i] + C[i]*K
LOADR 3 001 ;Read K from address 001 to Reg#3; Executed by the controller
VECMULADDMUL 3 100 200 300 500 50
Size of all vectors
Start address of Y vector
Start address of C vector
Start address of B vector
Start address of A vector
The number of input vectors
(b)

Figure 3.12: Vector instructions format for the third generation module's function cores. (a) Primary instruction. (b) One of the secondary instructions.

Execution of a primary instruction of a function core is the same as execution of the vector instruction in the second generation modules. The controller reads the numbers, triggers the core and when the result is available it writes the result back to the memory. On the other hand, execution of a secondary instruction requires initialization of some input registers with constant values. As it can be seen in Figure 3.12a a `LOADR` (Load Register) instruction is used to initialize register #3 with constant `K`.

In the third generation module, the opcodes are formed by combining two numbers, n and m. The first number, n, indicates the location of the function core in the module that executes the given vector instruction. Since there are only seven positions in the module that can be loaded with function cores n can take values from 0 to 6. The second number, m , indicates the number vector operands in the vector instructions. Using this approach in the instruction set allows the reuse of particular opcodes for different instructions. For example the opcodes two input vector adder, subtractor, multiplier and divider are the same when the function cores for these instructions are located in the same position in the module datapath.

This approach combines the advantages of RISC and CISC processor. Similar to the RISC processors, the number of instructions that the module controller has to decode was reduced significantly. As result, the decode logic was simplified and the controller's clock rate was increased. On the other hand, potentially each location in the module core can be configured with an infinite number of unique function cores to execute an infinite number of vector instructions. This feature of the instructions set provides the benefits of the CISC processors. The module assembly language instruction set for some function cores is presented in Appendix B.

Table 3.5 summarizes the Load/Store instructions and the execution control instruction. With the help of these instructions, the module becomes more flexible and is able to utilize function cores more efficiently.

Table 3.5: Execution control and load/store instructions.

Instruction	**Comment**
`RUN`	Start running
`PAUSE Cycl#`	Stop a core or a pipelined unit for a given # of cycles
`STOP`	Stop a core or a pipelined unit for an indefinite time
`HALT`	Completely stop the system.
`LOADR Reg#, [Address]`	Load data from a give address to a data register.
`LOADRIMM Reg#, Data`	Load immediate data to a data register.
`LOADC Cnt#, [Address]`	Load data from a given address to a Index Counter.
`LOADCIMM Cnt#, Data`	Load immediate data to an Index Counter.
`STORER Reg#, [Address]`	Store data from a data register to a given addr.
`STOREC Cnt#, [Address]`	Store data from an Index Counter to a given addr.

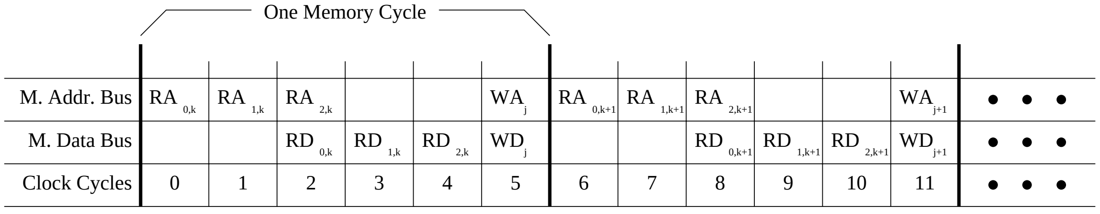
3.6 Mathematical Models for the Third Generation Modules

Performance of a module highly depends on the *memory cycle* of the function core unit which is utilized for the given vector operation. The *memory cycle* is the number of clock cycles during which the module completes reading one set of input data and writing one result back to the memory. The *Memory Cycle* of a function core unit can be calculated using Equation 3.5.

$$Memory\ Cycle = R_C + R_L + W_C + W_L \tag{3.5}$$

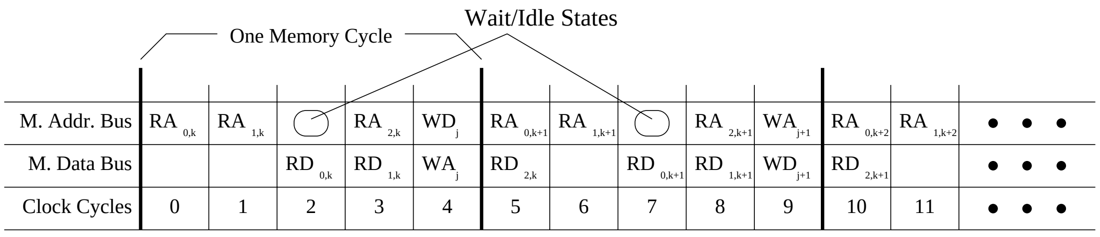
where, R_C is the number of input vectors to the function core unit and R_L is Memory Read Latency. W_C is equal to the number of results produced per *Memory Cycle* and W_L is Memory Write Latency. Memory write latency for most memory types is zero and can be eliminated from the equation. Figure 3.13a shows a sample memory cycle. In this figure, the module has to read data from three vectors and it has to write the result back to memory. The read latency is two and the write latency is zero. In Figure 3.13a and 3.13b, $RA_{i,k}$ is the read address asserted on the memory address bus by the module for the k^{th} element of the i^{th} input vector, $RD_{i,k}$ is the data asserted by the memory for the k^{th} element of the i^{th} input vector, WA_j is the write address and WD_j is the output data asserted by the module for the j^{th} element of the output vector.

The *Memory cycle* in Figure 3.13a can be improved by inserting a wait/idle state (a clock cycle during which the controller does not do a memory read request) in the read sequence before the last read request as shown in Figure 3.13b. This eliminates the effect of the write operation on the *memory cycle* and *memory cycle* becomes equal to $R_C + R_L$. Improvement in *memory cycle* through wait/idle state insertion is a special case and can only be applied if $R_C >= 2$ and $R_L = 2^i$, where $i = 1, 2, 3, \cdots$.



M. Addr. Bus	$RA_{0,k}$	$RA_{1,k}$	$RA_{2,k}$			WA_{j}	$RA_{0,k+1}$	$RA_{1,k+1}$	$RA_{2,k+1}$			WA_{j+1}	● ● ●
M. Data Bus			$RD_{0,k}$	$RD_{1,k}$	$RD_{2,k}$	WD_{j}			$RD_{0,k+1}$	$RD_{1,k+1}$	$RD_{2,k+1}$	WD_{j+1}	● ● ●
Clock Cycles	0	1	2	3	4	5	6	7	8	9	10	11	● ● ●

(a)



M. Addr. Bus	$RA_{0,k}$	$RA_{1,k}$	○	$RA_{2,k}$	WD_{j}	$RA_{0,k+1}$	$RA_{1,k+1}$	○	$RA_{2,k+1}$	WA_{j+1}	$RA_{0,k+2}$	$RA_{1,k+2}$	● ● ●
M. Data Bus			$RD_{0,k}$	$RD_{1,k}$	WA_{j}	$RD_{2,k}$		$RD_{0,k+1}$	$RD_{1,k+1}$	WD_{j+1}	$RD_{2,k+1}$		● ● ●
Clock Cycles	0	1	2	3	4	5	6	7	8	9	10	11	● ● ●

(b)

Figure 3.13: Sample memory cycles: (a) Memory cycle for the module that does three reads and one write, (b) Improved memory cycle for the same module.

Execution time of one vector instruction can be calculated with Equation 3.6.

$$I_{EX} = \frac{Memory\ Cycle * Vector\ Size + F_C + E_C}{Clock\ Rate} \tag{3.6}$$

where, F_C is the number of clock cycles required to fetch a vector instruction and E_C if the number of clock cycles required to empty a pipelined core unit.

In a session, a PE is assigned to execute a number of vector instructions. By multiplying the result of Equation 3.6 with the vector instruction count for a session and adding the Application Programming (API) overhead, we can calculate the execution time for the session, Equation 3.7.

$$T_{EX} = \left[\left(\frac{Memory\ Cycle * Vector\ Size + F_C + E_C}{Clock\ Rate}\right) * Inst.\ Count\right] + C_{API} \quad (3.7)$$

where, C_{API} is API overhead. By substituting Equation 3.5 into Equation 3.7 we get Equation 3.8 which is the mathematical model used by the Simulator for the modules.

$$T_{EX} = \left[\left(\frac{(R_C + R_L + W_C + W_L) * Vector\ Size + F_C + E_C}{Clock\ Rate}\right) * Inst.\ Count\right] + C_{API} \quad (3.8)$$

Equation 3.8 is a very generalized mathematical model for the modules. It has to be reevaluated for different types of modules. For example, an accumulation module does not write a result at every memory cycle rather it writes one result at the end of the vector instruction. This situation makes W_C and W_L zero. Since the module does not need to write back at every *Memory Cycle* it can continuously read from the memory in a pipelined fashion. This means that the module does not need to wait idle for the data it requested during the Memory Read latency period. In one clock cycle, the module can request input data and at the same time it can read the data previously requested. This eliminates R_L. The module waits idle for memory read latency only once at the beginning of the vector instruction which is negligible. With this information, we can rewrite Equation 3.8 as Equation 3.9 for the accumulation modules.

$$ACC_{EX} = \left[\left(\frac{R_C * Vector\ Size + F_C + E_C}{Clock\ Rate}\right) * Inst.\ Count\right] + C_{API} \quad (3.9)$$

Equation 3.9 is also valid for two memory unit accumulation or non-accumulation modules. Since either type of module does not write back to the input memory, they do not interrupt data reading from the input memory. Modules can continuously read from the input memory and write to the output memory if it needs to.

3.7 Module Execution versus GPP Execution

Although modules run at slower clock speeds, they are typically able to outperform much faster general purpose processors. This situation could be explained with the following example. Let us assume that the code fragment in Figure 3.14 is a part of a user application implemented in C/C++.

```
for (i=0; i<1000; i++)
   Result[i] = Input1[i] + Input2[i];
```

Figure 3.14: Sample code fragment.

The purpose of the code in Figure 3.14 is to perform the addition operation on individual elements of two input vectors and store the results in the `Result` vector. When this section is compiled with a C/C++ compiler, the compiler will produce machine code similar to the code in Figure 3.15. (Note: Compilers generate processor-specific code. Here, we demonstrate a generalized form of different machine code formats [85].) Lets assume that `R0`, `R1` and `R2` hold the starting addresses of vectors `Result`, `Input1` and `Input2`, respectively and `R3` holds the index value.

As it can be seen in Figure 3.15, to process one element from each input vector and to produce one result, a general purpose processor spends 19 clock cycles total. The above clock cycle numbers are for the best case. In the case of cache misses, a general purpose processor spends even more clock cycles to execute the above code. Moreover, a general purpose processor also spends extra clock cycles for operating system overhead. On the other hand, the same C/C++ code can be compiled to a single vector instruction for a specific hardware module as follows:

```
FPVECADD Addr1, Addr2, Addr0, Size
```

where `Addr1`, `Addr2`, and `Addr0` represent the starting addresses of `Input1`, `Input2` and `Result`, respectively and `Size` represents the size of the vector operation.

```
Label   OpCode  Operands      Clock Cycle
===========================================
        LOADW   R3, #1000           2
Loop:   LOADF   F1, 0(R1)           2
        LOADF   F2, 0(R2)           2
        ADDF    F0, F1, F2          3
        STORE   0(R0), F0           1
        ADDI    R0, R0, #2          2
        ADDI    R1, R1, #2          2
        ADDI    R2, R2, #2          2
        SUBI    R3, R3, #1          2
        BNEZ    R3, Loop            1
===========================================
                      Total cycles  19
```

Figure 3.15: Machine code for source code in Figure 3.14.

Since general purpose processors are designed to execute a wide variety of instructions, they have to execute a series of machine instructions repeatedly to perform a simple vector operation. At each iteration it has to fetch, decode and execute the same set of instructions. As a result, it spends a lot of time fetching and decoding the same set of instructions repeatedly. On the other hand, hardware modules are specialized on execution of a single vector instruction. They only fetch and decode a few instructions at the beginning of each vector operation and then perform the operation. Another advantage of the modules is that they can perform arithmetic operations in pipelined fashion depending on the numbers of basic cores used in the module. For example, the multiply-accumulate module multiplies numbers and at the same time it accumulates the multiplication results of the previous operation. And lastly, if multiple FPGAs are used, we can execute several instructions in parallel. These are the reasons why modules are able to outperform general purpose processors.

Chapter 4

The RCCT Compiler

The Compiler is the most important part of the Reconfigurable Computing Compilation Tool (RCCT). The purpose of the Compiler is to map user applications developed in C/C++ to Reconfigurable Computing (RC) environments to enhance performance of them. In this chapter, detailed information about the Compiler is provided. The chapter begins with an overview of the Compiler. The next section presents the inputs of the Compiler. The subsequent sections give detailed information about the sections of the Compiler.

4.1 An Overview of the RCCT Compiler

A compiler takes as input, source code, usually written in a high level programming language, and translates it to an equivalent representation in another language, usually to a machine language that is specific to a processor or a computer architecture. In our case, the RCCT Compiler takes the original source code written in C/C++ and translates it into a new C/C++ source code by modifying the computationally complex sections of it. It also produces one or more session files that contain machine language instructions for a standard reconfigurable processor.

General Purpose Processors (GPPs) perform very well on single mathematical operations. On the other hand, a well designed and optimized RC system offers excellent performance gains on vector operations compared to GPPs. For that reason, the Compiler targets computationally complex sections of the source code contained in nested loops and translates them into vector operations that are executed on a RC system. The Compiler targets the `for` loops, because they are the ones most frequently used to perform vector operations. The Compiler must calculate the iteration count of the loops being mapped to an RC system at compile time. The iteration counts of `while` and `do-while` loops can not be determined at compile time; therefore, these loops were not considered due to their indeterministic nature.

During the compilation process, the Compiler first determines the target sections of the user applications that are going to be mapped to the RC system. Then, it identifies the vector operations in these sections and writes them into one or more session files. These session files allow the Loader to run that section of the application on one or more FPGA devices. Next, the Compiler rewrites the source code by replacing these target sections with function calls to the Loader. The Compiler inserts four types of Loader functions in each modified section of code. These are, `IsModuleAvailable()` to check the availability of the hardware module that executes the given section on an RC system, `Store_Data()` to transfer data from host memory to the RC system, `Load_Data()` to transfer data from the RC system to the host memory, and `run()` to start execution of the given section on the RC system.

Figure 4.1 shows the top level flow chart of the Compiler. Besides the application source code, the Compiler reads a module definition file and a parameter file. The module definition file provides information about the hardware modules that the Compiler can use while mapping the user applications. The parameter file provides information about the RC environment that the user application is being mapped to. The parameter file also includes user preferences about the mapping process. Detailed information about these files is provided in Section 2 of this chapter. The Compiler outputs a modified user source code and one or more session files for each modified section of the user source code. Details of the syntax of a session file is found in section 5.2

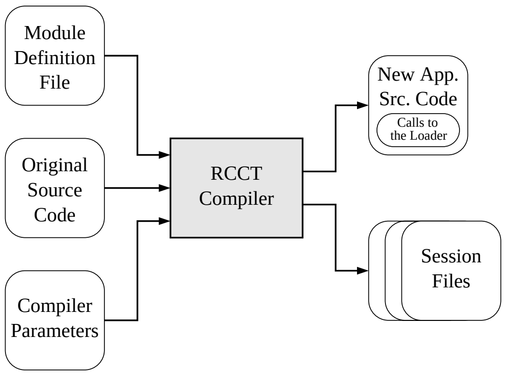


Figure 4.1: Top level flow chart of the Compiler.

Figure 4.2 shows the second level flow chart of the Compiler. The Compiler uses the same scanner to scan all three input files. On the other hand, different parsers were developed to parse each input file. Special parsers were implemented to parse the module definition file and the parameter file. A unique parser was designed and implemented to parse the user source code. The user source code parser is able to identify computationally complex sections of the given source code and form a parse tree for these sections. It is also able to generate symbol tables for these sections.

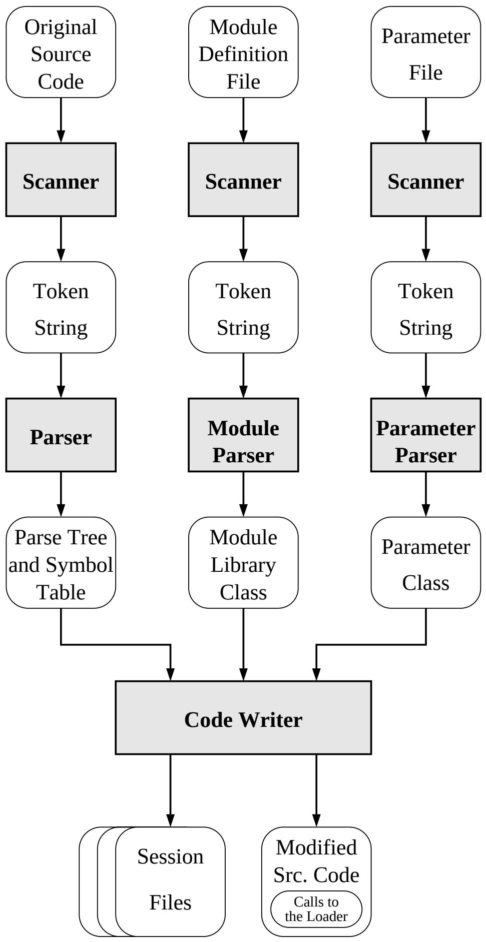


Figure 4.2: Second level flow chart of the Compiler.

The code writer is responsible for the rest of the mapping process. It takes the parse tree, produced by the parser, and performs several transformations before it writes the new source code and session files. Some of the steps that the code writer goes through are: identifying appropriate hardware modules for the parsed expressions in the parse tree, performing data dependency analysis, eliminating the sections that can not be mapped to the RC system, scheduling instructions to available PEs and organizing the memory assignments. Details of these steps are given in section 4.5.

4.2 Inputs to the Compiler

The Compiler reads three files. These are: the source code of the user application, a module definition file and a parameter file. In the following subsections, these input files are presented.

4.2.1 User Application

The current version of the Compiler accepts user applications implemented in C/C++. We selected C/C++ as the input source code Language for the RCCT Compiler due to the fact that it is one of the most popular high level programming languages used for scientific data processing. Another reason for selecting C/C++ is that, several typical RC Application Programming Interfaces (APIs) were developed in C/C++ language. Using an API developed in the same language as the user application, facilitates interfacing the user application with the RC. Although we selected C/C++, the Compiler can easily be adapted for other languages. Different languages such as Fortran or Java can be compiled with little modification to the Compiler. Only the front end, scanner and parser, needs to be modified.

4.2.2 Module Definition File

The module definition file includes specifications of the hardware module implementations available for the Compiler to use while mapping user applications to the RC system. Figure 4.3 shows a sample module definition file. In the module definition file, single line comments, that must start with "//" are allowed. Specification of a module should start at the beginning of the line and should end in the same line. Users can define two types of modules, regular modules and conditional modules. A "." before the module name indicates that the module is a regular module and a "-" indicates that the module is a conditional module.

```
//Sample Module Library

//MODULE_NAME    GRAMMAR                              OP1 OP2 OP3 OP4 OP5 OP6
//----------     ----------------------------------- --- --- --- --- --- ---
    //A second generation module
.FPVECADDS       VF[1]= VF[1]+VF[1];                  2   2   1   8   12  100
    //A third generation module
.FPVECACC        VF[0]+=VF[1];                        1   2   0   8   61  100
+FPVECMULACCS    VF[0]+=VF[1]*VF[1];                  2   2   0   0   62  100
+FPVECCOMP       VF[1]= VF[1]+(VF[-1]+VF*1.0)*VF[1]; 3   2   1   32  39  100
    //conditional modules
-FPEQ            VF[1]==VF[1];                        1   2   0   0   3   100
-FPBETWEEN       (VF[1]<VF[1]) && (VF[1]<VF[1]);      4   2   0   0   5   100
-FPLTC           VF    <VF;                           0   2   0   0   3   100
```

Figure 4.3: Sample module definition file.

Since the second generation modules include only one core unit, these modules can be specified with a single entry in the module definition file. In contrast, the third generation module includes more than one function core, and each core unit has to be specified as an entry in the definition file. The specification of the first function core unit must start with ".", and the specification of the other function core units combined in the same module have to start with "+" and should immediately come after the first function core specification. Since the specification of a second generation module and specification of a third generation function core are the same, we need only to explain the second generation module specification.

The user has to specify eight parameters for each type of module. The first parameter is the name of the module. Module names must be all uppercase and the first character must be a letter. Users are not allowed to use the "_" character in the module names, because this character is used by the Compiler to represent the combination of conditional and regular modules internally.

The second parameter is the module grammar. A module grammar is a mathematical expression that specifies the vector operation performed by the module on given vectors. In the module grammar, to specify integer constants and floating-point constants "`1`" and "`1.0`" strings must be used, respectively. Integer and floating-point variables can be specified by the words "`VI`" and "`VF`", respectively. Words "`VI`" and "`VF`" followed by square brackets are used to specify integer and floating-point vectors, respectively. Inside the square brackets, the user must define the amount of change or delta in the index values of the multi-dimensional vectors at each iteration while the module performs the vector operation. For example [1] means increment by one. In the module grammar, parentheses are allowed to define the order of the mathematical operations. Module grammars must end with ";" to assist the module definition file parser in detecting the end of a module grammar. The difference between the regular module grammar and the conditional module grammar is that in regular module grammars, relational operators are not allowed. In conditional module grammars, both relational and arithmetic operators can be used.

After the module grammar of functional definition, six module timing parameters must be specified. These parameters are used in the estimation of the module's execution time. The first parameter indicates the number of memory read operations the module has to perform at each iteration. The second parameter specifies the memory read latency in units of clock cycles. The third parameter specifies if the module writes back to memory at each iteration of the vector operation. For accumulation modules, this parameter must be set to "0" and for other types of modules it must be "1". Pipeline depth of the module is specified by the fourth parameter. The fifth parameter specifies the amount of time needed to empty a pipelined module and fetch a module instruction in units of clock cycles. The last parameter is reserved for future use.

The user of the RCCT tool can also include specifications of modules that have yet to be implemented. In such case, the Compiler assumes that the modules are available and it uses the specifications, to map the user application. On the other hand, when the user tries to execute the compiled application and when the execution order comes to the section for which a module specification is to execute that has not been implemented, the Loader activates the Simulator instead of accessing the actual RC system. The Simulator behaves like the module defined in the module definition file and executes the section. The Simulator returns the processed data along with an estimated execution time to give the the user an estimate of the performance that would be obtained using these hypothetical module designs.

4.2.3 Parameter File

The Compiler reads information about the target RC environment and the user preferences about the compilation process from the parameter file. There are nine parameters that the user can set before compiling applications. If the user does not set a value to a parameter, the Compiler uses the default value for that particular parameter.

In the parameter file, each parameter must be declared on a separate line. The format for the parameter declaration is that the declaration must start with the <*Parameter name*> and must be followed by the <*Value*> of the parameter.

All parameter names and values must be upper case. The following is a list of parameters along with their explanations.

- `OPERANDSEQMODE`: With this parameter, the user can indicate the order of the vector instructions' operands that the Compiler writes for modified sections of user applications. This parameter can take three values.
    - `ORGMODE`: In this mode, the Compiler does not change the order of operands. It keeps them in the order they appear in the source code. (Default).
    - `REGMODE`: The operands are placed in the following order. <*Output Op.*>, <*Inputs Ops.*> (Variables first), <*Size*>
    - `NEWMODE`: The operands are placed in the following order. <*Inputs Ops.*> (Variables first), <*Output Op.*>, <*Size*>
- `ASSIGNMEMMODE`: Can be used to indicate how vector instructions are extracted from each `for` block and scheduled to the PEs.
    - `SINGLEBLOCK`: All vector instructions of a `for` block are scheduled to one PE. (Default).
    - `SINGLEINST`: Each set of vector instructions of a `for` block is assigned to a PE.
    - `PARALLEL`: Each set of vector instructions is scheduled to all available PEs for parallel execution.
- `WORDSIZE`: Different RC systems can use different word lengths while addressing data in memory. The user can specify the word length with this parameter. (Default = 2).

- `BASEOFFSET`: Using this parameter, the user can tell the Compiler to start using RC memory starting from a base address. (Default = 1).

- `CLOCKSPEED`: Clock speed of the RC system can be defined with this parameter. (Default = 50).

- `VARNUMBER`: Controller of the third generation module needs to know how many of the input operands of a vector instruction are variable. If the applications are mapped to the third generation module, then this parameter must be set to `TRUE`. (Default = false).

- `PE_NUMBER`: The number of available PEs can be specified by this parameter. (Default = 1).

- `OUTPUTADDRESS`: Some dual memory unit PEs, in fact, have one memory unit but two ports to the same memory. The PE can read data from one port, while writing data to the other port. The output port usually starts from a specific memory location. The user can specify the starting address of the output port with this parameter. (Default = -1);

- `ASSIGNSIZE`: This parameter is used to specify the vector size form. The <*size*> operand in each vector operation can be the actual size of the vector operation or the offset size.

  – `ORIGINALSIZE` Original size of the vector operation if written in the vector instruction.(Default = `ORIGINALSIZE`).

  – `OFFSETSIZE` The offset size is calculated by adding the actual size of the vector operation to the start address of the first input vector.

4.3 Scanner

The Scanner, also called the lexical analyzer, reads the source code, which is usually in the form of ASCII character string, and divides into meaningful units called tokens. In typical source code, reserved words of the language, numbers, identifiers, some special character sequences, delimiters, comment and blank spaces are all considered tokens. Some scanners eliminate unnecessary tokens such as blank spaces and send the meaningful tokens to the next step of the Compiler.

A scanner can be hand written/coded or automatically generated. There are some tools such as *lex* and *flex* available that generate source codes for scanner. These tools accept a specification file in which tokens and instructions for the scanner tool are specified. The tool converts the given specification file into table driven source code usually generated in C..

The scanner of the RCCT Compiler was generated using the *lex* tool available in Unix environment. We started with Jef Lee's *lex* specification [86] as the specification file for our scanner. This specification file tells the scanner generator to only return the token itself. We modified the specification and made the scanner return additional information about the tokens to make the parsing process easier and to help the Compiler create the modified source code. The current version of the scanner returns the tokens, the token text, and classification information about the tokens. After the compilation process, the Compiler uses the token text to rewrite the source code. The specification file for the scanner used in this thesis can be found in the Appendix.

After we generation of the scanner program using *lex*, we converted it to a C++ class. Converting the scanner to a C++ class lets us use it, not only to scan user applications but, also to scan the module definition file and the parameter file.

4.4 Parser

A parser takes the source code in the form of tokens from the scanner and performs syntax analysis. During the syntax analysis phase, the parser determines the structure of the program. The parser creates a parse tree as an output and forwards it to the next step of the Compiler.

Similar to the scanner, a parser can also be generated using tools, such as *yacc* and *bison*. Such a tool accepts a grammar file in which the grammatical rules of the Language are specified and outputs source code for a program that performs the parsing operation.

Since the RCCT Compiler focuses only on a subset of C/C++ statements, we decided not to use parser generators and to implemented our own parser. The current version of the parser only parses `for` *loop statements*, *expression statements*, (arithmetic and conditional), and *conditional statements* (`if` statements). Future version of the Compiler will parse additional and more complex statements. One or more recursive parser functions were implemented for each statement. In this section, the data structure used to represent the parse tree is introduced first. Then, the parser functions and algorithms are explained.

4.4.1 Parser Data Types

Figure 4.4 shows the data type definitions for the parser. Note that it is not possible to present all details of the type definitions here; therefore, only the important ones are depicted in the figure and explained in the text.

Expression

NodeEnum	NodeType
char	*NodeName
Expression	*Index
Expression	*Left
Expression	*Right

(a)

IfNode

bool	Predicate
Expression	*Conditional
Expression	*ExpTrue
Expression	*ExpFalse

(c)

ForNode

int	NestLevel
Expression	*E1
Expression	*E2
Expression	*E3

(b)

Statement

STEnum	STType
Union	
ForNode	F
Expression	*E
IfNode	*I
Statement	*Sub
Statement	*Next

(d)

Figure 4.4: Data type definitions for the Parser.

For expression statements, `Expression` type is defined as shown in Figure 4.4a. This data type is used to form binary expression trees. The parser uses this type to store information about individual elements of expression trees. `Left` and `Right` fields are used to link left and right subtrees. The `Index` field is used to link indexing information of the array variables. This type definition is used for both arithmetic expressions and conditional expressions.

Figure 4.4b shows the data type definition for `for` statements. Since a `for` statement can include three sets of expressions, three expression fields, `E1`, `E2` and `E3` were added to the definition to link these expression sets.

The type definition for the `if` statement is shown in Figure 4.4c. The `Predicate` field is used to store the result of the evaluation of the conditional expression of the `if` statement. The `Conditional` field is used to link the conditional expression of the `if` statement. Since we only map arithmetic expressions under `if` statements to RC systems, we defined links for only the arithmetic operations under the `IfNode` data type. These are `ExpTrue` and `ExpFalse`.

Figure 4.4d shows the top level data type definition for statements. This is a wrapper definition for the other definitions. It starts with `STType`. This field holds the type information of the statement. The next field is a union of the types `Expression`, `ForNode`, and `IfNode`. Union is used to conserve memory. The `Sub` field is used to link statements under another statement. For example, expression statements under a `for` statement are linked to the `for` statement with this field. The `Next` field is used to link a series of statements in a block under a statement.

4.4.2 Parser Functions and Algorithms

The `main()` function of the Compiler calls the `Parse()` function of the Parser to parse the input source code. Figure 4.5 shows the algorithm of the `Parse()` function. The algorithm starts by forming a token list. By calling the Scanner, a linked list of tokens is formed. At this stage the parser does not eliminate unnecessary tokens such as comments and blank spaces because these tokens are used while rewriting the source code.

After forming a token list, the Parser enters a while loop and passes through the tokens. It parses the source code in a single pass. When the Parser finds a `for` reserved word while scanning the token list, it calls the `ParseFor()` function to parse the `for` loop block. The `ParseFor()` function returns a pointer to a `ForNode` variable if the `for` loop is successfully parsed or else it returns `NULL`. If the return of `ParseFor()` is not `NULL` then it is added to the end of the parse tree list.

```
Parse()
    Statement *$ST$;
    $PTHead \leftarrow$ **new** Statement;
    ◇ Scan the input file and form a linked list of tokens.
    **while** (! End of token list) **do**
        **if** (Current token is a RC Directive) **then**
            ◇ Skip tokens until the end RC directive.
        **end if**
        **if** (Current token is for) **then**
            $ST \leftarrow$ ParseFor();
            ◇ Add $ST$ to the linked list headed by PTHead
        **end if**
    **end while**
    **return** PTHead;
```

Figure 4.5: Algorithm for the `Parse()` function

The Compiler is able to process a few `RC directives`. Inside the loop, if the Parser sees an `RC directive`, it skips the tokens until the end of the `RC Directive`. These sections of the source code are processed by the code writer of the Compiler.

Only a limited number of statements can be mapped to the RC systems. These are: `for` loops, arithmetic expressions that perform vector operations, and `if` statements. Four functions were implemented to parse these statements. These functions are `ParseFor()`, `ParseMultStatement()`, `ParseExpressionStatement()` and `ParseIf()`. The parser calls the `ParseFor()` function and starts a sequence of recursive function calls to parse a `for` loop block. Then, these four functions call each other recursively by following the structure of the block to parse a `for` loop block. At any point, if a function detects an error, it returns `NULL` to the calling function. This `NULL` value is returned through all recursive calls and subsequently reaches the `Parse()` functions. When the `Parse()` function receives a `NULL` value, it eliminates the current `for` block and keeps searching for another one.

Each function returns a pointer to a `Statement` variable. The calling function links the return of a called function to the current `Statement` variable's `Sub` link to form a parse tree if the return value is not `NULL`.

4.4.3 Parsing a `for` Loop

Figure 4.6 shows the algorithm developed to parse `for` loops and the statements found in the `for` loop body. The algorithm first parses the header of the given `for` statement. It calls the expression parser to parse initialization, termination and increment expressions of the loop. After parsing the header, it calls the other functions to parse the body of the loop. If the next token is a curly bracket ("{"), then, it calls the `ParseMultStatement()` function to parse a block of functions. If the next token is an identifier then it calls the `ParseExpressionStatement()` to parse the expression. At this point, `ParseFor()` assumes that all statements starting with an identifier are expression statements. The `ParseExpressionStatement()` function checks if the statement is an expression statement or not. If the statement is not an expression statement then `ParseExpressionStatement()` returns `NULL`.

If the next token is the beginning of another `for` statement then `ParseFor()` calls itself recursively to parse the inner loop. As was mentioned, only a limited number of statements can be mapped to the RC systems using RCCT; therefore, `ParseFor()` returns `NULL` if it detects a statement other than the ones mentioned above.

```
ParseFor()
    Statement *$STCurrent$;
    $STCurrent \leftarrow$ new Statement;
    $STCurrent \rightarrow$STType $\leftarrow$ FOR_STATEMENT;
    ◇ Parse the header of the for statement.
    switch (Current Token)
        case "{":
            $STCurrent \rightarrow Sub \leftarrow$ ParseMultStatement();
        case Identifier:
            $STCurrent \rightarrow Sub \leftarrow$ ParseExpressionStatement();
        case "if":
            $STCurrent \rightarrow Sub \leftarrow$ ParseIf();
        default:
            ◇ Print an error message.
            return NULL
    end switch
    return $STCurrent$;
```

Figure 4.6: Algorithm for `ParseFor()` function.

4.4.4 Parsing an `if` Statement

The algorithm for the `ParseIf()` function is shown in Figure 4.7. It starts with parsing the header of a given `if` statement. The conditional expression of the `if` statement is parsed by the `BuildConditionalExp()` function.

After parsing the header, the function parses the body of the `if` statement. The current version of the Compiler allows only arithmetic expressions under the `if` statement due to the fact that it is not practical to design a module to execute a combination of the other statements under an `if` statement; therefore, the `ParseMultStatements()` function was not used here. Instead, we developed a special **switch-while** combination shown in Figure 4.7 (lines 08 through 21) to parse the true part of the `if` statement. The arithmetic expressions are linked to the *ExpTrue* field of the *IfCur* variable. The same lines are repeated if the `else` part of the `if` statement is present, but in this case, the expressions are linked to the *ExpFalse* field of the *IfCur* variable. In each part of the `if` statement, once a statement other than an arithmetic expression is detected, the `ParseIf()` function returns `NULL`. Currently, no **else-if** chains are allowed.

```
01 ParseIf()
02    Statement *STCurrent;
03    IfNode *IfCur;
04    Expression *ExCur;
05    STCurrent ← new Statement;
06    STCurrent↦STType ← IF_STATEMENT;
07    ◇ Parse the header of the if statement.
08    switch (Current Token)
09        case "{":
10            while (Current Token is not "}") do
11                if (Current Token is an Identifier) then
12                    ExCur ← BuildExpressionTree();
13                    ◇ Add ExCur to IfNode↦ExpTrue
14                else
15                    return NULL;
16                end if
17        case Identifier:
18            ◇ Repeat lines 12 and 13
19        default:
20            return NULL;
21    end switch
22    if (Current Token is else) then
23        ◇ Repeats the lines 08 through 21 but add expressions to IfNode↦ExpFalse
24    return STCurrent;
```

Figure 4.7: Algorithm for `ParseIf()` function.

4.4.5 Parsing Arithmetic and Conditional Expressions

Different implementations of the same expression parsing algorithm are used to parse arithmetic and conditional expressions. The difference between these two implementations is that the conditional parser implements the same algorithm in two levels. In the lower level, it only sees the arithmetic operators and operands as entities to parse in the given conditional expression. It parses these entities by calling the arithmetic expression parser. It exits from the lower level whenever it sees a conditional operator. In the upper level, it sees the conditional operators and arithmetic expressions parsed in the lower level as entities to parse and it applies the same algorithm. Since the conditional operators have lower priority than the arithmetic operators, they are considered at the upper level.

By implementing the same algorithm at multiple levels, expressions that include an arbitrary number of different operators with different priority levels can be parsed. The highest priority operators have to be parsed in the lowest level and the lowest priority operators have to be parsed in the upper most level. At each level, two priority levels can be parsed. For example, since the "+" and "-" pair and 'the '*" and "/" pair have two different priority levels, expressions including these operators can be parsed in one level. With a few modifications to the algorithm, the unary operators and increment and decrement operators can also be parsed in one level together with "+", "-", "*", and "/" operators.

Since the conditional and arithmetic expression parsers have the same algorithm, we only explain the arithmetic expression parser which is the `BuildExpressionTree()` function. 603 lines of C++ code was written to implement the `BuildExpressionTree()` function.

Figure 4.8 shows the algorithm for the expression parser. We will explain this algorithm with the example shown in Figure 4.9. The algorithm includes four steps. In the first step, the given expression is divided into pieces. A piece can include a lower priority level operator (in this case "+" and "-") or a group of operands and operators in which all the operators have higher priority level (in this case "*" and "/") and all operands and operators are consecutive. In this step everything inside a set of parentheses is parsed within a single recursive call and treated as an operand. The result operand and assignment operator are also counted as pieces. A stack is formed for each piece and by linking these stacks in the order they appear in the expression, a list of stacks is formed. Figure 4.9a shows the stack list for the expression in the same figure. In the figure, shaded nodes are empty nodes used to link stacks or binary trees.

```
BuildExpressionTree()
    Step 1 Form stack list for a given expression.
    Step 2 Convert the stack list to a list binary trees.
    Step 3 Built the final binary expression tree.
    Step 4 Return the final binary tree.
```

Figure 4.8: Algorithm for `BuildExpressionTree()` function.

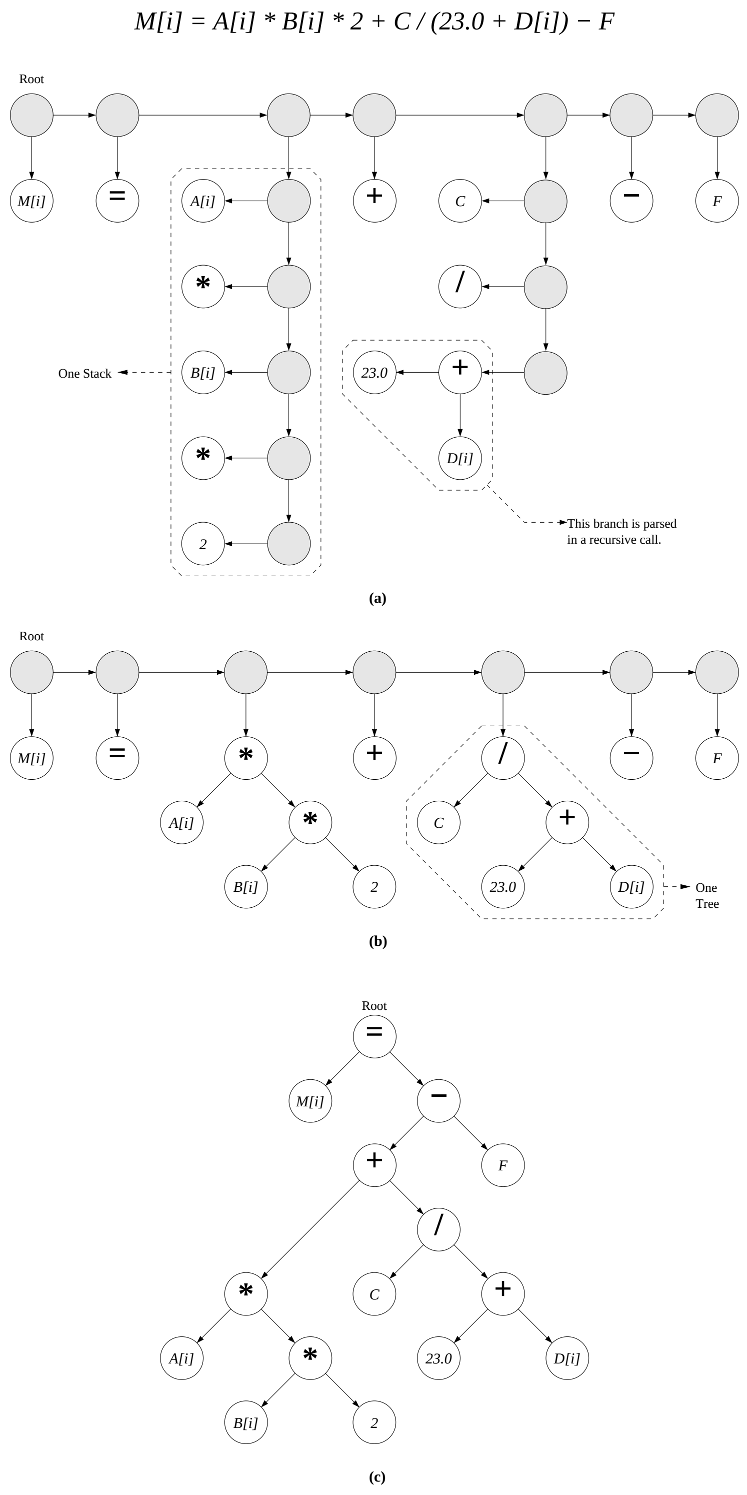

M[i] = A[i] * B[i] * 2 + C / (23.0 + D[i]) − F
Root
M[i]
=
A[i]
*
B[i]
*
2
+
C
/
23.0
+
D[i]
−
F
One Stack
This branch is parsed in a recursive call.
(a)
Root
One Tree
(b)
Root
(c)

Figure 4.9: Steps of the `BuildExpressionTree()` function on an example expression: **a)** Stack list after the first step, **b)** List of binary trees after the second step, **c)** Final binary tree after the third step.

In the second step of the algorithm, the function passes through the stack list once and converts each stack to a binary tree. The conversion procedure is applied to each vertical stack. Figure 4.9b shows the binary tree list equivalent of the stack list in Figure 4.9a. The third step of the algorithm assumes that the binary tree list is one horizontal stack and the root of each tree is a node in the stack. With this assumption, the stack conversion procedure is applied to shaded nodes (horizontally) to form the final binary tree. Figure 4.9c shows the final binary tree for the expression in the same figure. In the fourth step, a pointer to the root of the binary tree is returned to the calling function.

4.4.6 Symbol Tables

The parser is also responsible for forming the symbol tables that hold information about the program variables and constants. Since each `for` loop block is handled individually, a separate symbol table for each block is formed. In the table, the Parser stores the name, type, and dimension information for each of the variables. Constant values are also stored in the symbol table. A status field and a current value field are also defined for each entity in the symbol table. The symbol tables are used by the Code Writer while matching modules and extracting vector operations.

4.4.7 Parsing Module Definition and Parameter Files

A special function, `ParseModuleLib()`, was implemented to parse the module definition file. This function uses the `Parser` class' `BuildExpressionTree()` function to parse the given module grammars. It also forms a linked list in which each entity holds a module definition. The function returns a pointer to the head of the list to the calling function. The calling function links the module library to the parse tree and sends them to the code writer.

Another special parser is implemented for the parameter file. This parser initializes the parameter class with the default values. Then, it reads the parameter values from the parameter file. Any parameter which is not defined with a value in the parameter file is left initialized with the default value.

4.5 Code Writer

Over 5000 lines of C++ source code were written to implement the code writer. Figure 4.10 shows the flow chart of the code writer. It includes nine steps. At each step, each `for` loop block of the user code is processed individually. It starts with the module matching step. At this step, it tries to find hardware modules for the arithmetic and conditional expressions (the conditional expressions of `if` statements) in the parse tree formed by the parser. In the second step, by running the loops in the parse tree, it extracts vector instructions for the expressions and conditional statements inside the loops. The third step performs the data dependency analysis between the vector instructions extracted from a `for` loop block. The next step schedules the vector instructions to the available PEs if more than one PE is available. Step five collects information about the indices of the array operands of the vector instructions and optimizes this information by eliminating unnecessary repetitions. Step six does the memory assignment, both for the vector instructions and for the operands of the vector instructions. The next step sorts the operands of the vector instructions for different hardware module implementations. In the last two steps, session files for each `for` loop block and new source code are written.

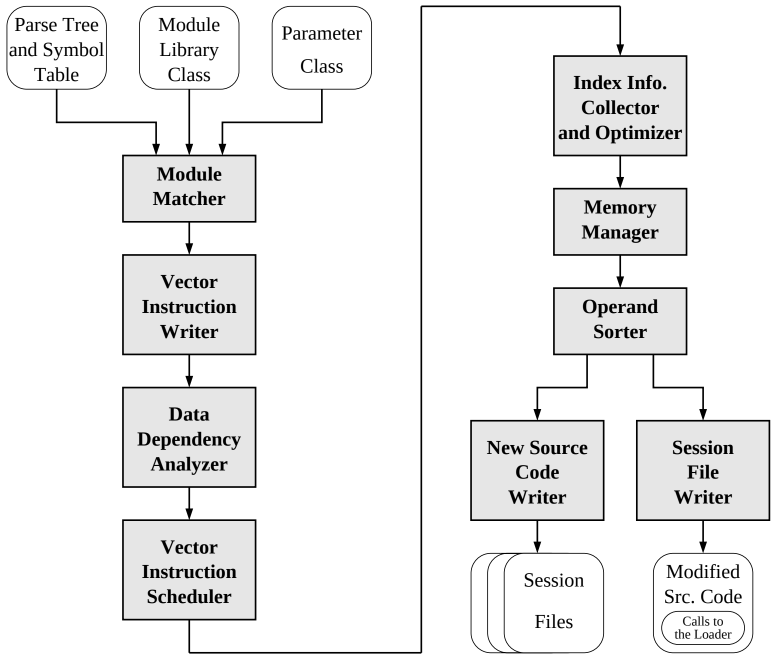


Figure 4.10: Code Writer flow chart.

In the rest of this section, the data types used by the code writer are introduced. Then, detailed information about each code writer step is provided.

4.5.1 Code Writer Data Types

Figure 4.11 shows the data type definitions for the code writer. Multiple instances of the data type `InstType` shown in Figure 4.11a are used to store one vector instruction with multiple operands. One instance of the `InstType` is used for single element variables and constant operands. An instance of `InstType` is used for each dimension of the array variable and these are linked together with the `*Same` field of this data type. The `*Next` field of the data type is used to link multiple operands together to form one vector instruction.

The Compiler extracts a set of vector instructions for most of the `for` loop blocks. `ILType` shown in Figure 4.11b is used to form a linked list of vector instructions extracted by the Compiler. The list is formed using the `*Next` field of the data type. This data type holds additional information regarding each vector instruction. For instance, a reference number for the hardware module matched for the instruction and the size of the vector instruction. Conditional expressions are also represented with this data type. The `*Sub` link field was defined to link the sub-instructions of a conditional vector instruction.

InstType

NodeEnum	NodeType
int	Value
int	FinalValue
int	Step
bool	Vaiable
char	*VName
int	IVal
float	FVal
int	MemLoc
InstType	*Same
InstType	*Next

(a)

ILType

STEnum	STType
int	STNo
int	MdNo
int	Var
int	Size
bool	Predicate
InstType	*Inst
ILType	*Sub
ILType	*Next

(b)

VarIdxType

NodeEnum	NodeType
char	*Vname
int	DimN
int	VarN
int	*Start
int	*Stop
int	MStart
int	MStop
VarIdxType	NextIndex
VarIdxType	NextVar

(c)

ILSetType

VarIdxType	*VInlist
VarIdxType	*VOutList
ILType	*IList
int	PE_Number
ILSetType	*Scheduled
ILSetType	*Next

(d)

ILBlock

ILSetType	*ILSet
SymbolTable	*TB
bool	DDep
ILBlock	*Next

(e)

Figure 4.11: Data type definitions for the Code Writer.

The Compiler collects information about the memory references performed by the vector instructions to determine which program variables and arrays (which sections of arrays) have to be transfered between the host computer and the RC system. `VarIdxType` was defined as shown in Figure 4.11c to store this information. This is also a multi-link type definition. In one direction, NextIndex, information regarding the same operand of a vector instruction is linked. In the other direction, NextVar, information regarding different operands of a vector instruction are linked together.

Some for loop blocks include more than one expression. The Compiler writes a group of vector instructions for each expression statement under the for loop block. ILSetType is defined as shown in Figure 4.11d to link these groups together and to store related information. This is also a dual linked data type. The *Next link is used to link the next group of vector instructions. The Compiler uses the *Scheduled link to store scheduled vector instructions.

The header type for the Instruction group is ILBlock, shown in Figure 4.11e. An instance of ILBlock is used for each for loop block, and they are linked together to form a linked list. The symbol tables formed by the parser for each block is also linked by this type definition. The DDEP field is used by the Data Dependency Analyzer. After the analysis of the block, this variable is set to true if there is a data dependency between the expressions in the given block.

4.5.2 Module Matcher

The purpose of the module matcher is to find appropriate hardware modules for the arithmetic and conditional expressions parsed by the parser. The matcher performs this task in three steps as shown in Figure 4.12. In the first step, it collects information about the vector operations performed by the for loop blocks. Basically, it runs the nested or non-nested for loop structures and looks for the array index changes in both arithmetic and conditional expression statements. This information is vital in the module matching procedure. The information collected in this step is stored in binary trees formed for the expression statements.

```
Match()
    Step 1 Collect information about the arithmetic and conditional expressions.
    Step 2 Try to find a hardware module for each expression.
    Step 3 Eliminate the for loop blocks which include
           one or more expressions without matching hardware modules.
```

Figure 4.12: Algorithm for `Match()` function.

In the second step, module matching is performed. Expressions are compared with the module definitions read from the module definition file. The Compiler compares an expression with all module definitions until a matching module is found for the expression. In the last step, `for` loop blocks containing one or more expressions that have not been matched with modules are eliminated. These `for` loops are executed on the host processor.

In order to have a successful module match, two conditions have to be satisfied. First, the binary trees of the expression and the module have to be identical. Second, array index information collected from the parsed loop block must match the index information provided in the module definition file for the candidate module. A function called `CmpExp()` was developed to compare binary trees of an expression and a module. The function starts from the roots of the trees and recursively spans the trees. While spanning the trees, it compares the structure of the trees as well as the array indexing information of the nodes.

Figure 4.13 shows the algorithm for the `CmpExp()` function. The algorithm performs a graph matching operation. Since the expression trees are acyclic graphs and we know the starting nodes (the root nodes), the algorithm compares trees in a single pass. Each recursive instance of the algorithm first checks the right and left subtrees, if they exist, it process them by calling itself recursively. Then, the current nodes are compared. The final result is calculated by performing a logical `AND` operation on the results of the right and left subtrees comparisons and the current node comparison.

```
CmpExp(Expression *ExA, Expression *ExB)
    bool RightResult, LeftResult, SelfResult
    //Compare right subtrees
    if (ExA has a right subtree)
        if (ExB has a right subtree)
            RightResult ← CmpExp(ExA↦Right, ExB↦Right);
        else
            RightResult ← false;
        end if
    else
        if (ExB has a right subtree)
            RightResult ← false;
        else
            RightResult ← true;
        end if
    end if
    //Compare left subtrees
    if (ExA has a left subtree)
        if (ExB has a left subtree)
            LeftResult ← CmpExp(ExA↦Left, ExB↦Left);
        else
            LeftResult ← false;
        end if
    else
        if (ExB has a left subtree)
            LeftResult ← false;
        else
            LeftResult ← true;
        end if
    end if
    //Compare current nodes
    if ((ExA↦Token == ExB↦Token) && (ExA↦NodeType == ExB↦NodeType))
        if ((ExA↦NodeType == VF) || (ExA↦NodeType == VI))
            SelfResult ← CmpIndex(ExA,ExB);
        else
            SelfResult ← false;
        end if
    end if
    return (RightResult && LeftResult && SelfResult);
```

Figure 4.13: Algorithm for the `CmpExp()` function.

4.5.3 Vector Instruction Writer

The purpose of the vector instruction writer step is to write an initial draft of the vector instructions of the `for` loop blocks whose expressions have been successfully matched in the previous step. Figure 4.14 shows the algorithm for the vector instruction writer. The `Write()` function processes all matched `for` loop blocks. First, it initializes the global variable *ILHeadGlobal* defined in the code writer class. The `Codewriter()` function recursively spans the given parse tree and runs the nested `for` loop structures in the tree. While running the loops in the tree, it extracts vector instructions and adds them to the list pointed to by the *ILHeadGlobal* variable.

```
Writer(ParseTree *$PTLoc$)
    while ($PTLoc$) do
        $ILHeadGlobal \leftarrow$ NULL;
        CodeWriter($PTLoc \mapsto ST$, $PTLoc \mapsto TB$,1);
        $ILHeadGlobal \leftarrow$ VarOperands($ILHeadGlobal$);
        $ILHeadGlobal \leftarrow$ SortInstList($ILHeadGlobal$);
        $\diamond$ Add $ILHeadGlobal$ to $ILBlockHead$
        $PTLoc \leftarrow PTLoc \mapsto NextTree$;
    end while
```

Figure 4.14: Algorithm for `Match()` function.

The third generation module requires the Compiler to identify the number of array operands in each vector operation. The `VarOperand` functions processes the list pointed to by the *ILHeadGlobal* variable, determines the number of array operands in each vector instruction, and stores this information in the same list.

In a nested `for` loop structure, expressions can be located at any nest level. Because of this arbitrary distribution of the expressions, the `CodeWriter()`

function writes vector instructions in a mixed order by giving them a reference number and links all vector instructions for all expressions in a block to one linked list. The `SortInstList()` function classifies these vector instructions and forms a linked list for each class. At the end, the function links these instruction lists together to form a list of vector instruction lists.

Figure 4.15 shows a sample user code fragment which includes three nested `for` loops and an expression statement at each nest level. We specified hardware modules in the module definition file for the expressions shown in the figure, and compiled the source code with the Compiler. Figure 4.16 shows the screen dump of the parse tree formed by the parser for the code fragment shown in Figure 4.15.

```
for (i=0; i<2; i++)
{
   aa[0] += bb[i];
   for (j=0; j<2; j++)
   {
      m[j] = b[j]+d[j];
      for (k=0;k<3; k++)
         e[k] = a[k]*b[k]*2;
   }
}
```

Figure 4.15: Sample user code fragment.

```
For Statement
        Expression Statement
        For Statement
                Expression Statement
                For Statement
                        Expression Statement
```

Figure 4.16: Parse tree formed by the parser for the code in Figure 4.15.

Figure 4.17 shows the screen dumps of the `CodeWriter()` function for the sample user code fragment shown in Figure 4.15. The first part of the figure shows the vector instructions after the `CodeWriter()` function returns. At this point, the vector instructions are in the order they were written and linked together in a single list with a reference number. The second part of the figure shows the instructions after they were classified into groups. The first group includes only one, the second group includes two, and the third group includes four instructions. These instruction groups are stored in a tree called the *Instructions Group Tree*.

```
FPVECMULMUL  [-842150451] 0 0 0 -1 Size = 3
FPVECMULMUL  [-842150451] 0 0 0 -1 Size = 3
FPVECADDS    [-842150451] 0 0 0 Size = 2
FPVECMULMUL  [-842150451] 0 0 0 -1 Size = 3
FPVECMULMUL  [-842150451] 0 0 0 -1 Size = 3
FPVECADDS    [-842150451] 0 0 0 Size = 2
FPVECACC     [-842150451] 0 0 Size = 2
```

```
FPVECACC     [1] 0 0 Size = 2

FPVECADDS    [2] 0 0 0 Size = 2
FPVECADDS    [2] 0 0 0 Size = 2

FPVECMULMUL  [2] 0 0 0 -1 Size = 3
FPVECMULMUL  [2] 0 0 0 -1 Size = 3
FPVECMULMUL  [2] 0 0 0 -1 Size = 3
FPVECMULMUL  [2] 0 0 0 -1 Size = 3
```

Figure 4.17: Vector instructions extracted from the source code in Figure 4.15.

In the nested `for` loop structures, the inner loops and the expressions inside these loops perform vector operations. The outer loops cause the repetition of the vector operation. In the inner most loop found in Figure 4.15, the k loop performs a single vector operation which we called `FPVECMULMUL`. The outer loop repeats this loop four times; therefore, the `CodeWriter()` function writes four vector instructions for the inner most loop. The same rule applies to the other expressions and the for loops in Figure 4.15.

4.5.4 Data Dependency Analyzer

In some cases, `for` loop blocks include more than one expression. When these kinds of blocks are mapped to the RC system, the Compiler writes a session file for each expression and puts all the vector instructions related to these expressions into separate session files. At run time, these session files are executed on the RC system individually in the order they are written. Because the session files are executed sequentially, data dependency between the expressions cause the RC system to calculate incorrect results. The Compiler performs a data dependency analysis on the `for` loop blocks to prevent the calculations of incorrect results on the RC system. The blocks with data dependencies are eliminated at the end of the analysis and the remaining blocks are sent to the next stage of the Compiler.

Figure 4.18 shows the algorithm for the data dependency analyzer. Since the individual `for` loop blocks in the original source code are executed sequentially both on the general purpose processor and on the RC system, no data dependency analysis is needed between these blocks; therefore, the algorithm analyzes each block independently.

The algorithm compares the output variables of each group of vector instruction with the input variables of the other groups of vector instructions in the same `for` loop block. A match indicates that there is data dependency between two instruction groups. In such a case, the algorithm marks the block and skips the next block immediately. After processing all blocks, the algorithm eliminates the blocks with data dependencies and sends the rest of them to the next step of the Compiler. The algorithm compares each instruction group with all other instruction groups in the same `for` loop block to detect the *loop-carried* data dependencies.

```
AnalyzeDataDepHigh()
    bool Result;
    ILBlock *ILBlockCur;
    ILSetType *ILSetCur,*ILSetTarget;
    InstType *InstCur;
    while (ILBlockCur) do
        if (The number of Instruction sets > 1) then
            Result ← false;
            ILSetCur ← ILBlockCur→ILSet;
            while (ILSetCur) do
                ◇ InstCur ← Output variable of ILSetCur
                ILSetTarget ← ILBlockCur→ILSet;
                while (ILSetTarget) do
                    if (ILSetCur != ILSetTarget) then
                        if (InstCur == one of ILSetTarget output variable) then
                            Result ← true;
                            break;
                        end if
                    end if
                    ILSetCur ← ILSetCur→Next;
                    if (Result) then break;
                end while
                ILSetCur ← ILSetCur→Next;
                if (Result) then break;
            end while
        end if
        ILBlockCur→DDep ← Result;
        ILBlockCur ← ILBlockCurILBlockCur→Next;
    end while
    ◇ Eliminate the blocks with data dependencies.
```

Figure 4.18: Algorithm for the data dependency analyzer.

4.5.5 Instruction Scheduler

The purpose of the instruction scheduler is to schedule one or more groups of vector instructions extracted from a `for` loop block to one or more PEs. The scheduling is important if there are more than one PE available. In such a case, the Compiler tries to exploit parallelism by distributing vector instructions to multiple PEs.

The scheduler performs the scheduling task according to the value of the `PE_NUMBER` parameter. If there is only one PE available, the scheduler schedules each group of instructions sequentially to the PE. In the single PE case, no modification is done to the instructions groups. Groups are executed sequentially on the PE. If there are more than one PE available, vector instructions in each group are evenly distributed to PEs for parallel execution. In this case, the scheduler divides each group of instructions into a number of subgroups and each subgroup is assigned to a PE.

4.5.6 Index Information Collector and Optimizer

A group of vector instructions extracted from a `for` loop block usually have one or more input variables and one output variable. These input and output variables are usually in the form of arrays. When we execute the vector instructions on PEs, all input variables have to be transfered from the user application to the PE memories and at the end of PE execution the results have to be transfered back to the user application.

Transferring data between the user application and the RC system is not an easy task. The Compiler has to know several things before transferring data in both directions. First of all, in some applications, some `for` loop blocks use only portions of the input array variables and produce results for a portion of the output array variable. We call this case the *Partial Use* of the variables. In this case, the Compiler needs to know which elements of the input array variables are used and which elements of the output variable are calculated by the vector instructions. With this knowledge, the Compiler writes the new source code so that only portions of the input and output variables are transfered between the user application and the PEs.

Secondly, in an instruction group, each vector instruction uses a portion of the input array variables and produces results for a portion of the output array variable. When the vector instructions are divided into subgroups and each group is assigned to a PE, correct segments of the input data must be transfered to the correct PE and after the PE execution completes, results collected from several PEs must be integrated correctly in the output array variable. We call this case *Data Scheduling*.

Thirdly, for some `for` loop blocks, the Compiler writes a group of vector instructions in which some vector instructions use the same portion of the same input array variable. This case is called *Data Sharing* for vector instructions. For example, in matrix multiplication, to calculate one column of the result matrix, one column of one input matrix is processed with all rows of the other input matrix. In this case, the input data segment that is shared by a number of vector instructions has to be transfered once and all vector instructions that use the piece of data have to be directed to the same memory location.

The Compiler collects information about indexing of the input and output array variables that are used by the vector instructions to handle these three cases: *Partial Use*, *Data Scheduling*, and *Data Sharing*. Figure 4.19 shows the algorithm for the function that performs the index information collection task.

```
BuildVarIdxList(ILBlokc *$ILBlockHead$)
    ILBlock *$ILBlockCur$;
    ILSetType *$ILSet$, *$ILSch$;
    ILType *$IL$;
    VarIdxType *$Vhead$;
    $ILBlockCur \leftarrow ILBlockHead$;
    while ($ILBlockCur$) do
        $ILSet \leftarrow ILBlockCur{\mapsto}ILSet$;
        while ($ILSet$) do
            $ILSch \leftarrow ILSet{\mapsto}Scheduled$;
            while ($ILSch$) do
                $IL \leftarrow ILSch{\mapsto}IList$;
                $VHead \leftarrow$ NULL;
                while ($IL$) do
                    $VHead \leftarrow$ BuildVarIdxOut($IL$, $IL{\mapsto}Inst$, $VHead$);
                end while
                $ILSch{\mapsto}VOutList \leftarrow$ VHead;
                $VHead \leftarrow$ NULL;
                while ($IL$) do
                    $VHead \leftarrow$ BuildVarIdxIn($IL$, $IL{\mapsto}Inst$, $VHead$, $ILSch{\mapsto}VOutList$. false);
                end while
                $ILSch{\mapsto}VInList \leftarrow$ VHead;
                $ILSch \leftarrow ILSch{\mapsto} Next$;
            end while
            $ILSet \leftarrow ILSet{\mapsto} Next$;
        end while
        $ILBlockCur \leftarrow ILBlockCur{\mapsto} Next$;
    end while
    $ILBlockCur \leftarrow ILBlockHead$;
    while ($ILBlockCur$) do
        $ILSet \leftarrow ILBlockCur{\mapsto}ILSet$;
        while ($ILSet$) do
            $ILSch \leftarrow ILSet{\mapsto}Scheduled$;
            while ($ILSch$) do
                $ILSch{\mapsto}VOutList \leftarrow$ ConcatVarIdxVariables($ILSch{\mapsto}VOutList$);
                $ILSch{\mapsto}VInList \leftarrow$ ConcatVarIdxVariables($ILSch{\mapsto}VInList$);
            end while
            $ILSet \leftarrow ILSet{\mapsto} Next$;
        end while
        $ILBlockCur \leftarrow ILBlockCur{\mapsto} Next$;
    end while
```

Figure 4.19: Algorithm for the `BuildVarIdxList()` function.

The `BuildVarIdxList()` function also processes each `for` loop block individually and independently. Each group of instructions written for a `for` loop block is also processed independently from the others due to the fact that groups are executed sequentially on PEs and a separate session file is written for each group. The `BuildVarIdxList()` function spans the *Instruction Group Tree* formed by the vector instruction writer and processes the instruction groups. It builds an input variable list and an output variable list for each group of instruction. Since, in each group, there is only one output variable, the output variable list is a one dimensional linked list. On the other hand, for the output variables, a two dimensional linked list is formed. In one dimension, information for different variables are linked together and in the other dimension indexing information of a single variable for all vector instructions are linked together. Note that variable index lists are also used to represent constants and single element variables with special settings.

In the accumulation expressions, the result variable also appears on the left side of the expressions as an input variable. While building the input variable list for this kind of expressions, each input variable is compared with the output variable of the expression. If there is a match, then this variable is excluded from the input list; otherwise, two separate memory spaces will be reserved for the same variable and this will cause confusion in the later steps of the Compiler.

The variable index list can include some entities that can be eliminated. For example, it can include some redundant entities due to problems found in the *Data Sharing* case. Some entities in the list can be concatenated in one entity. For example, in a list, one entity indicates that $A[3{:}7]$ (elements 3 through 7 of array variable A) are is used for one vector instruction and another entity indicates that $A[5{:}9]$ are used for another vector instruction. These two entities can be concatenated into one entity indicating $A[3{:}9]$. Concatenation reduces the amount of data communication between the host memory and PE memory. After the index lists are formed, the `BuildVarIdxList()` function spans the *Instruction Group Tree* one more time, eliminates the redundant entities and concatenates some entities if possible.

4.5.7 Memory Manager

The memory manager is responsible for assigning memory locations to both vector instructions and their operands. The memory assignment operation is completed in two steps. In the first step, memory locations are assigned to the variables listed in the variable index lists of the vector instructions and this information is stored in the variable index lists. In the second step, the memory information stored in variable index list is translated to the vector instructions and the second draft of the vector instructions is written.

In the first step, the memory assignment can be done in three different ways depending on the user's preference. The user specifies his preference by setting the `ASSIGNMEMORYMODE` parameter in the parameter file. In `SINGLEBLOCK` mode, continuous memory assignment is performed for all variables of all vector instructions in a single `for` loop block. This option is implemented for the third generation module. Since the third generation module can include several function cores, it can execute different types of vector instructions in one run. In the `SINGLEINST` mode, each group of vector instructions written for a `for` loop block is considered separately. In this mode, different groups can be assigned to different PEs for parallel execution but this does not guarantees the even distribution of the vector instructions to the PEs. Some instruction groups may include as few as one instruction and some may include as may as a thousand vector instructions. In the `PARALLEL` mode, since each subgroup of vector instructions scheduled is executed on a different PE-memory pair, memory assignment for each subgroup is performed separately.

While the hardware modules were designed, we reserved the first part of the PE memory starting at address `$00000` to store vector instructions. The second and third parts immediately following the first part are reserved for output variables and input variables respectively. For that reason, in each mode, the memory manager first calculates the amount of memory space needed to store vector instructions. Then, it performs the memory assignment to the output and input variables listed in the variable index list. Dual memory PEs are also considered during memory assignment.

In the second step, the memory manager spans the vector instruction group tree one more time and translates memory address information stored in variable index lists to the vector instructions. During the translation process, the manager takes a variable and searches the variable index list for that variable. When it finds the entity that covers the same indexing area with the variable, it calculates the final memory location for that variable and stores that information in the related field of the vector instruction.

4.5.8 Operand Sorter

The purpose of the operand sorter is to reorder the operands of the vector instructions to make them compatible with different module implementations. In the second draft of the vector instructions, operand order is first the <*Output Operand*>, then the <*Input Operands*> (in the order they appear in the original expression) and finally the <*Size*> of the vector instructions.

The operand sorter sorts the operands according to the value of the `OPERANDSEQMODE` parameter. If the parameter is set to `ORGMODE`, the sorter does not change the order of the operands. If the parameter is set to `REGMODE`, the sorter puts the operands into the following order. <*Output Operand*>, <*Input Operands*> (array variables first), <*Size*>. If the parameter is set to `NEWMODE`, operands are sorted as <*Input Operands*> (array variables first), <*Output Operand*>, <*Size*>. For the hardware modules that we developed `OPERANDSEQMODE` parameter has to be set to `NEWMODE`.

The operand sorter puts the vector instructions into the final form. After this step, vector instructions are ready to be written into session files.

4.5.9 New Source Code Writer

The purpose of the new source code writer is to rewrite the user source code by removing the sections that are successfully mapped to the RC system and replacing the removed sections by function calls to the Loader so that these sections can be executed on the RC system.

The source code writer uses the token list formed by the scanner. The source code writer starts writing the source code by including the Loader's header file, `loader.h`, into the user applications. Then, it declares a **boolean** variable called `RC_ModAva`. This variable is used by the Loader functions to determine the availability of the hardware modules and to activate the Simulator in the case that the required modules are not available. After that, it writes tokens until the first token of the first mapped `for` loop block. At this point, instead of writing the tokens of the mapped `for` loop block, it inserts the Loader functions. After all Loader functions are written, it skips the tokens until the end of the first mapped `for` loop block and keeps writing the source code from the token list. The source code writer repeats the same cycle for each successfully mapped section.

The new source code writer inserts four different functions for each modified section. First, it inserts `RC_IsModuleAvailable("<Module_Name>")` to check the availability of the FPGA board and hardware modules. This function returns **true** if the hardware module to execute current section on PEs is available. This result is passed to the other Loader functions to let them know that the module is available. If the module is not available, other Loader functions activate the Simulator. Next, the `Store_Data()` function calls are written to transfer data to the PEs. The `Run()` function is written to start the PEs and the `Load_Data()` functions are inserted to transfer the results back to the host processor memory. Detailed explanation of these functions can be found in Chapter 5.

The new source code writer also adds some comments, before and after each modified section. Figure 4.20 and 4.21 show a sample user program and the Compiler's output for this program, respectively.

```
#include <stdio.h>

void main()
{
   float aa[5],bb[5];
   float a[5],b[5],c,d[5],m[5],e[5];
   int i,j,k;

   for (i=0; i<2; i++) {
      aa[0] += bb[i];
      for (j=0; j<2; j++) {
         m[j] = b[j]+d[j];
         for (k=0;k<3; k++)
            e[k] = a[k]*b[k]*2
      }
   }
}
```

Figure 4.20: Sample user application.

```
#include "loader.h"

bool RC_ModAva;

#include <stdio.h>

void main()
{
   float aa[5],bb[5];
   float a[5],b[5],c,d[5],m[5],e[5];
   int i,j,k;

//***Begin Mapped Section***
//This section is written by software please do not edit.
//Section No: 1

   RC_ModAva = RC_IsModuleAvailable("FPVECACC");
   RC_StoreVector(1,&bb[0],0,0,0,0,0,1,1,1,1,0,1,0,2,5,6,RC_ModAva);
   RC_Run("dummy1.ses",RC_ModAva);
   RC_LoadVector(1,&aa[0],0,0,0,0,0,1,1,1,1,0,1,-1,1,4,4,RC_ModAva);

   RC_ModAva = RC_IsModuleAvailable("FPVECADDS");
   RC_StoreVector(1,&b[0],0,0,0,0,0,1,1,1,1,0,1,0,2,7,8,RC_ModAva);
   RC_StoreVector(1,&d[0],0,0,0,0,0,1,1,1,1,0,1,0,2,9,10,RC_ModAva);
   RC_StoreVector(2,&b[0],0,0,0,0,0,1,1,1,1,0,1,0,2,7,8,RC_ModAva);
   RC_StoreVector(2,&d[0],0,0,0,0,0,1,1,1,1,0,1,0,2,9,10,RC_ModAva);
   RC_Run("dummy2.ses",RC_ModAva);
   RC_LoadVector(1,&m[0],0,0,0,0,0,1,1,1,1,0,1,0,2,5,6,RC_ModAva);
   RC_LoadVector(2,&m[0],0,0,0,0,0,1,1,1,1,0,1,0,2,5,6,RC_ModAva);

   RC_ModAva = RC_IsModuleAvailable("FPVECMULMUL");
   RC_StoreVector(1,&a[0],0,0,0,0,0,1,1,1,1,0,1,0,3,9,11,RC_ModAva);
   RC_StoreVector(1,&b[0],0,0,0,0,0,1,1,1,1,0,1,0,3,12,14,RC_ModAva);
   RC_StoreVector(2,&a[0],0,0,0,0,0,1,1,1,1,0,1,0,3,9,11,RC_ModAva);
   RC_StoreVector(2,&b[0],0,0,0,0,0,1,1,1,1,0,1,0,3,12,14,RC_ModAva);
   RC_StoreVector(3,&a[0],0,0,0,0,0,1,1,1,1,0,1,0,3,9,11,RC_ModAva);
   RC_StoreVector(3,&b[0],0,0,0,0,0,1,1,1,1,0,1,0,3,12,14,RC_ModAva);
   RC_StoreVector(4,&a[0],0,0,0,0,0,1,1,1,1,0,1,0,3,9,11,RC_ModAva);
   RC_StoreVector(4,&b[0],0,0,0,0,0,1,1,1,1,0,1,0,3,12,14,RC_ModAva);
   RC_Run("dummy3.ses",RC_ModAva);
   RC_LoadVector(1,&e[0],0,0,0,0,0,1,1,1,1,0,1,0,3,6,8,RC_ModAva);
   RC_LoadVector(2,&e[0],0,0,0,0,0,1,1,1,1,0,1,0,3,6,8,RC_ModAva);
   RC_LoadVector(3,&e[0],0,0,0,0,0,1,1,1,1,0,1,0,3,6,8,RC_ModAva);
   RC_LoadVector(4,&e[0],0,0,0,0,0,1,1,1,1,0,1,0,3,6,8,RC_ModAva);

//***End Mapped Section***
}
```

Figure 4.21: Output of the RCCT Compiler for the program shown in Figure 4.20.

The new source code writer determines which variables and constants need to be transfered between the user application and the PEs by looking at the variable index lists. For each entity in these lists, it inserts a Load/Store Loader function. It also passes a number of parameters to the Load/Store functions. These parameters are: the PE Number, starting memory address of the array variable (up to 5 dimensions allowed), array dimension sizes, the starting index number of the segment that needs to be transfered, the transfer size, module availability information, and an address where the data is loaded from or stored to in PE memory.

4.5.10 Session File Writer

The purpose of the Session file writer is to write all vector instruction groups into text files, called session files, in a predetermined format. Details of the session file format is explained in Section 5.2. Similar to the previous Compiler steps, the session file writer spans the *instruction group tree* and writes one session file for each vector instruction group.

Since at compile time the user data is unknown, the Compiler can not insert the user data into session files; therefore, the user application has to transfer data to PEs before each session and transfer the results from PEs after the execution of a session. Due to this required data transfer between the user application and the PEs, it is not possible to put more than one session into each session file. Because, once the Loader starts executing a session file on PEs, it can not request data from the user application nor it can not send the results back to the user application between the execution of two sessions. This is due to the fact that the Loader is not able to initiate an action, like data transfer or start execution. All actions are started by the user application by calling the Loader functions.

When the sample user application shown in Figure 4.20 is compiled with the RCCT Compiler, the Compiler writes three session files one for each expression statement in the nested `for` loop block. Figure 4.22 shows the session file written by the Compiler for the second expression, $m[j] = b[j] + d[j]$.

```
dummy2 1
{
   SESSION 0 2
   {
      PE 1
      {
         Instructions 2 0x0
         {
            FPVECADDS 0x7 0x9 0x5 0x9
            HALT
         }
         Data 0
         {
         }
      }
      PE 2
      {
         Instructions 2 0x0
         {
            FPVECADDS 0x7 0x9 0x5 0x9
            HALT
         }
         Data 0
         {
         }
      }
      HOST
      {
         Instructions 4
         {
            LOADFILE 1 FPVECADDS.X86
            LOADFILE 2 FPVECADDS.X86
            SETCLOCK 50
            START ALL
         }
         DATA 0
         {
         }
      }
   }
}
```

Figure 4.22: A sample session file.

4.6 Handling RC Directives

There are a few cases that the Compiler may fail to map the user application to a desired hardware module. For example, a user may develop a single module that can execute multiple `for` loop blocks in the user application at once. In this case, to let the Compiler map these sections of the user application to the desired hardware module we defined several compiler directives.

By including these directives into the source code, the user can tell the Compiler that a certain section of the source code should be mapped to a certain hardware module. With directives, the user can also specify the program variables and constants that need to be transfered to the PEs and how the results can be collected.

Each directive has to be preceded by a "`#`" character and should start from the first column. The following is a list of user directives and their brief explanations.

- **RC_Start**: Marks the beginning of the desired section.
- **RC_End**: Marks the end of the desired section.
- **RC_Module**: Defines the module name and the module parameters.
- **RC_In_Vector**: Defines an array variable that needs to be transfered to the PE in which the module configuration is loaded.
- **RC_In_Single**: Defines a variable that needs to be transfered to the PE in which the module configuration is loaded.
- **RC_In_Const**: Defines a constant that needs to be transfered to the PE in which the module configuration is loaded.

- **RC_Out_Vector**: Defines an array variable or a single variable in which the results from PE is loaded.
- **RC_VectorSize**: Defines the size of vector operation that the module has to perform.

RC directives are handled at the very last step of the Compiler by the new source code writer. While writing the new source code from the token list, the source code writer monitors the tokens being written to output. When it sees an RC directive, it skips the tokens until it sees the directive that marks the end of directive section and calls the `MapRCDirective()` function. This function collects all necessary information from the directives to write a session file for the desired section and to replace the desired section with the function calls to the Loader. After all needed information is collected, it adds the require Loader functions to the new source code and writes a session file for the desired section. When the `MapRCDirective()` function returns, new source code writer continues to process other mapped sections.

Figure 4.23 shows a sample source code fragment that includes RC directives. In this code fragment, two `for` loop blocks are executed. The user has got a module called `DIR_EX_MODULE` that can calculate these two `for` loop blocks. When the code fragment was compiled with the RCCT Compiler, it generated the session file shown in Figure 4.24.

```
# RC_Start

# RC_Module DIR_EX_MODULE Dm Da Db Dc 23 Dd;
# RC_Out_Vector Dm 5 4 5;
# RC_In_Vector Da 5;
# RC_In_Vector Db 5;
# RC_In_Single Dc;
# RC_In_Const 23;
# RC_In_Vector Dd 5;
# RC_VectorSize 5;

for (i=0; i<5; i++)
{
   Da[0] += Db[i];
   for (j=0; j<5; j++)
   {
      Dm[j] = Db[j]+Dd[j];
      for (k=0;k<3; k++)
         Db[k] = Da[k]*Db[k]*2;
   }
}

for (fff=0;fff<2;fff++)
   for (jj=0;jj<5;jj++)
      for (i=0;i<5;i++)
         Dm[fff] += Da[i] + (Db[jj] + Dc * 23.0) * Dd[i];

# RC_End
```

Figure 4.23: Sample user code fragment with RC directives.

```
dummyD1 1
{
   SESSION 0 1
   {
      PE 1
      {
         Instructions 2 0x0
         {
            DIR_EX_MODULE 0xe 0x5 0x4 0x5 0x72 0x5 0x77 0x5 0x7c 0x7d 0x7e
                          0x5 0x5
            HALT
         }
         Data 0
         {
         }
      }
      HOST
      {
         Instructions 3
         {
            LOADFILE 1 DIR_EX_MODULE.X86
            SETCLOCK 50
            START ALL
         }
         DATA 0
         {
         }
      }
   }
}
```

Figure 4.24: Session file written by the RCCT Compiler for the code fragment shown in Figure 4.23.

Chapter 5

The RCCT Loader, Simulator and the Session File Format

The Loader/Simulator pair is another major component of the Reconfigurable Computing Compilation Tool (RCCT). It is analogous to the assembler and loader found in traditional General Purpose Processor (GPP) compilation systems. The Loader is a software program that works as an interface between the user applications and reconfigurable devices. The main purpose of this interface program is to synchronize the host computer with the FPGA device. The application utilizes reconfigurable resources by calling the functions available in the Loader. The Simulator was developed as an integral part of the Loader to simulate new hardware module designs and new RC systems. A session file is a text file that includes vector instructions, vector data and instructions for the Loader. The Loader instructions in a session file are executed by the Loader on the GPP and the vector instructions are executed by the hardware modules on the FPGA devices [81, 82]. In this chapter, the session file format is introduced after a brief overview of the Loader/Simulator pair. Subsequently, detailed information about two versions of the Loader and the Simulator is provided.

5.1 An Overview of the RCCT Loader and Simulator

Initially, we developed the Loader as an interface for the hardware modules explained in Chapter 3. We made the Loader so generalized that it can be used not only with our hardware modules but also with other reconfigurable applications. We call this version of the Loader the stand-alone version. Later, with some added functions, the Dynamic Link Library (DLL) version of the Loader was developed. This version was especially developed for the applications compiled by the RCCT Compiler. Finally, we added the Simulator to the Loader as an integral part for simulating new module designs and new RC systems before they are implemented.

Each version of the Loader accepts session files as input. They take the given session file and execute some portions of it on the host computer and some parts on FPGA devices. It accesses the reconfigurable device by calling FPGA device vendor's Application Programming Interface (API) functions. It is able to perform complex tasks, invoked in the given session file as host instructions, by calling several API functions. When instructed in the session file, it can produce a results file that includes the results collected from the reconfigurable devices after the execution of the session file on the device. The Simulator was added to the DLL version of the Loader. The DLL version of the Loader activates the Simulator to simulate a new module design or a new RC system when needed.

The DLL version of the Loader and the Simulator pair is used in the execution phase of the mapping process as illustrated in Figure 5.1. In this phase, sections of the application executable code that do not require high performance run on the general purpose processor. When the execution reaches the sections produced by the RCCT Compiler, the application utilizes the RC system via several function calls to the Loader. The Loader includes four types of functions, that can be called by the application. These are: `IsModuleAvailable`, `Store_Data`, `Load_Data` and `Run` functions. The `Store_Data` and `Load_Data` functions move data between FPGA devices and the host processor. By calling the `Run` function with a session file name as a parameter, it instructs the Loader to execute a given section on FPGA devices with given hardware modules.

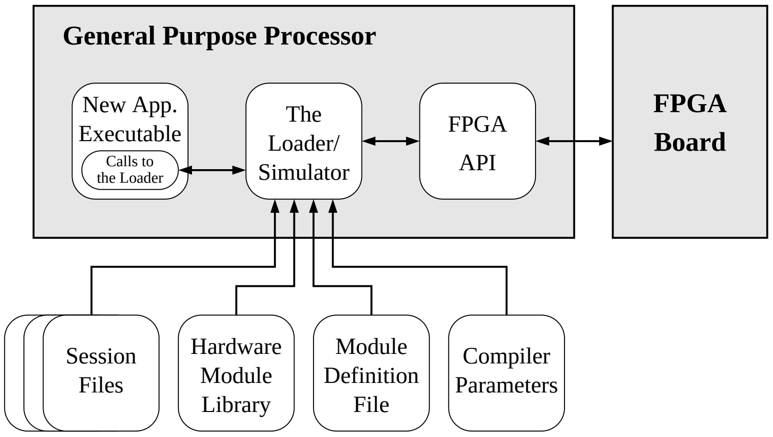


Figure 5.1: RCCT Execution phase.

Each Loader function that can be called by the user application accesses the RC system using API functions. If a module configuration file or the RC system is not available, these functions automatically activate the Simulator instead of calling the API functions. The Simulator provides several functions, which are associated with the API functions, to the Loader. The purpose of the Simulator is to calculate an estimated execution time for the mapped sections of the user applications if the configuration file for the module or the RC system is not available to run the user applications.

5.2 Session File

A session file is a text file that contains RC system specific commands that are executed on the General Purpose Processor (GPP) or on the FPGA devices [81, 82]. It is highly structured and includes instructions and data for both the Loader and the hardware modules. Figure 5.2 shows the general structure of the session file. The file is divided into nested regions. Each region includes a single line header and a body. The beginning and ending of a region's body are delimited by "{" and "}". The session file begins with the name of the file and a number that indicates the total number of sessions in the file. The main regions in the file are called *sessions* because all instructions and data in a *session* are processed at once by utilizing one or more Processing Elements (PEs). The header of a session region includes the word `SESSION`, session ID number, and the number of PEs that are going to be utilized in this particular session. Inside a session, a PE region is inserted for each PE.

A PE region always includes two sub-regions, instructions and data. An instruction sub-region begins with the word `INSTRUCTION` and a number that indicates the number of instructions inserted in that region. Instructions under this region are the ones that are executed by the modules. The final instruction in this sub-region should always be `HALT` to direct the `PE` to stop execution. The other sub-region of the PE region is the `DATA` sub-region. For flexibility, the data region is divided into further sub-regions called blocks. Each block begins with the word `BLOCK` and an address indicating where the data inside the block will be stored in the PE memory.

Inside a session region immediately following the `PE` regions, a `HOST` sub-region is inserted. Each session region must include only one host region. Similar to the PE region, a host region includes instructions and data sub-regions. The only difference is that instructions and data in these regions are processed by the Loader on a GPP.

```
<Session File Name> <Session Sount>
{
    SESSION <Session ID> <PE Count>
    {
        PE <PE ID>
        {
            INSTRUCTIONS <Instruction Count> <Address>
            {
                <Instruction_1>
                <Instruction_2>
                        :
                        :
                <Instruction_n>
                HALT
            }   # End of Instructions
            DATA <Block Count>
            {
                BLOCK <Size> <Address>
                {
                    <Data_1>
                    <Data_2>


                    <Data_n>
                }   # End of the Block
                :
                :   # More Data Blocks
                :
            }   # End of Data
        }   # End of a PE
        :
        :   # More PEs
        :
        HOST
        {
            INSTRUCTIONS <Instruction Count>
            {
                <Host Instruction_1>
                <Host Instruction_2>
                :
                <Host Instruction_m>
            }
            DATA <Block Count>
            {
                # Similar to the PE DATA Region
            }
        }   # End of Host
    }   # End of the Session
    :
    :   # More Sessions
    :
}   # End of the Session File
```

Figure 5.2: Session file format.

As was mentioned above, the session file is highly structured. The structure of the session file makes it easy to parse and store in data structures. The structure also helps in determining the schedule of the jobs for PEs. All jobs that can be executed in parallel can be included in one session depending on the availability of the PEs. If there is a dependency between Job_t and Job_{t+1} (Job_{t+1} depends on the result of Job_t) then Job_{t+1} must be scheduled in a session that comes after the session that includes Job_t. The session file format is also flexible. One can schedule as many sessions as needed in a session file. Further more, in a session, as many PEs as available can be utilized. All instruction and data regions are flexible. Data regions could be empty in cases that data is transfered directly from the user application to the PE memories.

Single line comments are allowed in the session file. They are especially useful for debugging the session file if the user is writing it by himself. Each comment line has to start with "#" symbol and must not overflow to the next line.

5.3 Stand-Alone Version of the Loader

The stand-alone version of the Loader is an interface program for RC systems that can be used not only with our hardware modules, but also with other reconfigurable applications. With current RC systems like the one available in our laboratory, RC applications developers typically need to write an interface program, to be able to utilize their hardware implementation on the RC system. The main purpose of this interface program is to synchronize the host computer with the FPGA device. Usually, an interface is responsible for initializing the FPGA devices by loading hardware configurations, starting and stopping the hardware implementation, and managing data movement between the host computer and the FPGA devices. Since each application is different, in terms of data inputs, outputs, and the method of processing data, designers are typically required to implement a specific interface program for each specific application.

Hence, we developed the stand-alone Loader program as a generalized form of the interface program. Rather than writing a new interface program and recompiling it for each application, the user simply writes a new session file (text-file). This removes the need for recompilation of the interface program. The stand-alone Loader program includes most of the basic functions that an interface can perform. For each interface function such as initialization of the FPGA devices or data movement, a loader instruction is defined and included in the session file. The Loader reads the session file and manipulates the RC system according to the instructions inserted in the session files.

The stand-alone version of the Loader was initially developed for testing the hardware modules. After new hardware modules are implemented, the developer can easily utilize them by writing new session files for the new modules and by running the Loader with the new session files.

As show in Figure 5.3, this version of the Loader includes two major parts. These are the session file scanner/parser, printer, and the Executor. In the following sub-sections, first the data type definitions are introduced. After the type definitions, two parts of the stand-alone Loader are introduced.

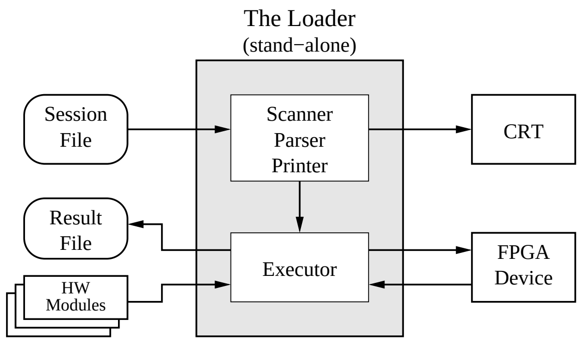


Figure 5.3: Flow chart of the Loader (stand-alone version).

5.3.1 Data Type Definitions to Store Parsed Session Files

Figure 5.4 shows the data structure defined in the `loader.h` file to store a parsed session file. For each region of the session file, a specific data type was defined. To be able to store structural features of the session file format, these data types are linked together. To make type definitions flexible, pointers were used as needed. Instead of using linked lists in the type definitions, we used the pointers as array pointers. The size of a region or the number of sub-regions included in that region is specified in the header of that region. This information helps the Loader in forming necessary arrays in the desired size before parsing the given region. In Figure 5.4, vertical dots are used to indicate variable size arrays.

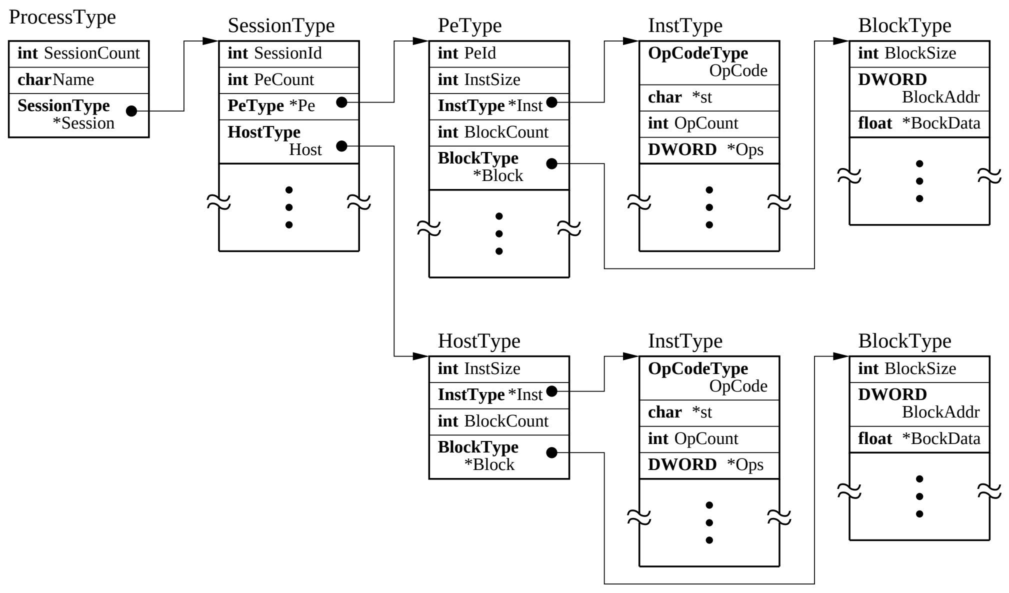


Figure 5.4: Data type definitions for the Loader.

5.3.2 Scanning, Parsing and Printing the Session File

The vocabulary of the session file only includes a few reserved words, two types of numbers, real and integer, and some delimiters, "{" and "}". No expressions or complex grammars are used in the session file format definition. With the help of the information included in the region headers, it is always possible for the parser to predict the next token or type of the next token in the session file while parsing it; therefore, a simple scanner and a parser were coded to scan and parse session files.

When the Loader is run, it first reads the session file in a character buffer and then calls the parser. Every time the parser needs to read a token from the character buffer, it calls the scanner function, `ReadNext()`. When it is called, the `ReadNext()` functions identifies the next token in the character buffer and returns it as a string to the parser. The parser takes this string and converts it to an appropriate type such as identifier, integer or real.

The information provided in each region's header and the structure of the session file makes the parser's job easier. By looking at the headers, the parser determines what it needs to parse next. One specific parser function was implemented to parse each region of the session file. Algorithmically, these functions are very similar. Figure 5.5 shows the algorithms for these functions. Each function first parses the header of the region, and then it reserves enough memory space to parse the body of the region. Finally, by calling other parser functions, the body of the region is parsed. Taking this approach, significantly reduced the programming effort was needed to implement a special parser for session files.

The parser has several features that are useful for debugging. It checks for reserved words and the boundaries of each region. When an error is detected, the parser sends an error message along with the location of the error to the user. A print function has been implemented for debugging purposes also. When printing is enabled, the Loader prints the session file after parsing is completed. These features are useful, especially when the session file is created manually.

```
ReadProcess()
    $P \leftarrow$ new ProcessType;      //A global variable defined in intprt class
    $P.Name \leftarrow$ ReadNext(“ ”);
    $P.SessionCount \leftarrow$ atoi(ReadNext(“ ”));
    $P.Session \leftarrow$ new SessionType[$P.SessionCount$];
    for $SesNum \leftarrow 0$ to $P.SessionCount$ do
        ReadSession($SesNum$);
    end for

ReadHost(int $SesNum$)
    $P.Session[SesNum].SessionId \leftarrow$ atoi(ReadNext(“ ”));
    $P.Session[SesNum].PeCount \leftarrow$ atoi(ReadNext(“ ”));
    $P.Session[SesNum].Pe \leftarrow$ new PeType[$P.Session[SesNum].PeCount$];
    for $PeNum \leftarrow 0$ to $P.Session[SesNum].PeCount$ do
        ReadPe($SesNum,PeNum$);
    end for
    ReadHost(SeNum);

ReadPe(int $SesNum$, int $PeNum$)
    $P.Session[SesNum].Pe[PeNum].PeId \leftarrow$ atoi(ReadNext(“ ”));
    $P.Session[SesNum].Pe[PeNum].InstSize \leftarrow$ atoi(ReadNext(“ ”));
    $P.Session[SesNum].Pe[PeNum].Inst \leftarrow$ ReadOp($P.Session[SesNum].Pe[PeNum].InstSize$);
    $P.Session[SesNum].Pe[PeNum].BlockCount \leftarrow$ atoi(ReadNext(“ ”));
    $P.Session[SesNum].Pe[PeNum].Block \leftarrow$
        new BlockType[$P.Session[SesNum].Pe[PeNum].BlockCount$];
    for $i \leftarrow 0$ to $P.Session[SesNum].Pe[PeNum].BlockCount$ do
        $P.Session[SesNum].Pe[PeNum].Block[i].BlockSize \leftarrow$ atoi(ReadNext(“ ”));
        $P.Session[SesNum].Pe[PeNum].Block[i].BlockAddress \leftarrow$ atoi(ReadNext(“ ”));
        $P.Session[SesNum].Pe[PeNum].Block[i].BlockData \leftarrow$
            new float[$P.Session[SesNum].Pe[PeNum].Block[i].BlockSize$];
        for $k \leftarrow 0$ to $P.Session[SesNum].Pe[PeNum].Block[i].BlockSize$ do
            $P.Session[SesNum].Pe[PeNum].Block[i].BlockData[k] \leftarrow$ atoi(ReadNext(“ ”));
        end for
    end for
```

Figure 5.5: Algorithms for parser functions.

```
ReadHost(int SesNum)
    P.Session[SesNum].Host.InstSize ← atoi(ReadNext(“ ”));
    P.Session[SesNum].Host.Inst ← ReadOp(P.Session[SesNum].Host.InstSize);
    P.Session[SesNum].Host.BlockCount ← atoi(ReadNext(“ ”));
    P.Session[SesNum].Host.Block ←
        new BlockType[P.Session[SesNum].Host.BlockCount];
    for i ← 0 to P.Session[SesNum].Host.BlockCount do
        P.Session[SesNum].Host.Block[i].BlockSize ← atoi(ReadNext(“ ”));
        P.Session[SesNum].Host.Block[i].BlockAddress ← atoi(ReadNext(“ ”));
        P.Session[SesNum].Host.Block[i].BlockData ←
            new float[P.Session[SesNum].Host.Block[i].BlockSize];
        for k ← 0 to P.Session[SesNum].Host.Block[i].BlockSize do
            P.Session[SesNum].Host.Block[i].BlockData[k] ← atoi(ReadNext(“ ”));
        end for
    end for

ReadOp(int InstSize)
    Inst ← new InstType[InstSize];
    for OpNum ← 0 to InstSize do
        Inst[OpNum].Opcode ← ReadNext(“ ”);
        If Inst[OpNum].OpCode is an Host instruction then
            ◇ Read parameters for the specific Host instruction to Inst[OpNum].Opx
        else
            ◇ Save Current buffer position
            ◇ Count the number of parameters and store in Cnt.
            Inst[OpNum].Ops ← new DWORD[Cnt]
            ◇ Restore saved buffer position
            for i ← 0 to Cnt do
                Inst[OpNum].Op[i] = atoh(ReadNext(“ ”));
            end for
        end if
    end for
```

Figure 5.5: Algorithms for the parser functions (continues).

5.3.3 Running the Parsed Session File

The most important part of the Loader is the executor. This part is the one that interacts with the RC system. It runs the parsed session file on the RC system. It also manages data movements between the FPGA devices and the software portions of the applications.

Figure 5.6 shows the algorithms for the executor functions of the Loader. The `RunProcess()` function goes through parsed sessions and executes them in sequence by calling the `RunSession()` function. The sessions are executed in the order they are placed in the session file. The `RunSession()` function first initializes the local PE memories by scanning all parsed PE regions under the current session region. It moves all module instructions and module data located under these parsed PE regions to local memories of their associated actual PEs. After that, it calls the `RunHost()` function to execute the host instructions located under the parsed host region of the current session region.

The `RunHost()` function assumes that at least one `START` instruction is located among the host instructions and some `LOADFILE` instructions are used before the `START` instruction. The `LOADFILE` instructions tells the Loader to configure PEs with module configuration files. The Loader first executes all host instructions until the `START` instructions. When it sees a `START` instruction, it fires all hardware modules and lets modules execute their specific module instructions on their local data previously moved to their memory by the `RunSession()` function. When all modules are fired, the `RunHost()` functions enters a wait state and waits for the interrupt signals from the hardware modules. It waits until all modules assert interrupt signals. When a module asserts an interrupt signal, it declares that it has finished execution of its specific module instructions. When all hardware modules assert interrupt signals, the `RunHost()` functions exits from the wait state and executes the rest of the host instructions under the host region.

```
RunProcess()
    ◇ Open connection with the available RC resource.
    for $SesNum \leftarrow 0$ to $P.SessionCount$ do
        RunSession($SesNum$);
    end for
    ◇ Close connection with the available RC resource.

RunSession(bf int $SesNum$)
    for $PeNum \leftarrow 0$ to $P.Session[SesNum].PeCount$ do
        ◇ Initialize PEs
            • Move Module data to Local PE Memory.
            • Move Module Instructions to Local PE Memory.
    end for
    RunHost(SeNum);

RunHost(bf int $SesNum$)
    ◇ Execute all instructions in $P.Session[SesNum].Host.Inst$ until START instruction.
    ◇ Store current time information.
    ◇ Start all PEs
    ◇ Wait until all PEs send interrupt signals
    ◇ Calculate timing using current time and stored timing information.
    ◇ Execute the rest of the instructions in $P.Session[SesNum].Host.Inst$.
```

Figure 5.6: Algorithms for executor functions.

While entering and exiting the wait state, the `RunHost()` functions records the current time and calculates the execution time of each session. To calculate timing information precisely, the functions reads general purpose processor's (in our case Pentium X) *Time-Stamp-Counter* every time it needs to record current time. This counter is incremented by one at every clock pulse applied to the processor. The execution time of a session is calculated using Equation 5.1, where T_S is session execution time, t_0 and t_1 are current values of the *Time-Stamp-Counter* read while entering and exiting wait state respectively and C_{CPU} is processor's clock rate.

$$T_S = \frac{t_1 - t_0}{C_{CPU}}\ sec \tag{5.1}$$

The `RunHost()` functions executes host instructions by calling `RunOp()` function. Table 5.1 shows the `RunOp()` function's instruction set and the meaning of each instruction. This table is also the instruction set of the Loader. The `RunOp()` function executes host instructions by calling one or more API functions of the FPGA device. Being able to execute host instructions isolates the users from complex API functions and offers a user friendly environment. Currently, the instructions set is limited, but new instructions can be added by simply modifying the `RunOp()` function as required.

Table 5.1: The Loader instruction set.

Instruction		**Meaning**
`LOADFILE`	`ID, FILE`	Loads a configuration bit stream to a PE. `ID`: PE ID number, `FILE`: Configuration file name
`SETCLOCK`	`FR`	Sets the clock frequency of the RC system. `FR`: Frequency
`START`	`ID`	Starts one or all PE by releasing reset signal. `ID`: PE ID Number, `ID = ALL`=> Start all available PEs

`PRINTMEMBLOCK`	`ID,ADDR,`
	`SIZE,TYPE`
		Prints a block of data from a PE's memory in Integer, Float or Hex format.
		`ID`: PE ID number, `ADDR` : PE Memory Address,
		`SIZE`: Size of the data block, `TYPE`: Printing type

`MOVEMEMBLOCK`	`ID1,ADDR1,`
	`ID2,ADDR2,`
	`SIZE`
		Moves a block of data from one PE's memory to another PE's memory
		`ID1`: Src. PE ID number, `ADDR1` : Src. address
		`ID2`: Dest. PE ID number, `ADDR2` : Dest. address

5.4 DLL Version of the Loader

The easiest and the most efficient way to incorporate the Loader into the user applications compiled by the RCCT Compiler is to convert the Loader to a Dynamic Link Library (DLL) and call it from the applications during runtime. For that reason, the DLL version of the Loader was implemented by adding some exportable functions to it. When a user application is compiled with the RCCT Compiler, the Compiler modifies the source code and produces a new application source code by replacing computationally complex sections of the application with exportable Loader functions. This new source code is then recompiled with the original programming language compiler to get the application executable code.

The DLL version of the Loader was built on top of the Stand-alone version. This means that the DLL version of the Loader includes the stand-alone version plus some exportable functions that can be called by the user applications compiled by the RCCT Compiler. Four types of exportable functions were added. These are `IsModuleAvailable()`, `Store`_*Data*`()`, `Load`_*Data*`()` and `Run()` functions.

With the `IsModuleAvailable()` function, a user application can check the availability of hardware modules. If a requested hardware module is available, the `IsModuleAvailable()` function returns `TRUE`. The user application forwards the result of `IsModuleAvailable()` to the other DLL functions. This is necessary for informing the Loader about the availability of the module. If the hardware module is not available, the Loader activates the Simulator instead of utilizing the FPGA device.

User applications can utilize the `Store_`*`Data`*`()` and `Load_`*`Data`*`()` functions to move data between FPGA devices and the host computer. There are four `Store_`*`Data()`* functions available. These are: `StoreSingle()`, `StoreConst()`, `StoreWholeVector()` and `StoreVector()`. `StoreSingle()` and `StoreConst()`. These functions can be used to store the content of a single variable and a constant value to a specific memory address location of a specific FPGA device. User applications can copy whole arrays (up to 5 dimensions) to a specific FPGA device's local memory, starting from a specific memory location with the `StoreWholeVector()` function. The `StoreVector()` function is the most sophisticated store function among the others. With this function, applications can select one piece of one dimension of an array (up to 5 dimensional) and copy it to a specific FPGA device's local memory, starting from a specific memory location. For every `Store_`*`Data`*`()` function, excluding `StoreConst()`, a matching `Load_`*`Data`*`()` function that does the reverse operation is also implemented.

Figure 5.7 shows data flow while a user application accesses reconfigurable resources through the DLL version of the Loader. It first checks the availability of the hardware module and forwards the result to the other functions that it calls from the Loader. After checking the availability of the required hardware module, the application initializes FPGA devices by storing data to their local memories using `Store_`*`Data()`* functions. When all data that needs to be processed is transfered to FPGA memories, the application calls the `Run()` function to start hardware modules. It calls the `Run()` function with the name of a session file generated by the RCCT Compiler when the application is compiled. The `Run()` function parses and executes the given session file as was explained in the previous section. After the module execution is completed, the `Run()` function exits. At this point, the application collects the results from FPGA memories by utilizing

`Load_`*`Data`*`()` functions. In some cases, the RCCT Compiler modifies more than one section of the user applications. In such cases, the execution cycle is repeated for each modified section of the user application.

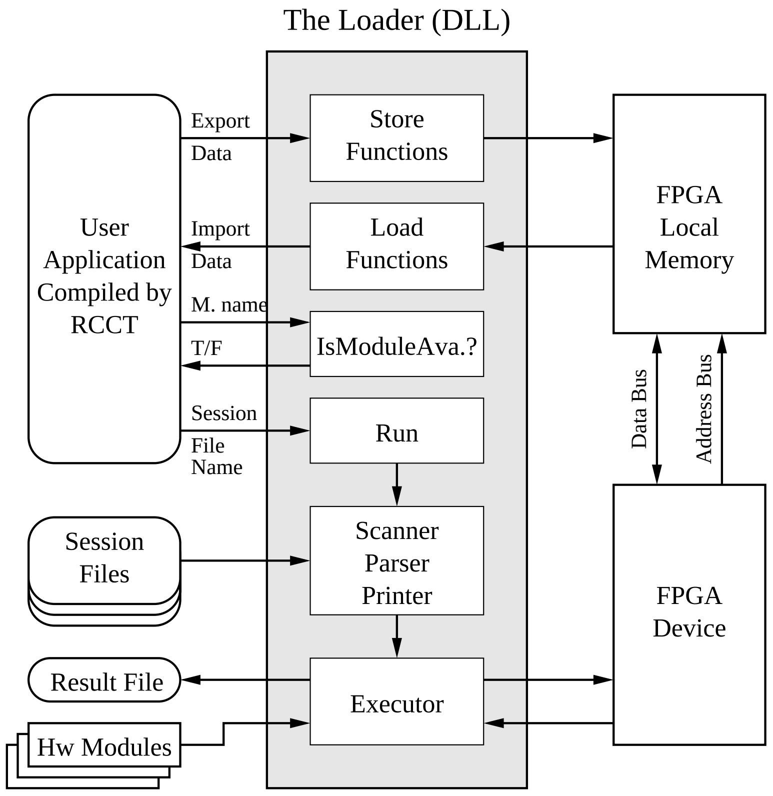


Figure 5.7: Flow chart of the Loader (DLL version).

5.5 The Simulator

We developed an RC resource Simulator by adding several functions to the DLL version of the Loader. We chose to combine the Simulator with the Loader because the Loader had a lot of features that are also needed for the Simulator. The Simulator is able simulate modules and alter the content of the FPGA memories. When module specifications are provided, the Simulator is able to simulate any module that performs vector operations.

There are several reasons behind developing a simulator for the modules. First, designing and implementing a new module could take a significant amount of time. With the Simulator, users can simulate their module definitions and make necessary adjustments on their designs before they implement them. As a result, design and implementation time of the new modules can be reduced significantly. Secondly, when an appropriate hardware module is not available to execute a modified section of the user application, the Loader activates the Simulator. The Simulator executes that section of the application on the host computer as if it were being executed on the FPGA chip and it returns an estimated execution time along with the results. By examining the estimated execution time, users can determine if a new hardware module can reduce execution time of an application before implementing the module. Third, by adjusting the RCCT Compiler parameters, such as the number of PEs available or the maximum clock rate of the modules, users can simulate their applications on virtually defined reconfigurable environments. This is especially useful when users need to make a decision on purchasing new reconfigurable hardware for their applications. Users can see how much performance gain they will get when they purchase a specific reconfigurable system.

As was mentioned, the Simulator was built on top of the DLL version of the Loader. Figure 5.8 shows how user applications can utilize the Simulator. Beside the session file and input data provided by user applications, the Simulator requires some additional inputs to be able to simulate compiled user applications. One of the additional inputs that the Simulator needs is the `Module.lib` file. The user should carefully specify his modules in the `module.lib` file. The Simulator reads module specifications from this file and uses them as templates while executing module instructions. The Simulator also needs to know values of the compiler parameters used while compiling the user application. The Simulator needs this information in order to understand how the RCCT Compiler compiled the user application. For example, the Simulator collects the number of PEs available from this file.

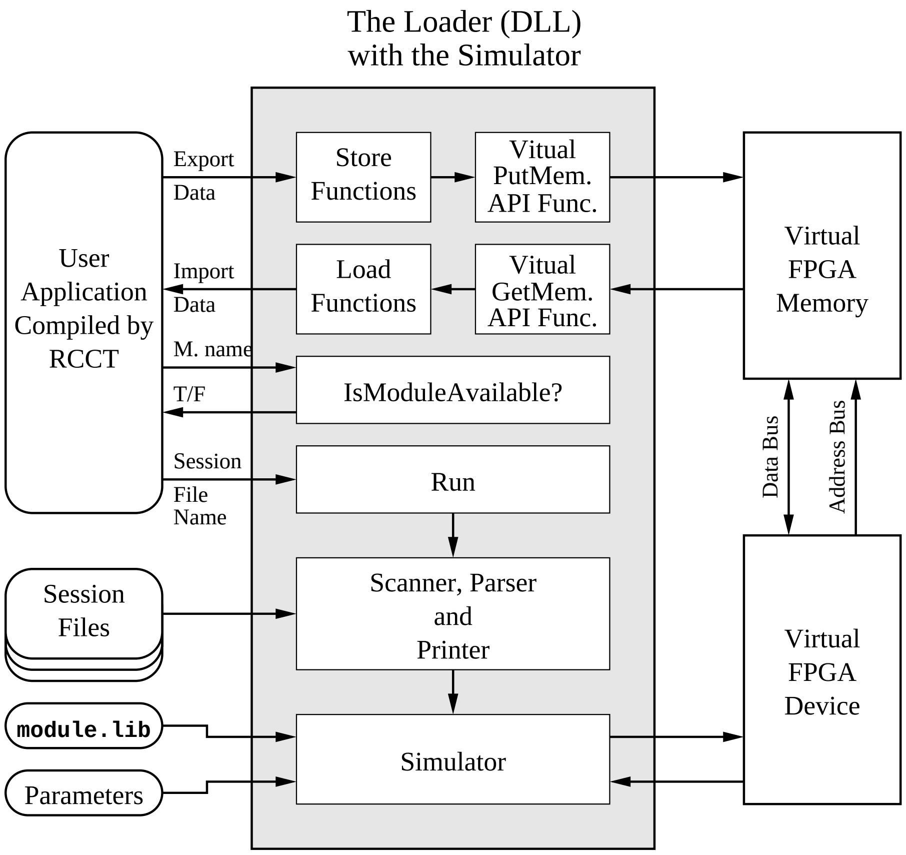


Figure 5.8: Flow chart of the Loader (DLL version) with the Simulator.

As seen in Figure 5.8, when a user application is started, it first checks the availability of the required hardware module. When the requested hardware module is not available, the `IsModuleAvailable()` function returns `FALSE`. After that, the user application calls `Store_`*`Data()`* functions with the result of `IsModuleAvailable()` function. This situation causes `Store_`*`Data()`* functions to activate the Simulator. Instead of accessing the real reconfigurable system, these functions write data to virtual memory by calling the Simulator's virtual API functions. The `Load_`*`Data()`* behaves the same way. When the user application needs to read results from the RC system, they access the virtual memory.

After initializing the Simulator memory, the user application calls the `Run()` function with a session file name as a parameter to this function. The `Run()` function first gets the session file parsed by calling the Loader's scanner and parser. After the session file is parsed, instead of calling the Executor, the `Run()` function calls the Simulator with the parsed session file. The Simulator goes through the given session file and simulates it using virtual module definitions by accessing previously initialized virtual memory.

The most important part of the Simulator is the module simulator. Figure 5.9 shows the algorithm for this part of the Simulator that executes module instructions on virtual module definitions. The Simulator is able to simulate both regular modules and the conditional modules. The algorithm first reads necessary parameter values used during the compilation of the user applications to understand how the application is compiled. After that, the Simulator reads the module definitions from the `module.lib` file. It reads two things for each module from this file. These are grammar and timing information of the modules used in the session currently being simulated. Module grammars specified in the `module.lib` file are actually mathematical expressions defining vector operations that the module performs on the given vectors, variables and constant values. The timing information is used to estimate the execution time of the module using equations provided in Chapter 3.

```
SimulateModule(int SesNum, int PeNum)
    ◇ Read compiler parameters from par.txt file.
    ◇ Find specification for the given virtual module from module.lib
    ◇ Construct operand map for the module
    ◇ EXECUTE ALL MODULE INSTRUCTIONS
        for $i \leftarrow 0$ to Process.Session[SesNum].Pe[PeNum].InstCount do
            ◇ Calculate estimated execution time of the instruction.
            ◇ Initialize addresses on the operand map of the module
            ◇ EXECUTE ONE MODULE INSTRUCTION
                for $k \leftarrow 0$ to VectorSize do
                    • Grab data from memory and put it into the temporary data array.
                    • Put the data into virtual in correct module.
                    • Execute the virtual module for one iteration
                        of the vector instruction.
                    • Put the result back to the memory.
                    • Update addresses on the module map.
                end for
        end for
```

Figure 5.9: Algorithm for the module simulator.

Using the module grammar, the Simulator constructs a binary calculation tree for the given module. The Simulator uses this tree as a template to execute module instructions. Figure 5.10 shows an example module grammar and the binary calculation tree for the given grammar. In the tree, nodes can be either a mathematical operator or a data node. Each operator node holds a mathematical operator that is applied to the results of left and right sub-trees of the node. Data nodes can hold an array element, a singe variable or a constant value.

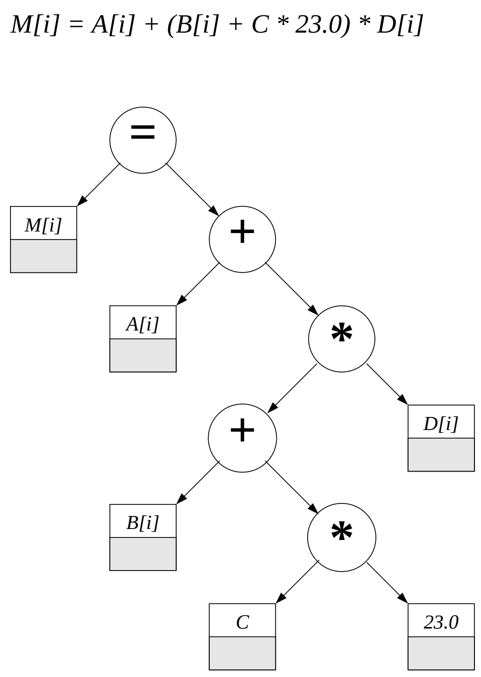


Figure 5.10: Calculation tree.

In the next step of the algorithm, the Simulator constructs an operand map. During the compilation phase, the RCCT Compiler uses the same module grammars provided in the `module.lib` file and writes module instructions. While writing module instructions, the Compiler does not keep the operands in the order they appear in module grammar. Depending on parameter values specified in `par.txt`, the Compiler reorders the operands. The Simulator has to find the correct order of the operands; therefore, it looks at the parameter file and constructs an operand map while parsing module grammar. This map is used to match the operands read from module instruction generated by the RCCT Compiler and the nodes of the calculation tree, while setting data in the tree. The order of the operands in the tree is equal to their order of appearance in the module grammar when the calculation tree is scanned recursively (left first scan).

After the construction of the map, the algorithm enters a loop and goes through all module instructions. In the loop, first, the module execution time for the current instruction is estimated and is added to the total elapsed time. After that, address fields in the operand map are initialized with the values read from the given module instruction. The algorithm enters the second loop to execute one module instruction. At each iteration of the inner loop, first, a set of data is read from the virtual memory to a temporary data array and then using the map, the calculation tree is initialized with the data read in the temporary array. After initialization, the Simulator evaluates the value of the tree and writes the result back to the virtual memory.

After the Simulator finishes simulating modules, the user application can collect results from the virtual memory. Beside the actual results, the Simulator also reports an estimated execution time for each session in the session file. One session could include utilization of more than one modules on PEs. In such a case, the Simulator assumes that all modules work in parallel on different PEs. For that reason, the Simulator compares all modules' estimated execution time and returns the execution time of a module with the biggest execution time.

A mapped application can include several mapped sections and for some mapped sections actual module configuration file may not be available. When this type of applications are executed using RCCT, the Loader/Simulator pair execute the sections on the RC system for which actual module configuration files are found while simulating the other sections.

Chapter 6

Experimental Setups and Test/Simulation Results

Several applications were mapped using the tool Compiler and the resulting new applications were simulated for different Reconfigurable Computing (RC) environments. The main goal of the experimental tests and simulations was to demonstrate the effectiveness of our Reconfigurable Computing Compilation Tool (RCCT). The other goals of the tests and simulations are:

- Verifying the accuracy and validity of the Simulator.
- Measuring execution time of the selected user applications on general purpose processors.
- Estimating execution time of the mapped user applications using the Simulator.
- Measuring speedup gained when the user applications are mapped to different RC systems.

The applications that we selected are considered computationally complex applications. The common feature of these applications is that most of the operations are performed in one or more nested loops. Since we have floating-point modules and floating-point applications require significantly more CPU time, we selected floating-point versions of the applications. The following is a list of the applications that we mapped using RCCT.

1. Matrix Multiplication
2. 3-D Image Correlation
3. Image Intensity Calculation
4. Frequency Domain Filtering

The chapter is organized as follows. The following section presents how the validity and accuracy of the Simulator is verified. The next section presents the computational environments used in the experiments. The subsequent sections present how the selected applications were mapped to the RC system and how much speedup was attained over the general purpose processor implementation of the same application.

6.1 Validating the Simulator

The accuracy of the Simulator in estimating the execution time of the mapped applications depends on the mathematical models of the modules used in the mapping of these applications. A generalized form of the mathematical models for the modules are introduced in Chapter 3. The Simulator uses these models and the module parameters provided in the module library file to estimate the execution times of the mapped sections of the user applications.

When a section of an application is executed on an RC system using the modules, the total execution time of the section is equal to the module's execution time plus the Application Programming Interface (API) overhead. Our experiences with the WildForce board and the associated API showed that the API overhead for starting and stopping a Processing Element (PE) is about 0.1 millisecond on the average. API overhead is variable and depends on the host computer's processor speed and the work load at the time the API function is called. While estimating the execution time of a user application section, the Simulator is able to calculate the execution time of a module precisely and adds 0.2 millisecond API overhead to calculate the total execution time. Since the API overhead is variable, a small difference occurs between the real execution time and the estimated execution time.

To find the accuracy of the Simulator, we performed some vector operations on the WildForce board using our module implementations introduced in Chapter 3. In these experiments, we utilized only one Processing Element and we clocked the modules at 50 MHz. The vector size(s) for each vector operation was 131000. We let the modules process given vectors and measured the execution time of each vector operation. We also implemented these vector operations in C and mapped them to the same RC system using the RCCT Compiler. This time, instead of running the mapped applications on the RC system, we simulated them using the Simulator and recorded the estimated execution time of each vector operation calculated by the Simulator. Table 6.1 shows the results collected from both the RC system and the Simulator. As seen in the table, except for the accumulator case, the Simulator is able to estimate execution time with less than 2% error. As explained above, this error occurs due to the fact that the API overhead in the real word is variable and the Simulator uses a constant API overhead value. The Accumulator case is not a good example for evaluating the Simulator's performance. Since the module execution time is very small, a little change in the API overhead greatly affects the total execution time.

For better evaluation of the Simulator's performance, we tested it with an implementation of the matrix multiplication algorithm. Again, the Simulator was able to estimate the total execution time with less than 2% error, the second part of the Table 6.1. The results presented in Table 6.1 verifies the validity of the Simulator in estimating execution time.

Table 6.1: Comparing real RC execution times of the modules with the Simulator's estimated execution times.

Module Name	**Vector Size**	**RC Execution**	**Simulator Estimation**	**Difference**	**Percent Error**
4 Op. Vec. Addition	131000	10.80	10.68	0.12	1.11
Accumulation	131000	2.70	2.82	-0.12	-4.30
Multiply-Accumulate	131000	5.43	5.44	-0.01	-0.17

Application Name	**Matrix Size**	**RC Execution**	**Simulator Estimation**	**Difference**	**Percent Error**
Matrix Multiplication	200 x 200	95.25	93.90	1.35	1.43

The Simulator is not only able to estimate the execution time but also able to Simulate modules' functionality. Using the module specifications provided in the module definition file, the Simulator is able to process the given data as if it were a module running on a RC system. Functional correctness of the Simulator was verified by comparing the calculation results collected from the RC system and the Simulator.

6.2 Computing Systems Used in the Experiments

Three General Purpose Processor (GPP) computers were used in the experiments to compare with RC systems. Table 6.2 summarizes the specifications of these computer systems.

Table 6.2: General purpose processors used in the experiments

	GPP1	**GPP2**	**GPP3**
Processor Type	Pentium II	Pentium III	Pentium IV
Processor Speed	400 MHz	866 MHz	1700 MHz
Memory Size	128 MB	256 MB	384 MB
Operating System	Windows NT	Windows 2000	Windows 2000

In addition to the above computer systems, we modeled two actual reconfigurable system for simulation purposes. These are WildForce [83] and WildCard [87] reconfigurable cards. These cards are currently available in our laboratory. Instead of using these cards to test the RCCT tool, we preferred to model and simulate them due to the fact that most of the module designs for selected applications do not fit onto these cards. Compared to the state-of-the-art RC cards, these cards have very limited hardware and memory resources available. While modeling the cards, we assumed that they have adequate memory and hardware resources. Our intention here is to demonstrate how these cards can outperform GPPs even though they run at a much slower clock rate than the GPPs.

The WildForce card has five processing elements and one local memory unit is available for each PE. The maximum clock frequency that can be applied to the PEs is 50 MHz. Since one of the PEs was used as a controlling PE and some part of it was used for implementation of the card's own design, we excluded it when we modeled the card. The second card includes one processing element and two local memories. This card can be clocked up to 100 MHz. Through the rest of the chapter, WildForce and WildCard will be called **RC1** and **RC2**, respectively.

For timing estimations, the Simulator needs to know three parameters of the cards. The first parameter is the number of processing elements available on the card. The availability of more PEs on a card offers more chances for the RCCT Compiler to exploit parallelism in the user applications when they are mapped to the RC systems. The second parameter is the clock rate that can be applied to the processing elements when they are configured with hardware modules. The third parameter is the number of local memory units available for each PE. If there is only one memory unit available for each PE, the PE has to read and write to the same memory unit. On the other hand, if a PE has two memory units, it can read from one memory and write to the other memory unit. PEs with two local memory units can offer more throughput compared to one memory unit PEs.

By changing card parameters required for the Simulator, we defined several virtual reconfigurable systems for our tests. The first virtual system, **VRC1**, contains four PEs and each PE has one local memory unit. We defined the card's clock speed as 200 MHz. The second virtual system, **VRC2**, has the same features as **VRC1** expect for that each PE has two local memory units. This configuration added to the tests to demonstrate effects of dual memory unit PEs in execution time. The third and final virtual RC system, **VRC3**, has four PEs and each PE has one local memory unit. The clock speed for **VRC3** is defined as 400 MHz.

6.3 Application 1: Matrix Multiplication

Matrix multiplication was selected because it is one of the major mathematical operations used in several engineering applications such as image processing and solving linear equations. It is also one of the most popular operations that scientists try to speed up. An ordinary matrix multiplication algorithm performs $O(n^3)$ computation on $O(n^2)$ data where n is the size of a row or column [88]; therefore, it requires huge amount of CPU time especially if the input matrix sizes are larger than 100x100. Hence, we think that matrix multiplication is a good initial candidate application to show effectiveness of our tool. The product of two matrices is defined in Equation 6.1.

$$C_{ij} = \sum_{k=0}^{n-1} A_{ik} B_{kj} \tag{6.1}$$

$$\begin{aligned} i &= 0, 1, 2, \cdots, m-1 \\ j &= 0, 1, 2, \cdots, p-1, \end{aligned}$$

where C, A, and B matrices have dimensions of (m, p), (m, n) and (n, p), respectively.

Figure 6.1 shows the C++ code fragment that performs matrix multiplication on two square matrices. As it can be seen from the source code, the inner most loop (lines `03` and `04`) performs the multiply-accumulate vector operation on the columns and rows of the input matrices to calculate individual elements of the output matrix. The RCCT Compiler is able to identify this vector operation and map it to the RC systems. Since calculation of the output matrix requires $size^2$ times repetition of the inner most loop, the Compiler generates $size^2$

vector operations for the matrix multiplication. Depending on the RC system configuration, the Compiler distributes these vector operations through PEs for parallel calculation of the matrix multiplication.

```
for (i=0; i<size; i++)
   for (j=0; j<size; j++)
      for (k=0; k<size; k++)
         c[i][j] = c[i][j] + a[i][k] * b[k][j];
```

Figure 6.1: The `for` loop for matrix multiplication.

The function core shown in Figure 6.2 was designed and specified in the module definition file for the RCCT Compiler to perform multiply-accumulate vector operation. This function core unit includes two basic cores to perform multiplication and accumulation. The input numbers are first multiplied by the multiplier and are forwarded to the adder for accumulation.

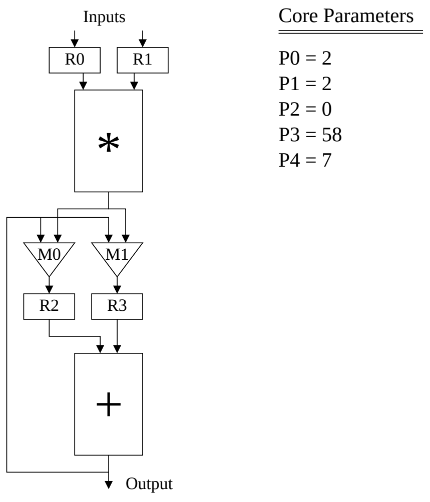


Figure 6.2: Function core design for matrix multiplication.

The multiplier is able to produce one result every other clock cycle. Since the adder is an 8 stage pipeline, adding one output of the multiplier to the current accumulator value normally takes 8 cycles. To speed up the adder and synchronize it with the multiplier, an approach called *rotating partial results* is used. At any moment, the pipelined adder holds four partial results. The adder continuously rotates these partial results through itself and adds the multiplier outputs to these partial results in order. At the end of the vector operation, with help of the module controller, the adder adds up these partial results and sends the final result to the core output.

Since this core is an accumulation core, and it does not have to write a result back to the memory at every *memory cycle*, it can continuously read data from memory. With the help of the *rotating partial results* technique, 100% memory address and data bus utilization and 50% multiplier and adder basic core utilizations were achieved.

Table 6.3 shows the execution times, in milliseconds, of the matrix multiplication algorithm for different matrix sizes on **GPP1**, **GPP2** and **GPP3**. For simplicity, we used square matrices and the matrix sizes were 128 x 128, 256 x 256, 512 x 512 and 1024 x 1024 Each matrix multiplication experiment was run four times on each computer to eliminate the unwanted effects of operating systems on the execution time. Each set of results was evaluated individually. If all four results were very close to each other (if the difference between each individual result and the average of the results was less than 5%), we put the average of them in the table. If three results were close to each other and one result was abnormal, then we eliminated the abnormal result and put the average of the three results in the table. If more that one result looked abnormal, then we repeated the test. This technique worked very well. Less than 10% of the time, one result was abnormal and less than 1% of the time, two results were abnormal. The same technique was used for the other applications.

In Table 6.3, the first column indicates the matrix size and the other columns show the execution time of the algorithm on different computers. When the matrix size was doubled in both dimensions the number of floating-point operations needed to perform matrix multiplication and the execution time increased exponentially. Clearly, this can be observed from the table and this verifies the validity of the tests.

Table 6.3: Matrix multiplication execution times on **GPP**s.

Matrix Size	**GPP1** (msec.)	**GPP2** (msec.)	**GPP3** (msec.)
128 x 128	227.29	82.41	36.89
256 x 256	1758.69	816.11	471.37
512 x 512	19427.59	8396.39	4348.76
1024 x 1024	173185.50	71102.99	40277.95

Table 6.4 shows the RCCT Simulator's estimated execution times of the matrix multiplication on different RC environments for different matrix sizes. Note that **VRC1** and **VRC2** showed the same performance. They both have the same number of PEs and their clock rate is the same, but **VRC2** has a dual memory unit for each PE. Since the function core designed for matrix multiplication is an accumulation core, it is not able to take advantage of dual memory PEs. This happens due to the fact that the core has to write only one result at the end of each vector operation and this is negligible.

Tables 6.5, 6.6, and 6.7 give the estimated speedups of various RC configurations over **GPP1**, **GPP2** and **GPP3**, respectively. The first column of each table gives the matrix size. The subsequent columns indicate the speedup attainable by the RC systems when the RCCT is used. Speedups in these tables were calculated dividing execution time of **GPPx** by the estimated execution time of **(V)RCx**.

Table 6.4: Estimated execution times of matrix multiplication on the RC systems.

Matrix Size	**RC1** (msec.)	**RC2** (msec.)	**VRC1** (msec.)	**VRC2** (msec.)	**VRC3** (msec.)
128 x 128	26.96	53.12	7.12	7.12	3.81
256 x 256	190.23	379.65	47.93	47.93	24.22
512 x 512	1430.50	2860.19	358.00	358.00	179.25
1024 x 1024	11089.19	22177.58	2772.67	2772.67	1386.59

Table 6.5: Estimated speedup of the RC systems over **GPP1** for matrix multiplication.

Matrix Size	**RC1 Speedup**	**RC2 Speedup**	**VRC1 Speedup**	**VRC2 Speedup**	**VRC3 Speedup**
128 x 128	8.431	4.279	31.945	31.945	59.695
256 x 256	9.245	4.632	36.692	36.692	72.626
512 x 512	13.581	6.792	54.267	54.267	108.383
1024 x 1024	15.618	7.809	62.462	62.462	124.901

Table 6.6: Estimated speedup of the RC systems over **GPP2** for matrix multiplication.

Matrix Size	**RC1 Speedup**	**RC2 Speedup**	**VRC1 Speedup**	**VRC2 Speedup**	**VRC3 Speedup**
128 x 128	3.057	1.551	11.583	11.583	21.644
256 x 256	4.290	2.150	17.027	17.027	33.701
512 x 512	5.870	2.936	23.454	23.454	46.842
1024 x 1024	6.412	3.206	25.644	25.644	51.279

Table 6.7: Estimated speedup of the RC systems over **GPP3** for matrix multiplication.

Matrix Size	**RC1 Speedup**	**RC2 Speedup**	**VRC1 Speedup**	**VRC2 Speedup**	**VRC3 Speedup**
128 x 128	1.368	0.694	5.185	5.185	9.689
256 x 256	2.478	1.242	9.834	9.834	19.465
512 x 512	3.040	1.520	12.147	12.147	24.261
1024 x 1024	3.632	1.816	14.527	14.527	29.048

6.4 Application 2: 3-D Image Correlation

Image Correlation is an image processing algorithm typically used for template matching. Using this algorithm, the target image is compared with the input image. The algorithm processes two images and finds a position on the input image where the best match happens between the input image and the target image. It has many application areas from medical imaging to military applications.

Image correlation is basically image convolution. Computational complexity of the regular implementation of the algorithm for three-dimensional square images is $O(n^6)$ [89]. Due to its popularity and high demand for CPU time, we selected the correlation algorithm as an example application. Nikolaidis et. al. [89] implemented the image correlation algorithm for three dimensional images. Their algorithm first transforms the input and target images from the time domain to the frequency domain and then performs a multiplication of the two transformed images. After the multiplication, the result is transformed back to the time domain. The transformations are done by a special function developed by Nikolaidis et. al. on the general purpose processor.

The correlation algorithm includes three `for` loop blocks. The first `for` block initializes the temporary arrays with zero and the second `for` block copies the input and target images to the temporary arrays as shown in Figure 6.3. Since both of these `for` blocks include assignment operations and there is no module definition provided in the module library file for this operation, the RCCT Compiler parsed these `for` blocks but did not map them to the RC systems.

```
for(nf=0,nff=0; nf<L2;nf++, nff++)
   for(nr=0,nrr=0; nr<N2;nr++, nrr++)
      for(nc=0,ncc=0; nc<M2;nc++, ncc++)
      {
         matmere1[nff][nrr][ncc] = mats1[nf][nr][nc];
         matmere2[nff][nrr][ncc] = mats2[nf][nr][nc];
      }
```

Figure 6.3: A sample `for` loop block in 3-D image correlation (Not mapped).

Figure 6.4 shows the third `for` loop block which performs the multiplication of the transformed images. The inner most loop includes two expressions which are similar. For these expressions we designed the function core shown in Figure 6.5 and specified it in the module definition file. When the source code is compiled with the RCCT Compiler, the Compiler matched these two expressions in the third `for` block with the module specification in the module definition file and mapped the `for` block to the RC systems. While mapping a block, the Compiler checks the expressions if there is any data dependency between them. In this `for` block, the Compiler did not find any data dependency between the expressions. The Compiler generated two session files for this block, one for each expression.

```
for(nf=0;nf<frames2;nf++)
   for(nr=0;nr<rows2;nr++)
      for(nc=0;nc<columns2;nc++)
      {
         Mmere1[nf][nr][nc] = matmere1[nf][nr][nc] * matmeim1[nf][nr][nc]
                            + matmere2[nf][nr][nc] * matmeim2[nf][nr][nc];
         Mmeim1[nf][nr][nc] = matmere2[nf][nr][nc] * matmeim1[nf][nr][nc]
                            + matmere1[nf][nr][nc] * matmeim2[nf][nr][nc];
      }
```

Figure 6.4: Mapped `for` loop block in the 3-D image correlation algorithm.

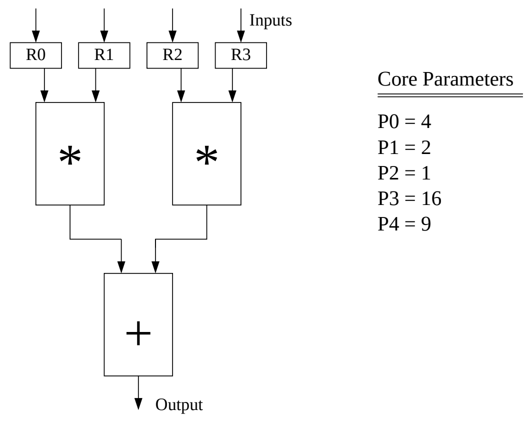


Figure 6.5: Function core design for 3-D image correlation.

Table 6.8 shows the execution times when the algorithm was run on **GPP1**, **GPP2** and **GPP3**. The first column shows the size of the three dimensional images. The first number specifies the frame count and the subsequent numbers specify the number of columns and rows in each frame, respectively. The algorithm uses several temporary arrays and the size of these temporary arrays is equal to two times the size of the original image in each dimension. **GPP1** fails for the images of size 4 x 400 x 600 and up and **GPP2** fails for the images of size 4 x 400 x 800. These two computers actually completed the job but due to their memory limitations they used virtual memory on the hard drive to store arrays used by the algorithm. Using virtual memory makes these two computers extremely slow and was not fair for comparisons with the RC systems for the image sizes indicated above.

Table 6.9 shows the execution times, in milliseconds, estimated by the RCCT Simulator. Since the function core designed for this application is not an accumulation module it takes advantage of double memory unit PEs. Clearly, this can be seen when the estimated execution times of **VRC1** and **VRC2** are compared.

Table 6.8: 3-D image correlation execution times on **GPP**s.

3D Image Size	**GPP1** (msec.)	**GPP2** (msec.)	**GPP3** (msec.)
2 x 100 x 200	100.41	54.42	21.83
4 x 100 x 200	203.70	108.89	43.56
2 x 200 x 400	411.45	220.58	70.64
4 x 200 x 400	820.82	442.09	142.53
2 x 400 x 600	1241.41	651.14	193.90
4 x 400 x 600	Failed	1308.07	396.94
2 x 400 x 800	Failed	869.93	262.79
4 x 400 x 800	Failed	Failed	526.71

Table 6.9: Estimated execution times of 3-D image correlation on the RC systems.

3-D Image Size	**RC1** (msec.)	**RC2** (msec.)	**VRC1** (msec.)	**VRC2** (msec.)	**VRC3** (msec.)
2 x 100 x 200	20.38	26.40	5.85	4.25	3.42
4 x 100 x 200	39.77	52.40	10.69	7.50	5.85
2 x 200 x 400	78.17	103.60	20.29	13.90	10.65
4 x 200 x 400	155.34	206.80	39.58	26.80	20.29
2 x 400 x 600	232.14	309.20	58.78	39.60	29.89
4 x 400 x 600	463.27	618.00	116.57	78.20	58.78
2 x 400 x 800	308.94	411.60	77.98	52.40	39.49
4 x 400 x 800	616.87	822.80	154.97	103.80	77.98

Tables 6.10, 6.11, and 6.12 shows the estimated speedup of **RC1**, **RC2**, **VRC1**, **VRC2**, **VRC3** over three different general purpose processors. The first column in each table indicates the size of three dimensional input images and the other columns shows the speedup attainable by the RC systems.

Table 6.10: Estimated speedup of the RC systems over **GPP1** for 3-D image correlation.

3-D Image Size	**RC1 Speedup**	**RC2 Speedup**	**VRC1 Speedup**	**VRC2 Speedup**	**VRC3 Speedup**
2 x 100 x 200	4.926	3.803	17.176	23.626	29.334
4 x 100 x 200	5.122	3.887	19.052	27.160	34.844
2 x 200 x 400	5.264	3.972	20.276	29.601	38.648
4 x 200 x 400	5.284	3.969	20.736	30.628	40.450
2 x 400 x 600	5.348	4.015	21.118	31.349	41.530

Table 6.11: Estimated speedup of the RC systems over **GPP2** for 3-D image correlation.

3-D Image Size	**RC1 Speedup**	**RC2 Speedup**	**VRC1 Speedup**	**VRC2 Speedup**	**VRC3 Speedup**
2 x 100 x 200	2.670	2.061	9.309	12.805	15.898
4 x 100 x 200	2.738	2.078	10.184	14.519	18.626
2 x 200 x 400	2.822	2.129	10.870	15.869	20.720
4 x 200 x 400	2.846	2.138	11.168	16.496	21.786
2 x 400 x 600	2.805	2.106	11.077	16.443	21.783
4 x 400 x 600	2.824	2.117	11.222	16.727	22.252
2 x 400 x 800	2.816	2.114	11.155	16.602	22.028

Table 6.12: Estimated speedup of the RC systems over **GPP3** for 3-D image correlation.

3-D Image Size	**RC1 Speedup**	**RC2 Speedup**	**VRC1 Speedup**	**VRC2 Speedup**	**VRC3 Speedup**
2 x 100 x 200	1.071	0.827	3.734	5.136	6.377
4 x 100 x 200	1.095	0.831	4.074	5.808	7.451
2 x 200 x 400	0.904	0.682	3.481	5.082	6.635
4 x 200 x 400	0.918	0.689	3.601	5.318	7.024
2 x 400 x 600	0.835	0.627	3.299	4.896	6.487
4 x 400 x 600	0.857	0.642	3.405	5.076	6.753
2 x 400 x 800	0.851	0.638	3.370	5.015	6.654
4 x 400 x 800	0.854	0.640	3.399	5.074	6.754

6.5 Application 3: Image Intensity Calculation

Color images are represented by the combination of three gray scale images. Each gray scale image represents a different main color, red, green and blue (RGB). The algorithms developed for gray scale images are applied to all three gray scale images to analyze one color image. Applying the same algorithm to three gray scale images triples the amount of data needed to process a color image compared to a gray scale image. One way to solve this problem is to perform data reduction algorithms prior to running any image analysis algorithm. The most frequently used data reduction algorithms are image intensity and hue algorithms. Seul et. al. [90] implemented these algorithms in one application.

Figure 6.6 and 6.7 show the code segments that perform hue and intensity calculations, respectively. After parsing the application source code, the RCCT Compiler only mapped the second `for` loop block and kept the first block unchanged. The first block, hue calculations, was not mapped by the Compiler due to the fact that the Compiler detected function calls inside the for block. As shown in Figure 6.6, lines `09` and `16` includes `sqrt()` and `acos()` functions, respectively. The nature of RC mapping process forbids the Compiler mapping these types of `for` blocks to the RC systems. If the Compiler maps a `for` including a function call, during the execution of the `for` block, the RC system will not be able to evaluate the function call; therefore, the Compiler does not map this kind of `for` blocks.

```
for (iy = 0; iy < height; iy++) {
   for (ix = 0; ix < width; ix++) {
      r = ImgIn[0][iy][ix] / (float)255.0;
      g = ImgIn[1][iy][ix] / (float)255.0;
      b = ImgIn[2][iy][ix] / (float)255.0;
      c = (float)(0.5 * (2.0 * r - g - b));
      d = (float)(sqrt ((r - g) * (r - g) + (r - b) * (g - b)));
      if (d == 0.0)
         ImgOut[iy][ix] = 255.0;  /* arbitrary value -> hue undefined */
      else {
         temp = c / d;         /* imprecision causes > |1| */
         if (temp > 1.0)
            temp = 1.0;
         else if (temp < -1.0)
            temp = -1.0;
         temp = (float)(acos (temp));
         if (b > g)
            temp = (float)(2.0 * M_PI - temp);
         ImgOut[iy][ix] = (float)(temp * 100.0 / M_PI);  /* scale 0-200 */
      }
   }
}
```

Figure 6.6: The `for` loop block for image hue calculation (Not Mapped).

```
for (iy = 0; iy < height; iy++) {
   for (ix = 0; ix < width; ix++) {
      ImgOut2[iy][ix] = (ImgIn[0][iy][ix] + ImgIn[1][iy][ix] +
                         ImgIn[2][iy][ix]) / (float)3.0 + (float)0.5;
   }
}
```

Figure 6.7: The `for` loop block for image intensity calculation (Mapped).

We designed the function core shown in Figure 6.8, and specified it in the module definition file to help the Compiler in mapping the intensity `for` block shown in Figure 6.7. Since each basic core inside the function core has 8 stages pipelined, the total pipeline stages of the function core unit is equal to 32. One delay unit was used to synchronize the data streams inside the function core unit. Two 32-bit registers were used to store constant values of the expression in the `for` block. The module reads these constant values only once at the beginning of a vector operation and uses them until the end of the operation. When the source code is mapped, the Compiler matched the module definition with the expression inside the `for` loop and mapped the block to the RC systems. The Compiler generated only one session file for this block.

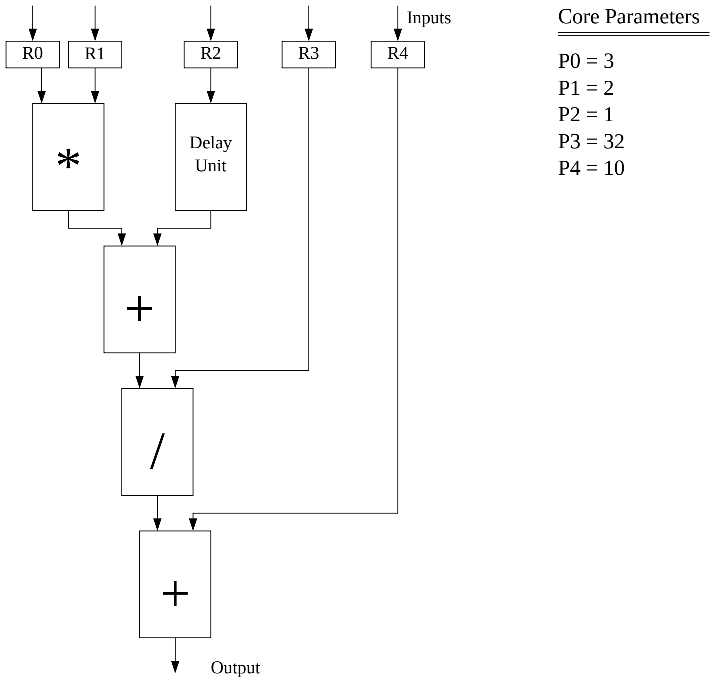


Figure 6.8: Function core design for image intensity calculation.

Table 6.13 shows the execution times of the intensity calculation `for` block on **GPP1**, **GPP2**, and **GPP3**. The first column of the table indicates the image size and the others indicate the execution times in milliseconds. The execution time of the `for` loop block increases parallel to the number of floating-point operations needed to process the given image. For example, on row 2, the image size is 200 x 200 and the execution time is 7.68 milliseconds on **GPP1**. On row three, the image size is doubled in both dimensions. This means that the number of floating-point operations needed to process a given image increases four times and the execution time is quadrupled on **GPP1**. This verifies the correctness of the data collected from the tests.

Table 6.13: Image intensity calculation execution times on **GPP**s.

Image Size	**GPP1** (msec.)	**GPP2** (msec.)	**GPP3** (msec.)
100 x 100	1.61	0.78	0.28
200 x 200	7.68	4.58	1.31
400 x 400	31.75	18.08	4.65
800 x 800	131.03	72.65	18.69
1000 x 1000	203.83	117.52	29.24
2000 x 2000	814.15	460.77	113.93

Table 6.14 shows the estimated execution time of the intensity `for` block on different RC systems. Since the function core design is not an accumulation core, this application is also able to take advantage of dual memory unit PEs systems. The effect of dual memory can be seen clearly when results for **VRC1** and **VRC2** are compared. For smaller image sizes, because of constant C_{API} overhead, estimated execution times seem very close on **VRC1** and **VRC2**. The difference can be observed clearly on larger image sizes.

Table 6.14: Estimated execution times of image intensity calculation on the RC systems.

Image Size	**RC1** (msec.)	**RC2** (msec.)	**VRC1** (msec.)	**VRC2** (msec.)	**VRC3** (msec.)
100 x 100	1.77	0.93	1.57	1.54	1.53
200 x 200	2.53	1.87	1.76	1.66	1.63
400 x 400	5.56	5.54	2.52	2.12	2.01
800 x 800	17.63	20.07	5.53	3.93	3.52
1000 x 1000	26.66	30.94	7.79	5.29	4.65
2000 x 2000	101.82	121.28	26.58	16.59	14.04

Tables 6.15, 6.16, and 6.17 shows the estimated attainable speedups of **RC1**, **RC2**, **VRC1**, **VRC1**, **VRC1** over **GPP1**, **GPP2** and **GPP3**. For small images speedup is very small, and in some cases it is less than 1 time. This happens because of two reasons. First, when the algorithm executed for smaller images general purpose processors takes the advantage of cache memory. Second, there is a greater influence of C_{API} on estimated execution time for smaller images.

Table 6.15: Estimated speedup of the RC systems over **GPP1** for image intensity calculation.

Image Size	**RC1 Speedup**	**RC2 Speedup**	**VRC1 Speedup**	**VRC2 Speedup**	**VRC3 Speedup**
100 x 100	0.912	1.724	1.028	1.044	1.050
200 x 200	3.033	4.111	4.369	4.631	4.715
400 x 400	5.706	5.735	12.619	14.998	15.812
800 x 800	7.433	6.528	23.686	33.307	37.267
1000 x 1000	7.646	6.588	26.166	38.513	43.882
2000 x 2000	7.996	6.713	30.630	49.090	57.988

Table 6.16: Estimated speedup of the RC systems over **GPP2** for image intensity calculation.

Image Size	**RC1 Speedup**	**RC2 Speedup**	**VRC1 Speedup**	**VRC2 Speedup**	**VRC3 Speedup**
100 x 100	0.442	0.835	0.498	0.506	0.509
200 x 200	1.809	2.452	2.605	2.762	2.812
400 x 400	3.249	3.266	7.186	8.540	9.004
800 x 800	4.121	3.619	13.133	18.467	20.663
1000 x 1000	4.408	3.798	15.086	22.205	25.300
2000 x 2000	4.525	3.799	17.335	27.782	32.818

Table 6.17: Estimated speedup of the RC systems over **GPP3** for image intensity calculation.

Image Size	**RC1 Speedup**	**RC2 Speedup**	**VRC1 Speedup**	**VRC2 Speedup**	**VRC3 Speedup**
100 x 100	0.159	0.300	0.179	0.182	0.183
200 x 200	0.517	0.701	0.745	0.790	0.804
400 x 400	0.836	0.840	1.848	2.197	2.316
800 x 800	1.060	0.931	3.379	4.751	5.316
1000 x 1000	1.097	0.945	3.754	5.525	6.295
2000 x 2000	1.119	0.939	4.286	6.869	8.115

6.6 Application 4: Frequency Domain Filter

The image filterization process can be performed in the frequency domain. For this purpose, the image is transfered from the time domain to the frequency domain. Then, the filterization process is applied to the image in the frequency domain. After that, the image is transformed back to the time domain by applying the inverse transformation. Seul et. al. [90] implemented the image filterization algorithm in C++. The implementation includes several `for` loops. When we compiled it with the RCCT Compiler, the Compiler selected four `for` blocks to map the RC systems shown in Figure 6.9.

```
for (y = 0; y < nRow; y++)
   for (x = 0; x < nCol; x++)
      imgIn[y][x] = 255 - imgIn[y][x];
```

```
for (y = 0; y < nRow; y++)
   for (x = 0; x < nCol; x++)
      image[y][x] = image[y][x] * hammingX[x] * hammingY[y];
```

```
for (y = 0; y < nRow2; y++)
   for (x = 0; x < nCol2; x++)
      imgOut[y][x] = (unsigned char) ((imgReal[y][x] - min) * norm + 0.5);
```

```
for (y = 0; y < nRow2; y++)
   for (x = 0; x < nCol2; x++)
      imgOut[y][x] = 255 - imgOut[y][x];
```

Figure 6.9: Mapped `for` loop blocks in frequency domain filter.

We designed three function core units shown in Figure 6.10 for the `for` blocks selected by the RCCT Compiler. The function core design in Figure 6.10a was used both for the first and the last `for` blocks in Figure 6.9. The core design in Figure 6.10b and 6.10c were used for calculation of the second and third `for` blocks in Figure 6.9, respectively.

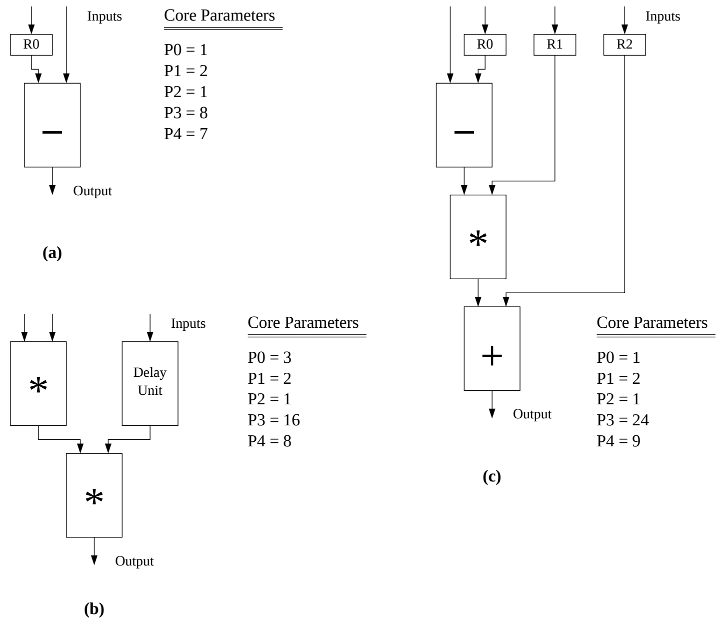


Figure 6.10: Function core designs for frequency domain filter.

Table 6.18 shows the execution time of the selected `for` blocks on **GPP**s. The first column indicates the image size and the other columns show total execution time of the four `for` blocks in milliseconds.

Table 6.18: Frequency domain filtering execution times on **GPP**s.

Image Size	**GPP1** (msec.)	**GPP2** (msec.)	**GPP3** (msec.)
128 x 128	26.35	14.38	5.04
128 x 256	55.97	29.33	9.98
256 x 256	113.47	59.34	20.21
256 x 512	226.70	118.80	40.17
512 x 512	453.04	238.28	80.20
512 x 1024	916.47	477.63	165.86
1024 x 1024	1815.81	951.32	319.43
1024 x 2048	22459.31	1913.69	653.26

The RCCT Simulator's estimated execution times are presented in Table 6.19. Again in this table, the first column indicates the image size and the other columns indicate the total estimated execution time of the four mapped `for` blocks.

Table 6.19: Estimated execution times of frequency domain filtering on the RC systems.

Image Size **Size**	**RC1** (msec.)	**RC2** (msec.)	**VRC1** (msec.)	**VRC2** (msec.)	**VRC3** (msec.)
128 x 128	8.70	6.05	6.68	6.46	6.34
128 x 256	11.32	9.49	7.33	6.89	6.67
256 x 256	16.65	16.58	8.66	7.77	7.33
256 x 512	27.13	30.34	11.28	9.49	8.64
512 x 512	48.26	58.28	16.57	12.99	11.28
512 x 1024	90.21	113.34	27.05	19.87	16.53
1024 x 1024	174.41	224.27	48.10	33.73	27.05
1024 x 2048	342.18	444.47	90.05	61.26	48.02

Tables 6.20, 6.21, and 6.22 show the total attainable speedups of mapped four `for` blocks. Since none of the function cores designed for this applications is accumulation, dual memory PEs offer better performance gain over the **GPP**s.

Table 6.20: Estimated speedup of RC systems over **GPP1** for frequency domain filtering.

Image Size	**RC1 Speedup**	**RC2 Speedup**	**VRC1 Speedup**	**VRC2 Speedup**	**VRC3 Speedup**
128 x 128	3.03	4.36	3.95	4.08	4.16
128 x 256	4.94	5.90	7.64	8.13	8.40
256 x 256	6.82	6.84	13.10	14.60	15.48
256 x 512	8.36	7.47	20.09	23.88	26.23
512 x 512	9.39	7.77	27.35	34.89	40.15
512 x 1024	10.16	8.09	33.88	46.13	55.46
1024 x 1024	10.41	8.10	37.75	53.83	67.12
1024 x 2048	65.64	50.53	249.42	366.63	467.68

Table 6.21: Estimated speedup of the RC systems over **GPP2** for frequency domain filtering.

Image Size	**RC1 Speedup**	**RC2 Speedup**	**VRC1 Speedup**	**VRC2 Speedup**	**VRC3 Speedup**
128 x 128	1.65	2.38	2.15	2.23	2.27
128 x 256	2.59	3.09	4.00	4.26	4.40
256 x 256	3.56	3.58	6.85	7.63	8.09
256 x 512	4.38	3.92	10.53	12.51	13.75
512 x 512	4.94	4.09	14.38	18.35	21.12
512 x 1024	5.29	4.21	17.66	24.04	28.90
1024 x 1024	5.45	4.24	19.78	28.20	35.17
1024 x 2048	5.59	4.31	21.25	31.24	39.85

Table 6.22: Estimated speedup of the RC systems over **GPP3** for frequency domain filtering.

Image Size	**RC1 Speedup**	**RC2 Speedup**	**VRC1 Speedup**	**VRC2 Speedup**	**VRC3 Speedup**
128 x 128	0.58	0.83	0.76	0.78	0.80
128 x 256	0.88	1.05	1.36	1.45	1.50
256 x 256	1.21	1.22	2.33	2.60	2.76
256 x 512	1.48	1.32	3.56	4.23	4.65
512 x 512	1.66	1.38	4.84	6.18	7.11
512 x 1024	1.84	1.46	6.13	8.35	10.04
1024 x 1024	1.83	1.42	6.64	9.47	11.81
1024 x 2048	1.91	1.47	7.25	10.66	13.60

6.7 Summary of the Results

In general, the RC systems significantly outperformed the GGPs except for some rare cases. **RC1** and **RC2** are the same age RC systems with **GPP1**, and **VRC1**, **VRC2** and **VRC3** are the same age RC systems with **GPP2** and **GPP3**. To be fair, we only look at the comparisons between the same age RC systems and GPPs.

Figures 6.11, 6.12, 6.13 and 6.14 show the estimated speedups of the RC systems compared to GPPs for four selected applications. **RC1** and **RC2** can speed up applications 2 to 15 times compared to **GPP1**. Although **RC1** and **RC2** are very slow systems compared to the *state-of-the-art* reconfigurable systems, they were able to outperform today's *state-of-the-art* GPPs in most cases. The results showed that **VRC1**, **VRC2** and **VRC3** can speed up the applications 3 to 125 times compared to **GPP2** and **GPP3**. In most cases, the speedups of the RC systems are more that 10 times.

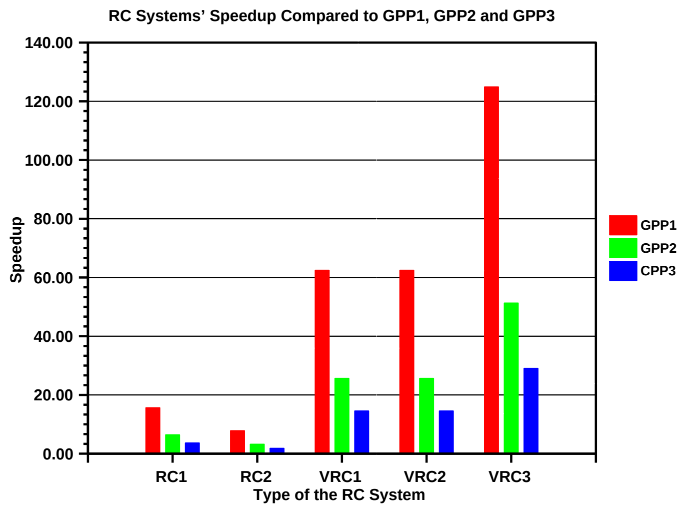


Figure 6.11: Estimated speedups of the RC systems for matrix multiplication (matrix size is 1024 x 1024).

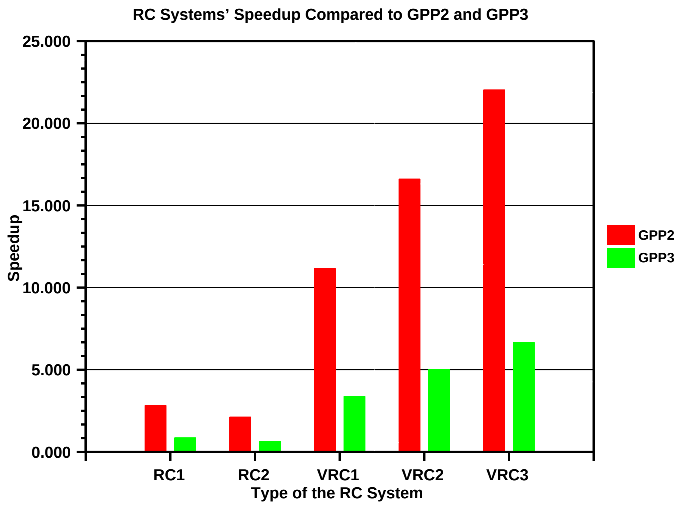


Figure 6.12: Estimated speedups of the RC systems for 3-D image correlation (image size is 2 x 400 x 800).

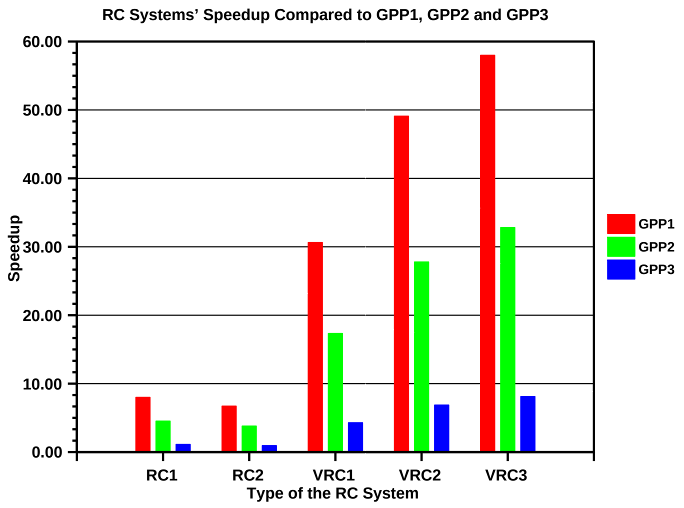


Figure 6.13: Estimated speedups of the RC systems for image intensity calculation (image size is 2000 x 2000).

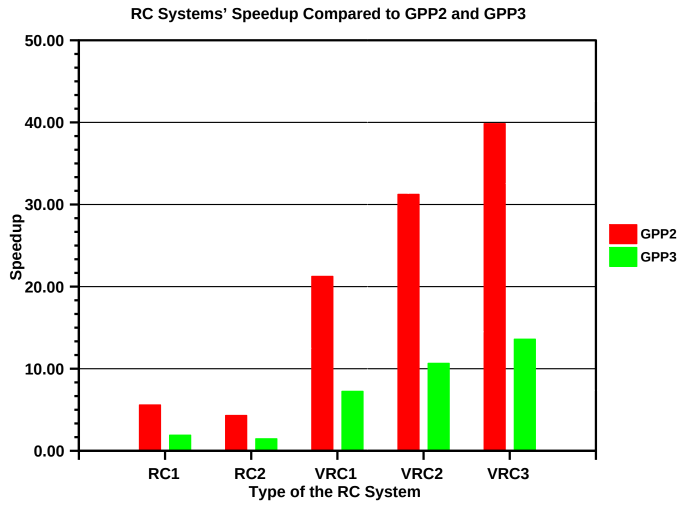


Figure 6.14: Estimated speedups of the RC systems for frequency domain filter (image size is 1024 x 2048).

VRC1 and **VRC2** have the same number of PEs and their clock speeds are the same, but PEs in **VRC2** have dual memory units. **VRC2** is added to the tests to show the effects of the dual memory unit PEs in the performance of the RC systems. In matrix multiplications, accumulation modules were used; therefore, the dual memory unit PEs did not showed any additional performance gain. As illustrated in Figure 6.11, results for **VRC1** and **VRC2** are the same. This happened due to the fact that the accumulation modules does not need to write a result back to memory at every iteration of the module; therefore, the second memory unit reserved for the results has no effect on the performance of the RC system. On the other hand, all other three applications use non-accumulation modules, and dual memory unit PEs showed an additional 45% to 60% performance gain compared to the single memory unit PEs. This can be seen in Figures 6.12, 6.13 and 6.14 when **VRC1** and **VRC2** are compared.

During our experiments and simulations we noticed one weak point of the current RC systems which is the data transfer rate between the PEs and the

PE memories. PEs in the future RC systems must be supported with multiple memory units to increase data transfer rate between the PE and the PE memories. For example, our third generation module includes function cores that can handle complex arithmetic operations in a pipelined fashion. Each function core has multiple data inputs. With the current RC systems, all core inputs are fed from a single memory unit resulting in a poor core utilization. The cores must be supplied with multiple data inputs from multiple memory units to increase the core utilization and to further enhance the performance of the module and the user applications. We also noticed that to exploit parallelism in the user applications, future RC systems must have multiple PEs.

Chapter 7

Conclusion and Future Research Possibilities

Beside their performance advantages over general-purpose processors, reconfigurable (RC) systems have a few disadvantages. First, RC systems require more application development time than general purpose processors, but significantly less than developing an application specific integrated circuit. Second, RC system designers need to be knowledgeable in the areas of hardware and software system design. Third, since each application is different in terms of data inputs, outputs, and the method of processing data, designers are required to design a specific RC implementation for each specific problem.

Our main contribution in this research is the development a design automation tool called Reconfigurable Computing Compilation Tool (RCCT). By developing the tool, our major goals were to address the problems mentioned above and to automate the process of mapping applications onto the RC systems.

The tool includes four majors components. These are: The Compiler, the Hardware Module Library, a generalized interface program called Loader and the Simulator. The purpose of the Compiler is to identify computationally complex portions of user applications and replace them with appropriate function calls to the Loader so that these portions can be executed on RC systems. The Compiler is also responsible for writing session files that includes vector instructions extracted from the selected portions of user applications. The Loader's job is to work as an interface between the modified user applications and the RC resources. When it is called by a modified section of a user application, it executes that section of the application by execution the session file written for that specific section on the RC systems. The Module library includes configuration files of hardware modules that were specifically designed and implemented to perform vector operations on RC systems. The Loader uses these modules to execute vector instructions written in session files on RC systems. We also developed a Simulator to assist the user of the tool in evaluating performance of new RC systems or a new hardware module design before the module is implemented. Additionally, a novel assembly language instruction set for the hardware modules and a session file format, a new assembly language program format for RC systems, were developed.

The tool was tested on several applications to demonstrate its effectiveness. We selected matrix multiplication, and some image processing algorithms such as 3-D Image correlation, to test the tool. First, the applications were compiled with the tool Compiler. The tool Compiler selected the computationally complex sections of the applications and mapped them to the RC systems. Then, the mapped applications were simulated with the tool's Loader-Simulator pair. For some of the applications, some new hardware module designs were added to the Module Library. The selected applications were also run on General Purpose Processors for comparison purposes. We compared the execution times of the mapped sections of the applications when they were run on different GPPs and when they were mapped to different RC configurations to demonstrate the tools effectiveness.

Our results showed that the tool is able to enhance the performance of the applications by mapping portions of them to the RC systems. The tool's Simulator showed that when the user applications are mapped to the RC systems, significant speedups (around 10 times to 100 times) can be attained for the mapped sections of the applications. We also noticed that the design and implementation time of the RC versions of the applications were reduced significantly. With the tool in a matter of minutes RC version of the applications were created. It is also observed that with RCCT, no special skills are needed to map applications to RC systems if the required hardware modules are available.

7.1 Future Research

A fair amount of future research possibilities are available. The following is a list of some important improvements can be done to enhance the performance of the RCCT tool.

1. Currently, the Module Library includes a limited number of hardware module implementations. More modules can be designed and implemented to increase the Compiler's chance to map more applications.

2. During the module matching step, the current version of the Compiler performs a one-to-one comparison between the hardware module specifications and the expressions it finds in the given user application. When the Compiler finds a complex expression for which a module definition has not been done, it skips the expression. With an additional step to the Compiler,

these types of expressions can be divided in to some smaller expressions that can be calculated by the basic hardware modules available in the module library.

3. If there is a data dependency between two expressions in a `for` loop block, the Compiler skips the block. The Compiler can be improved to handled some data dependency types.

4. Java/Fortran are other high level programming languages that are used to implement computationally complex scientific applications. A new version of the tool can be developed to map applications implemented in Java/Fortran. For this purpose, the front end of the Compiler has to be re-designed to be able to scan and parse Fortran source code. Also, some modifications may be need in the Loader for interfacing with an executable code generated from a Fortran source code.

Bibliography

[1] D. Bhatia, "Reconfigurable computing," *Tenth International Conference on VLSI Design*, pp. 356–359, Jan. 1997.

[2] F. Rincon and L. Teres, "Reconfigurable Hardware Systems," *1998 International Semiconductor Conference*, vol. 1, pp. 45–54, Oct. 1998.

[3] Xilinx Inc, *The Programmable Logic Data Book*, San Jose, CA, 1994.

[4] E. Cerro-Prada, S.M. Charlwood, P.B., and James-Roxby, "Image Processing and Its Applications," *Seventh International Conference on Image Processing and Its Applications*, vol. 1, pp. 450–454, Jul. 1999.

[5] R.C.D.M. Tavares, C.J.N. Jr. Coelho, A.D.A. Araujo, and A.O. Fernandes, "Implementation of an Edge Detection Algorithm in a Reconfigurable Computing System," *Proceedings of the Eleventh XI Brazilian Symposium on Integrated Circuit Design*, pp. 38–41, Sep. 1998.

[6] M. A. Figueiredo and C. Gloster, "Implementation of a Probabilistic Neural Network for Multi-spectral Image Classification on an FPGA Based Custom Computing Machine," *Proceedings of 5th Brazilian Symposium on Neural Networks*, pp. 174–179, Dec. 1998.

[7] M. Figueiredo, C. Gloster, M Stephens, C Graves, and M. Nakkar, "Implementation of Multi-spectral Image Classification on a Remote Adaptive Computer," *Journal of VLSI Design Special Issue on Reconfigurable Computing*, vol. 10, no. 3 pp. 307–319, 2000.

[8] P. Graham and B. Nelson, "Genetic Algorithms in Software and in Hardware," *Fourth IEEE Workshop on FPGAs for Custom Computing Machines*, Apr. 1996.

[9] H. Hogl, A. Kugel, J. Ludvig, R. Manner, K.H. Noffz, and R. Zoz, "Enable++: A Second Generation FPGA Processor," *Third IEEE Workshop on FPGAs for Custom Computing Machines*, 1995.

[10] W.B. Ligon III, S. McMillan, G. Monn, K. Schoonover, F. Stivers, and K.D. Underwood, "A Re-evaluation of the Practicality of Floating-point Operations on FPGAs," *Proceedings of the IEEE Symposium on FPGAs for Custom Computing Machines*, Apr. 1998.

[11] A. DeHon and J. Wawrzynek, "Reconfigurable Computing: What, Why, and Implications for Design Automation," *Proceedings of 36th Design Automation Conference*, pp. 610–615, New Orleans, Louisiana, 1999.

[12] J. Villasenor and B. Hutchings, "The Flexibility of Configurable Computing," *IEEE Signal Processing Magazine*, vol. 15, no. 5, pp. 67–84, 1998.

[13] M. John S. Smith, *Application-Specific Integrated Circuits*. Addison-Wesley Inc., 1997.

[14] R. Tessier and W. Burleson, "Reconfigurable Computing for Digital Signal Processing: A Survey," *Journal of VLSI Signal Processing*, vol. 28, pp. 7–27, 1998.

[15] S. Hauck, "The Roles of FPGSs in Reprogrammable Systems," *Proceedings of the IEEE*, pp. 615–638, 1998.

[16] S. Brown and J. Rose, "Architecture of FPGSs and CPLDs: A Tutorial," 2002. http://klabs.org/richcontent/Tutorial/fpga/Toronto_tutorial.pdf.

[17] "Field Programmable Gate Arrays (FPGAs) An Enabling Technology," 2002. http://www.vcc.com/fpga4000.html.

[18] Xilinx Inc, *Virtex-II Pro^{TM} Platform FPGAs: Functional Description*, San Jose, California, 2002.

[19] D. A. Buell, J. M. Arnold, and W. J. Kleinfelder, *SPLASH 2: FPGAs for Custom Computing Machines*, IEEE Computer Society Press, Los Alamitos, 1996.

[20] N. K. Ratha and A. K. Jain, "Computer Vision Algorithms on Reconfigurable Logic Arrays," *IEEE Transactions on Parallel and Distributed Systems*, vol. 10, no. 1, pp. 29–43, 1999.

[21] N. K. Ratha, A. K. Jain, and D. T. Rover, "FPGA-Based Coprocessor for Text String Extraction," *IEEE International Workshop on Computer Architectures for Machine Perception*, pp. 217–221, Padovay, Italy, 2000.

[22] J. Vuillemin, P. Bertin, D. Roncin, M Shand, H. Touati, and Ph. Boucard, "Programmable Active Memories: Reconfigurable Systems Come of Age," *IEEE Transactions on VLSI Systems*, vol. 4, no. 1, pp. 56–69, Mar. 1996.

[23] T. Lewis, M. Perkowski, and L. Jozwiak, "Learning in Hardware: Architecture and Implementation of an FPGA-Based Rough set Machine," *Proceedings of 25th EUROMICRO Conference*, pp. 326–334, Milan, Italy, 1999.

[24] M. Perkowski, A. Chebotarev, and A. Mishchenko, "Evolvable Hardware or Learning Hardware? Induction of State Machines from Temporal Logic Constraints," *Proceedings of the First NASA/DoD Workshop on Evolvable Hardware*, pp. 129–138, Pasadena, CA, 1999.

[25] B. Hutchings and B. Nelson, "Developing and Debugging FPGA Applications in Hardware with JHDL," *Conference Record of the 33rd Asilomar Conference on Signals, Systems, and Computers*, vol. 1, pp. 554–558, Oct. 1999.

[26] E. Waingold, M. Taylor, D. Srikrishna, V. Sarkar, W. Lee, V. Lee, , and J. Kim, "Baring it All to Software: RAW Machines," *IEEE Computer*, vol. 30, no. 9, pp. 86–93, Sep. 1997.

[27] M. Weinhardt and W. Luk, "Pipelined Vectorization for Reconfigurable Systems," *Proceedings of the 7th Annual IEEE Symposium on Field-Programmable Custom Computing Machines*, pp. 52–62, Apr. 1999.

[28] T. J. Callahan, J. R. Hauser, and J. Wawrzynek, "The Garp Architecture and C compiler," *IEEE Computer*, vol. 33, no. 4, pp. 62–69, Apr. 2000.

[29] S. C. Goldstein, H. Schmit, M. Budiu, S. Cadambi, M. Moe, and R. R. Taylor, "PipeRench: a Reconfigurable Architecture and Compiler," *IEEE Computer*, vol. 33, no. 4, pp. 70–77, Apr. 2000.

[30] D. C. Cronquist, C. Fisher, M. Figueroa, P. Franklin, and C. Ebeling, "Architecture Design of Reconfigurable Pipelined Datapaths," *Proceedings of the 20th Anniversary Conference on Advanced Research in VLSI*, pp. 23–40, Mar. 1999.

[31] M. Gokhale and D. Gomersall, "High Level Compilation for Fine Grained FPGAs," *Proceedings of the 5th Annual IEEE Symposium on Field-Programmable Custom Computing Machines*, pp. 165–173, Apr. 1997.

[32] E. Mirsky and A. DeHon, "MATRIX: a Reconfigurable Computing Architecture with Configurable Instruction Distribution and Deployable Resources," *Proceedings of the IEEE Symposium on FPGAs for Custom Computing Machines*, pp. 166–157, Apr. 1996.

[33] S. Hauck and A. Agarwal, "Software Technologies for Reconfigurable Systems," *Northwestern University, Dep. of ECE, Technical Report*, 1996.

[34] P. M. Athanas and H. F. Silverman, "Processor Reconfiguration Through Instructions-Set Metamorphosis," *IEEE Computer*, vol. 26, no. 3, pp. 11–18, Mar. 1993.

[35] M Wazlowski, L. Agarwal, T. Lee, A. Smith, E. Lam, P. Athanas, H. Silverman, and S. Ghosh, "PRISM-II Compiler and Architecture," *First IEEE Workshop on FPGAs for Custom Computing Machines*, pp. 9–16, 1993.

[36] L. Agarwal, M Wazlowski, and S. Ghosh, "An Asynchronous Approach to Efficient Execution of Programs on Adaptive Architectures Utilizing FPGAs," *Second IEEE Workshop on FPGAs for Custom Computing Machines*, pp. 101–110, 1994.

[37] D. Wo and K. Forward, “Compiling to the Gate Level for a Reconfigurable Co-Processor,” *1985 International Symposium On Circuits and Systems*, pp. 2–4, 1985.

[38] D. Galloway, “The Transmogrifier C Hardware Description Language and Compiler for FPGAs,” *Third IEEE Workshop on FPGAs for Custom Computing Machines*, 1995.

[39] T. Isshiki and W. W.-M. Dai, “High-Level Bit-Serial Datapath Synthesis for Multi-FPGA Systesm,” *ACM/SIGDA Symposium on Field Programmable Gate Arrays*, pp. 167–173, 1995.

[40] J. B. Peterson, R. B. O’Connor, and P. M. Athanas, “Scheduling and Partitioning ANSI-C Programs onto Multi-FPGA CCM Architectures,” *Fourth IEEE Workshop on FPGAs for Custom Computing Machines*, 1996.

[41] D. A. Clark and B. L. Hutchings, “Supporting FPGA Microprocessors Through Retargetable Software Tools,” *Fourth IEEE Workshop on FPGAs for Custom Computing Machines*, pp. 195–205, 1996.

[42] T. Yamauchi, S. Nakaya, and N. Kajihara, “SOP: A Reconfigurable Massively Parallel System and Its Control-Data-Flow Based Compiling Method,” *Fourth IEEE Workshop on FPGAs for Custom Computing Machines*, 1996.

[43] C. Iseil and E. Sanchez, “Spyder: A SURE, SUperscalar and REconfigurable Processor,” *Journal of Supercomputing*, vol. 9, pp. 231–252, 1993.

[44] M. F. Dossis, J. M. Noras, and G. J. Porter. *Custom Co-processor Compilation*, Abingdon EEE and CS Books, Oxford, England, 1994.

[45] I. Page and W. Luk, *Compiling Occam into FPGAs* Abingdon EE and CS Books, pp. 271–283, 1991.

[46] W. Luk, D. Ferguson, and I. Page, *Structured Hardware Compilation of Parallel Programs*, Abingdon EEE and CS Books, Oxford, England, 1994.

[47] M. Gokhale and R. Minnich, "FPGA Programming in a Data Parallel C," *First IEEE Workshop on FPGAs for Custom Computing Machines*, pp. 94–102, 1993.

[48] S. A. Guccione and M. J. Gonzalez, "A Data-Parallel Programming Model for Reconfigurable Architectures," *First IEEE Workshop on FPGAs for Custom Computing Machines*, pp. 79–87, 1993.

[49] B. Pottier and J. Llopis, "Revisiting Smalltalk-80: A Logic Generator for FPGAs," *Fourth IEEE Workshop on FPGAs for Custom Computing Machines*, 1996.

[50] R. Razdan, *PRISC: Programmable Reduced Instruction Set Computers*, PhD thesis, Harvard University, Cambridge, MA, 1994.

[51] S. Singh, "Architectural Descriptions for FPGA Circuits," *Proceedings of the Third IEEE Workshop on FPGAs for Custom Computing Machines*, pp. 145–154, Apr. 1995.

[52] G. Brown and A. Wenban, "A Software Development System for FPGA-Based Data Acquisition Systems," *Fourth IEEE Workshop on FPGAs for Custom Computing Machines*, 1996.

[53] B. Radunovic and V. Milutinovic, "A Survey of Reconfigurable Computing Architectures," *Proceedings of FPL 98 Eigth International Workshop on Field Programmable Logic and Applications*, Tallin, Estonia, 1998.

[54] B. Radunovic, "An Overview of Advances in Reconfigurable Computing Systems," *Proceedings of 32th Havaii International Conference on System Science*, Havaii, 1999.

[55] R.W. Hartenstein, R. Kress, and H. Reinig, "A Reconfigurable Data-Driven ALU for Xputers," *Proceedings. IEEE Workshop on FPGAs for Custom Computing Machines*, pp. 139–146, Napa Valley, CA, Apr. 1994.

[56] R Kress. *A Fast Reconfigurable ALU for Xputers*, PhD Thesis, Kaiserslautern University, 1996.

[57] M. Gokhale, W. Holmes, A. Kopser, S. Lucas, R. Minnich, D. Sweely, and D. Lopresti, "Building and Using a Highly Parallel Programmable Logic Array," *Computer*, vol. 24, no. 1, pp. 81–89, Jan. 1991.

[58] J. O. Haenni, J. L Beuchat, and E. Sanchez, "RENCO: A Reconfigurable Network Computer," *IEEE Symposium on FPGAs for Custom Computing Machines*, pp. 288–289, Napa Valley, CA, Apr. 1998.

[59] J. R. Hauzer and J. Wawrzynek, "GARP: A MIPS Processor with a Reconfigurable Coprocessor," *Proceedings of the IEEE Symposium on FPGAs for Custom Computing Machines*, pp. 12–21, Apr. 1997.

[60] J. M Arnold et al, "The SPLASH 2," *Proceedings of the 4th Annual ACM Symposium on Parallel Algorithms and Architectures*, pp. 316–324, Jun. 1992.

[61] B. Hutchings, B. Nelson, and M. J. Wirthlin, "Designing and Debugging Custom Computing Applications," *IEEE Design and Test of Computers*, vol. 17, no. 1, pp. 20–28, Mar. 2000.

[62] B. L. Hutchings and B. E. Nelson, "Unifying Simulation and Execution in a Design Environment for FPGA Systems," *IEEE Transactions on Very Large Scale Integration (VLSI) Systems*, vol. 9, no. 1, pp. 201–205, Feb. 2001.

[63] M. Chu, N. Weaver, K. Sulimma, A. Dehon, and J. Wawrzynek, "Object Oriented Circuit-Generators in Java," *Proceedings of the IEEE Symposium on FPGAs for Custom Computing Machines*, pp. 158–166, Apr. 1998.

[64] J. Babb, M. Frank, V. Lee, E. Waingold, R. Barua, M. Taylor, and J. Kim, "The RAW Benchmark Suite: Computation Structures for General Purpose Computing," *Proceedings of he 5th Annual IEEE Symposium on Field-Programmable Custom Computing Machines*, pp. 134–143, Apr. 1997.

[65] R. Barua, W. Lee, S. Amarasinghe, and A. Agarwal, "Compiler Support for Scalable and Efficient Memory Systems," *IEEE Transactions on Computers*, vol. 50, no. 11, pp. 1234–1247, Nov. 2001.

[66] C. A. Moritz, D. Yeung, and A. Agarwal, "SimpleFit: A Framework for Analyzing Design Trade-Offs in Raw Architectures," *IEEE Transactions on Parallel and Distributed Systems*, vol. 12, no. 7, pp. 730–742, Jul. 2001.

[67] M. Weinhardt and Wayne Luk, "Pipeline Vectorization," *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, vol. 20, no. 2, pp. 234–248, Feb. 2001.

[68] I. Page, "Closing the Gap Between Hardware and Software: Hardware-Software Cosynthesis at Oxford," *EE Colloquium on Hardware-Software Cosynthesis for Reconfigurable Systems*, pp. 2/1–2/11, Feb. 1996.

[69] I. Page and R. Dettmer, "Software to Silicon," *IEE Review*, vol. 46, no. 5, pp. 15–19, Sep. 2000.

[70] M. Fleury, R. P. Self, and A. C. Downtown, "Hardware Compilation for software Engineers: An ATM Example," *IEE Proceedings - Software*, vol. 148, no. 1, pp. 31–42, Feb. 2001.

[71] K. Buchenrieder, A. Pyttel, and A. Sedlmeier, "A Powerful System Design Methodology Combining OCAPI and Handel-C for Concept Engineering," *Proceedings of Design, Automation and Test in Europe Conference and Exhibition*, pp. 870 –874, Mar. 2002.

[72] R. P. Wilson, R. S. French, C. S. Wilson, S. P. Amarasinghe, J-A. M. Anderson, S. W. K. Tjiang, S-W. Liao, C-W. Tseng, M. W. Hall, M. S. Lam, and J. L. Hennessy, "SUIF: An Infrastructure for Research on Parallelizing and Optimizing Compilers," *SIGPLAN Notices*, vol. 29, no. 12, pp. 31–37, 1994.

[73] H. Schmit, D. Whelihan, A. Tsai, M. Moe, B. Levine, and R. R. Taylor, "PipeRanch: A Virtualized Programmable Datapath in 0.18 Micron Technology," *IEEE 2002 Custom Integrated Circuits Conference*, pp. 63–66, 2002.

[74] S. C. Goldstein, H. Schmit, M. Budiu, S. Cadambi, M. Moe, and R. R. Taylor, "PipeRench: A Reconfigurable Architecture and Compiler," *Computer*, vol. 33, no. 4, pp. 70–77, 2000.

[75] Y. C. Chou, P. Pillai, H. Schmit, and J. P. Shen, “PipeRench Implementation of the Instruction Path Coprocessor,” *International Symposium on Microarchitecture*, pp. 147–158, 2000.

[76] S. C. Goldstein, H. Schmit, M. Moe, M. Budiu, S. Cadambi, R. R. Taylor, and R. Laufer, “Piperench: A Coprocessor for Streaming Multimedia Acceleration,” *ISCA*, pp. 28–39, 1999.

[77] H. Schmit et al, “Pipeline Reconfigurable FPGAs,” *Journal of VLSI Signal Processing*, pp. 1–18, 2000.

[78] A. A. Duncan, D. C. Hendry, and P. Gray, “An Overview of the Cobra-abs High Level Synthesis System for Multi-FPGA Systems,” *IEEE Proceedings of FPGAs for Custom Computing Machines*, pp. 106–115, 1998.

[79] A. A. Duncan, D. C. Hendry, and P. Gray, “The COBRA-ABS High-Level Synthesis System for Multi-FPGA Custom Computing Machines,” *IEEE Transactions Very Large Scale Integration (VLSI) Systems*, vol. 9, no. 1, pp. 218–223, 2001.

[80] IEEE Standards Board, “IEEE Standard for Binary Floating-Point Arithmetic,” Aug. 1985.

[81] I. Sahin, C. Gloster, and C. Doss, “Feasibility of Floating-Point Arithmetic in Reconfigurable Computing Systems,” *Military and Aerospace Applications of Programmable Devices and Technology Conference*, Washington, DC., Sep. 2000.

[82] I. Sahin and C. Gloster, "Floating-Point Modules Targeted for Use with RC Compilation Tools," *Earth Science Technology Conference (ESTC) 2001*, College Park, MD, Aug. 2001.

[83] *WildForce Reference Manual*, Annapolis Micro Systems Inc., 1997. Rev 3.4.

[84] *Xilinx Data Book 2000*, Oct. 1999.

[85] J. L. Hennessy and D. A. Patterson, *Computer Architecture a Quantitative Approach*. Morgan Kaufmann Publisher, Inc., San Francisco, CA, 1996.

[86] Jutta Degener, "ANSI C Grammar: Lex Specification," 2002. http://www.lysator.liu.se/c/ANSI-C-grammar-l.htm.

[87] *WildCard Reference Manual*, Annapolis Micro Systems Inc., 1999.

[88] K. Li, Y. Pan, and S.Q. Zheng, "Fast and Processor Efficient Parallel Matrix Multiplication Algorithms on a Linear Array with a Reconfigurable Pipelined Bus System," *IEEE Transactions on Parallel and Distributed Systems*, vol. 9, no. 8, pp. 705–720, Aug. 1998.

[89] N. Nikolaidis and I. Pitas, *3-D Image Processing Algorithms*, John Wiley and Sons, Inc., New York, NY, 2001.

[90] M. Seul, L. O'Gorman, and M. J. Sammon, *Practical Algorithms for Image Analysis*, Cambridge University Press, New York, NY, 2000.

Appendix A

Lex Specification for the Scanner

```
D                       [0-9]
L                       [a-zA-Z_]
H                       [a-fA-F0-9]
E                       [Ee][+-]?{D}+
FS                      (f|F|l|L)
IS                      (u|U|l|L)*

%{
#include <stdio.h>
#include <stdlib.h>
#include "y.tab.h"
FILE *fpin, *fpout;

struct LexType{
   int  Token;
   int  Code;
   char *TokenText;
};

main()
{  char ch;
   struct LexType  a;
   fpin = fopen("test.c","r");
   fpout = fopen ("test.lex","w");
   yyin = fpin;
   yyout = fpout;
   printf("Hit enter to start scanning\n");
   scanf("%c",&ch);
   while (!feof(fpin)){
      a = yylex();
      /*if (yytext!="\0" && yytext != "\t" &&
            yytext != "\n" && yytext != "\v" &&
            yytext != "\f")*/
      fprintf(fpout,"\n");
      /*printf("yytext = %s\n",yytext);*/
      printf("Text: %20s Code: %4i Token: %4i\n",a.TokenText,a.Code,a.Token);
   }
   printf("End of scanning\n");
}

void Count();
%}

%%
"/*"        {Comment(); LexReturn->Token = -1; LexReturn->Code = WHT;
             strcpy(LexReturn->TokenText,"/*Comment*/"); return LexReturn;}
"//"        {Comment1(); LexReturn->Token = -1; LexReturn->Code = WHT;
             strcpy(LexReturn->TokenText,"/*Comment*/"); return LexReturn;}
"auto"      {Count(); LexReturn->Token = (int)AUTO; LexReturn->Code = RES;
             strcpy(LexReturn->TokenText,yytext); return LexReturn;}
"break"     {Count(); LexReturn->Token = (int)BREAK; LexReturn->Code = RES;
             strcpy(LexReturn->TokenText,yytext); return LexReturn;}
"case"      {Count(); LexReturn->Token = (int)CASE; LexReturn->Code = RES;
             strcpy(LexReturn->TokenText,yytext); return LexReturn;}
"char"      {Count(); LexReturn->Token = (int)CHAR; LexReturn->Code = RES;
             strcpy(LexReturn->TokenText,yytext); return LexReturn;}
"const"     {Count(); LexReturn->Token = (int)CONST; LexReturn->Code = RES;
             strcpy(LexReturn->TokenText,yytext); return LexReturn;}
"continue"  {Count(); LexReturn->Token = (int)CONTINUE; LexReturn->Code = RES;
             strcpy(LexReturn->TokenText,yytext); return LexReturn;}
"default"   {Count(); LexReturn->Token = (int)DEFAULT; LexReturn->Code = RES;
             strcpy(LexReturn->TokenText,yytext); return LexReturn;}
"do"        {Count(); LexReturn->Token = (int)DO; LexReturn->Code = RES;
             strcpy(LexReturn->TokenText,yytext); return LexReturn;}
"double"    {Count(); LexReturn->Token = (int)DOUBLE; LexReturn->Code = RES;
             strcpy(LexReturn->TokenText,yytext); return LexReturn;}
"else"      {Count(); LexReturn->Token = (int)ELSE; LexReturn->Code = RES;
             strcpy(LexReturn->TokenText,yytext); return LexReturn;}
"enum"      {Count(); LexReturn->Token = (int)ENUM; LexReturn->Code = RES;
             strcpy(LexReturn->TokenText,yytext); return LexReturn;}
"extern"    {Count(); LexReturn->Token = (int)EXTERN; LexReturn->Code = RES;
             strcpy(LexReturn->TokenText,yytext); return LexReturn;}
"float"     {Count(); LexReturn->Token = (int)FLOAT; LexReturn->Code = RES;
             strcpy(LexReturn->TokenText,yytext); return LexReturn;}
"for"       {Count(); LexReturn->Token = (int)FOR; LexReturn->Code = RES;
             strcpy(LexReturn->TokenText,yytext); return LexReturn;}
"goto"      {Count(); LexReturn->Token = (int)GOTO; LexReturn->Code = RES;
             strcpy(LexReturn->TokenText,yytext); return LexReturn;}
"if"        {Count(); LexReturn->Token = (int)IF; LexReturn->Code = RES;
             strcpy(LexReturn->TokenText,yytext); return LexReturn;}
"int"       {Count(); LexReturn->Token = (int)INT; LexReturn->Code = RES;
             strcpy(LexReturn->TokenText,yytext); return LexReturn;}
"long"      {Count(); LexReturn->Token = (int)LONG; LexReturn->Code = RES;
             strcpy(LexReturn->TokenText,yytext); return LexReturn;}
"register"  {Count(); LexReturn->Token = (int)REGISTER; LexReturn->Code = RES;
             strcpy(LexReturn->TokenText,yytext); return LexReturn;}
"return"    {Count(); LexReturn->Token = (int)RETURN; LexReturn->Code = RES;
             strcpy(LexReturn->TokenText,yytext); return LexReturn;}
"short"     {Count(); LexReturn->Token = (int)SHORT; LexReturn->Code = RES;
             strcpy(LexReturn->TokenText,yytext); return LexReturn;}
"signed"    {Count(); LexReturn->Token = (int)SIGNED; LexReturn->Code = RES;
             strcpy(LexReturn->TokenText,yytext); return LexReturn;}
"sizeof"    {Count(); LexReturn->Token = (int)SIZEOF; LexReturn->Code = RES;
             strcpy(LexReturn->TokenText,yytext); return LexReturn;}
"static"    {Count(); LexReturn->Token = (int)STATIC; LexReturn->Code = RES;
             strcpy(LexReturn->TokenText,yytext); return LexReturn;}
"struct"    {Count(); LexReturn->Token = (int)STRUCT; LexReturn->Code = RES;
             strcpy(LexReturn->TokenText,yytext); return LexReturn;}
"switch"    {Count(); LexReturn->Token = (int)SWITCH; LexReturn->Code = RES;
             strcpy(LexReturn->TokenText,yytext); return LexReturn;}
```

```
"typedef"   {Count(); LexReturn->Token = (int)TYPEDEF; LexReturn->Code = RES;
             strcpy(LexReturn->TokenText,yytext); return LexReturn;}
"union"     {Count(); LexReturn->Token = (int)UNION; LexReturn->Code = RES;
             strcpy(LexReturn->TokenText,yytext); return LexReturn;}
"unsigned"  {Count(); LexReturn->Token = (int)UNSIGNED; LexReturn->Code = RES;
             strcpy(LexReturn->TokenText,yytext); return LexReturn;}
"void"      {Count(); LexReturn->Token = (int)VOID; LexReturn->Code = RES;
             strcpy(LexReturn->TokenText,yytext); return LexReturn;}
"volatile"  {Count(); LexReturn->Token = (int)VOLATILE; LexReturn->Code = RES;
             strcpy(LexReturn->TokenText,yytext); return LexReturn;}
"while"     {Count(); LexReturn->Token = (int)WHILE; LexReturn->Code = RES;
             strcpy(LexReturn->TokenText,yytext); return LexReturn;}
{L}({L}|{D})* {Count(); LexReturn->Token = (int)IDENTIFIER; LexReturn->Code = IDENT;
             strcpy(LexReturn->TokenText,yytext); return LexReturn;}
0[xX]{H}+{IS}? {Count(); LexReturn->Token = (int)CONSTANT; LexReturn->Code = HEXC;
             strcpy(LexReturn->TokenText,yytext); return LexReturn;}
0{D}+{IS}?  {Count(); LexReturn->Token = (int)CONSTANT; LexReturn->Code = INTC;
             strcpy(LexReturn->TokenText,yytext); return LexReturn;}
{D}+{IS}?   {Count(); LexReturn->Token = (int)CONSTANT; LexReturn->Code = INTC;
             strcpy(LexReturn->TokenText,yytext); return LexReturn;}
'(\\.|[^\\'])+' {Count(); LexReturn->Token = (int)CONSTANT; LexReturn->Code = UNKNW;
             strcpy(LexReturn->TokenText,yytext); return LexReturn;}
{D}+{E}{FS}? {Count(); LexReturn->Token = (int)CONSTANT; LexReturn->Code = REALC;
             strcpy(LexReturn->TokenText,yytext); return LexReturn;}
{D}*"."{D}+({E})?{FS}?
            {Count(); LexReturn->Token = (int)CONSTANT; LexReturn->Code = EXPC;
             strcpy(LexReturn->TokenText,yytext); return LexReturn;}
{D}+"."{D}*({E})?{FS}?
            {Count(); LexReturn->Token = (int)CONSTANT; LexReturn->Code = EXPC;
             strcpy(LexReturn->TokenText,yytext); return LexReturn;}
\"(\\.|[^\\"])*\"
            {Count(); LexReturn->Token = (int)STRING_LITERAL; LexReturn->Code = STRC;
             strcpy(LexReturn->TokenText,yytext); return LexReturn;}
">>="       {Count(); LexReturn->Token = (int)RIGHT_ASSIGN; LexReturn->Code = ASGN;
             strcpy(LexReturn->TokenText,yytext); return LexReturn;}
"<<="       {Count(); LexReturn->Token = (int)LEFT_ASSIGN; LexReturn->Code = ASGN;
             strcpy(LexReturn->TokenText,yytext); return LexReturn;}
"+="        {Count(); LexReturn->Token = (int)ADD_ASSIGN; LexReturn->Code = ASGN;
             strcpy(LexReturn->TokenText,yytext); return LexReturn;}
"-="        {Count(); LexReturn->Token = (int)SUB_ASSIGN; LexReturn->Code = ASGN;
             strcpy(LexReturn->TokenText,yytext); return LexReturn;}
"*="        {Count(); LexReturn->Token = (int)MUL_ASSIGN; LexReturn->Code = ASGN;
             strcpy(LexReturn->TokenText,yytext); return LexReturn;}
"/="        {Count(); LexReturn->Token = (int)DIV_ASSIGN; LexReturn->Code = ASGN;
             strcpy(LexReturn->TokenText,yytext); return LexReturn;}
"%="        {Count(); LexReturn->Token = (int)MOD_ASSIGN; LexReturn->Code = ASGN;
             strcpy(LexReturn->TokenText,yytext); return LexReturn;}
"&="        {Count(); LexReturn->Token = (int)AND_ASSIGN; LexReturn->Code = ASGN;
             strcpy(LexReturn->TokenText,yytext); return LexReturn;}
"^="        {Count(); LexReturn->Token = (int)XOR_ASSIGN; LexReturn->Code = ASGN;
             strcpy(LexReturn->TokenText,yytext); return LexReturn;}
"|="        {Count(); LexReturn->Token = (int)OR_ASSIGN; LexReturn->Code = ASGN;
             strcpy(LexReturn->TokenText,yytext); return LexReturn;}
">>"        {Count(); LexReturn->Token = (int)RIGHT_OP; LexReturn->Code = OPRT;
             strcpy(LexReturn->TokenText,yytext); return LexReturn;}
"<<"        {Count(); LexReturn->Token = (int)LEFT_OP; LexReturn->Code = OPRT;
             strcpy(LexReturn->TokenText,yytext); return LexReturn;}
"++"        {Count(); LexReturn->Token = (int)INC_OP; LexReturn->Code = OPRT;
             strcpy(LexReturn->TokenText,yytext); return LexReturn;}
```

```
"--"        {Count(); LexReturn->Token = (int)DEC_OP; LexReturn->Code = OPRT;
             strcpy(LexReturn->TokenText,yytext); return LexReturn;}
"->"        {Count(); LexReturn->Token = (int)PTR_OP; LexReturn->Code = OPRT;
             strcpy(LexReturn->TokenText,yytext); return LexReturn;}
"&&"        {Count(); LexReturn->Token = (int)AND_OP; LexReturn->Code = OPRT;
             strcpy(LexReturn->TokenText,yytext); return LexReturn;}
"||"        {Count(); LexReturn->Token = (int)OR_OP; LexReturn->Code = OPRT;
             strcpy(LexReturn->TokenText,yytext); return LexReturn;}
"<="        {Count(); LexReturn->Token = (int)LE_OP; LexReturn->Code = OPRT;
             strcpy(LexReturn->TokenText,yytext); return LexReturn;}
">="        {Count(); LexReturn->Token = (int)GE_OP; LexReturn->Code = OPRT;
             strcpy(LexReturn->TokenText,yytext); return LexReturn;}
"=="        {Count(); LexReturn->Token = (int)EQ_OP; LexReturn->Code = OPRT;
             strcpy(LexReturn->TokenText,yytext); return LexReturn;}
"!="        {Count(); LexReturn->Token = (int)NE_OP; LexReturn->Code = OPRT;
             strcpy(LexReturn->TokenText,yytext); return LexReturn;}
";"         {Count(); LexReturn->Token = 59; LexReturn->Code = DELI;
             strcpy(LexReturn->TokenText,yytext); return LexReturn;}
"{"         {Count(); LexReturn->Token = 123; LexReturn->Code = CPAR;
             strcpy(LexReturn->TokenText,yytext); return LexReturn;}
"}"         {Count(); LexReturn->Token = 125; LexReturn->Code = CPAR;
             strcpy(LexReturn->TokenText,yytext); return LexReturn;}
","         {Count(); LexReturn->Token = 44; LexReturn->Code = DELI;
             strcpy(LexReturn->TokenText,yytext); return LexReturn;}
":"         {Count(); LexReturn->Token = 58; LexReturn->Code = DELI;
             strcpy(LexReturn->TokenText,yytext); return LexReturn;}
"="         {Count(); LexReturn->Token = 61; LexReturn->Code = ASGN;
             strcpy(LexReturn->TokenText,yytext); return LexReturn;}
"("         {Count(); LexReturn->Token = 40; LexReturn->Code = PAR;
             strcpy(LexReturn->TokenText,yytext); return LexReturn;}
")"         {Count(); LexReturn->Token = 41; LexReturn->Code = PAR;
             strcpy(LexReturn->TokenText,yytext); return LexReturn;}
"["         {Count(); LexReturn->Token = 91; LexReturn->Code = SPAR;
             strcpy(LexReturn->TokenText,yytext); return LexReturn;}
"]"         {Count(); LexReturn->Token = 93; LexReturn->Code = SPAR;
             strcpy(LexReturn->TokenText,yytext); return LexReturn;}
"."         {Count(); LexReturn->Token = 46; LexReturn->Code = DELI;
             strcpy(LexReturn->TokenText,yytext); return LexReturn;}
"&"         {Count(); LexReturn->Token = 38; LexReturn->Code = OPRTS;
             strcpy(LexReturn->TokenText,yytext); return LexReturn;}
"!"         {Count(); LexReturn->Token = 33; LexReturn->Code = OPRTS;
             strcpy(LexReturn->TokenText,yytext); return LexReturn;}
"~"         {Count(); LexReturn->Token = 126; LexReturn->Code = OPRTS;
             strcpy(LexReturn->TokenText,yytext); return LexReturn;}
"-"         {Count(); LexReturn->Token = 45; LexReturn->Code = OPRTS;
             strcpy(LexReturn->TokenText,yytext); return LexReturn;}
"+"         {Count(); LexReturn->Token = 43; LexReturn->Code = OPRTS;
             strcpy(LexReturn->TokenText,yytext); return LexReturn;}
"*"         {Count(); LexReturn->Token = 42; LexReturn->Code = OPRTS;
             strcpy(LexReturn->TokenText,yytext); return LexReturn;}
"/"         {Count(); LexReturn->Token = 47; LexReturn->Code = OPRTS;
             strcpy(LexReturn->TokenText,yytext); return LexReturn;}
"%"         {Count(); LexReturn->Token = 37; LexReturn->Code = OPRTS;
             strcpy(LexReturn->TokenText,yytext); return LexReturn;}
"<"         {Count(); LexReturn->Token = 60; LexReturn->Code = OPRTS;
             strcpy(LexReturn->TokenText,yytext); return LexReturn;}
">"         {Count(); LexReturn->Token = 62; LexReturn->Code = OPRTS;
             strcpy(LexReturn->TokenText,yytext); return LexReturn;}
"^"         {Count(); LexReturn->Token = 94; LexReturn->Code = OPRTS;
```

```
             strcpy(LexReturn->TokenText,yytext); return LexReturn;}
"|"         {Count(); LexReturn->Token = 124; LexReturn->Code = OPRTS;
             strcpy(LexReturn->TokenText,yytext); return LexReturn;}
"?"         {Count(); LexReturn->Token = 63; LexReturn->Code = OPRTS;
             strcpy(LexReturn->TokenText,yytext); return LexReturn;}
"#"         {Count(); LexReturn->Token = 35; LexReturn->Code = DIRF;
             strcpy(LexReturn->TokenText,yytext); return LexReturn;}
[ \t\v\n\f] {Count(); LexReturn->Token = -1; LexReturn->Code = WHT;
             strcpy(LexReturn->TokenText,yytext); return LexReturn;}
.           { /* ignore bad characters */ }

%%
yywrap()
{ return(1);
}

Comment()
{  char c, c1;
   loop:
   while ((c = lex_input()) != '*' && c != 0)
      putchar(c);
   if ((c1 = lex_input()) != '/' && c != 0){
      unput(c1);
      goto loop;
   }
   if (c != 0) putchar(c1);
}

Comment1()
{  char c;
   while ((c = lex_input()) != '\n' && c != 0)
      putchar(c);
}

int column = 0;

void Count()
{  int i;
   for (i = 0; yytext[i] != '\0'; i++)
      if (yytext[i] == '\n') column = 0;
      else if (yytext[i] == '\t') column += 8 - (column % 8);
      else column++;
  ECHO;
}
```

Appendix B

Assembly Language Instruction Set for the Third Generation Module

In the instruction set below, it is assumed that all function cores to execute instructions are located in the first position of the third generation module.

```
//====KEY
VEC : Vector
ADD : Add
SUB : Subtract
MUL : Multiply
DIV : Divide
ACC : Accumulate
DACC : DeAccumulate
PROD : Product
DPROD : DeProduct
K : Constant
R : Register
C : Counter
IMM : Immediate

//====BASIC OPERATIONS              //====Comments
                              Op-   Core    Variable
                              Code  Number  Count
VECADD  2 Yi=Ai+Bi            12    1     2
VECADD  1 Yi=Ai+K             11    1     1
VECSUB  2 Yi=Ai-Bi            12    1     2
VECSUB  1 Yi=Ai-K             11    1     1
VECMUL  2 Yi=Ai*Bi            12    1     2
VECMUL  1 Yi=Ai*K             11    1     1
VECDIV  2 Yi=Ai/Bi            12    1     2
VECDIV  1 Yi=Ai/K             11    1     1
```

```
//====COMBINED INSTRUCTIONS         //====Comments
//====WITHOUT CONSTANT
                              Op-   Core    Input Vector
                              Code  Number  Count
VECADDMUL 3 Yi=(Ai+Bi)*Ci     13    1       3
VECMULADD 3 Yi=(Ai*Bi)+Ci     13    1       3
VECADDDIV 3 Yi=(Ai+Bi)/Ci     13    1       3
VECDIVADD 3 Yi=(Ai/Bi)+Ci     13    1       3
VECADDSUB 3 Yi=(Ai+Bi)-Ci     13    1       3
VECSUBADD 3 Yi=(Ai-Bi)+Ci     13    1       3
VECSUBMUL 3 Yi=(Ai-Bi)*Ci     13    1       3
VECMULSUB 3 Yi=(Ai*Bi)-Ci     13    1       3
VECSUBDIV 3 Yi=(Ai-Bi)/Ci     13    1       3
VECDIVSUB 3 Yi=(Ai/Bi)-Ci     13    1       3
VECMULDIV 3 Yi=(Ai*Bi)/Ci     13    1       3
VECDIVMUL 3 Yi=(Ai/Bi)*Ci     13    1       3

//====COMBINED BASIC                //====Comments
//====OPERATIONS (+,*)        Op-   Core    Input Vector
                              Code  Number  Count
VECADDMUL   3 Yi=(Ai+Bi)*Ci   13    1       3
VECADDMUL   2 Yi=(Ai+Bi)*K    12    1       2
VECADDKMUL  2 Yi=(Ai+K) *Ci   12    1       2
VECADDKMUL  1 Yi=(Ai+K1)*K2   11    1       1
VECMULADD   3 Yi=(Ai*Bi)+Ci   13    1       3
VECMULADD   2 Yi=(Ai*Bi)+K    12    1       2
VECMULKADD  2 Yi=(Ai*K) +Ci   12    1       2
VECMULKADD  1 Yi=(Ai*K1)+K2   11    1       1

//====COMBINED BASIC                //====Comments
//====OPERATIONS (+,/)        Op-   Core    Input Vector
                              Code  Number  Count
VECADDDIV   3 Yi=(Ai+Bi)/Ci   13    1       3
VECADDDIV   2 Yi=(Ai+Bi)/K    12    1       2
VECADDKDIV  2 Yi=(Ai+K) /Ci   12    1       2
VECADDKDIV  1 Yi=(Ai+K1)/K2   11    1       1
VECDIVADD   3 Yi=(Ai/Bi)+Ci   13    1       3
VECDIVADD   2 Yi=(Ai/Bi)+K    12    1       2
VECDIVKADD  2 Yi=(Ai/K) +Ci   12    1       2
VECDIVKADD  1 Yi=(Ai/K1)+K2   11    1       1

//====COMBINED BASIC                //====Comments
//====OPERATIONS (+,-)        Op-   Core    Input Vector
                              Code  Number  Count
VECADDSUB   3 Yi=(Ai+Bi)-Ci   13    1       3
VECADDSUB   2 Yi=(Ai+Bi)-K    12    1       2
VECADDKSUB  2 Yi=(Ai+K) -Ci   12    1       2
VECADDKSUB  1 Yi=(Ai+K1)-K2   11    1       1
VECSUBADD   3 Yi=(Ai-Bi)+Ci   13    1       3
VECSUBADD   2 Yi=(Ai-Bi)+K    12    1       2
VECSUBKADD  2 Yi=(Ai-K) +Ci   12    1       2
VECSUBKADD  1 Yi=(Ai-K1)+K2   11    1       1

//====COMBINED BASIC                //====Comments
//====OPERATIONS (-,*)        Op-   Core    Input Vector
                              Code  Number  Count
VECSUBMUL   3 Yi=(Ai-Bi)*Ci   13    1       3
VECSUBMUL   2 Yi=(Ai-Bi)*K    12    1       2
VECSUBKMUL  2 Yi=(Ai-K) *Ci   12    1       2
VECSUBKMUL  1 Yi=(Ai-K1)*K2   11    1       1
VECMULSUB   3 Yi=(Ai*Bi)-Ci   13    1       3
VECMULSUB   2 Yi=(Ai*Bi)-K    12    1       2
VECMULKSUB  2 Yi=(Ai*K) -Ci   12    1       2
VECMULKSUB  1 Yi=(Ai*K1)-K2   11    1       1

//====COMBINED BASIC                //====Comments
//====OPERATIONS (-,/)         Op-   Core    Input Vector
                              Code  Number  Count
VECSUBDIV   3 Yi=(Ai-Bi)/Ci   13    1       3
VECSUBDIV   2 Yi=(Ai-Bi)/K    12    1       2
VECSUBKDIV  2 Yi=(Ai-K) /Ci   12    1       2
VECSUBKDIV  1 Yi=(Ai-K1)/K2   11    1       1
VECDIVSUB   3 Yi=(Ai/Bi)-Ci   13    1       3
VECDIVSUB   2 Yi=(Ai/Bi)-K    12    1       2
VECDIVKSUB  2 Yi=(Ai/K) -Ci   12    1       2
VECDIVKSUB  1 Yi=(Ai/K1)-K2   11    1       1

//====COMBINED BASIC               //====Comments
//====OPERATIONS (*,/)         Op-   Core    Variable
                              Code  Number  Count
VECMULDIV   3 Yi=(Ai*Bi)/Ci   13    1       3
VECMULDIV   2 Yi=(Ai*Bi)/K    12    1       2
VECMULKDIV  2 Yi=(Ai*K) /Ci   12    1       2
VECMULKDIV  1 Yi=(Ai*K1)/K2   11    1       1
VECDIVMUL   3 Yi=(Ai/Bi)*Ci   13    1       3
VECDIVMUL   2 Yi=(Ai/Bi)*K    12    1       2
VECDIVKMUL  2 Yi=(Ai/K) *Ci   12    1       2
VECDIVKMUL  1 Yi=(Ai/K1)*K2   11    1       1

//====ACCUMULATION                  //====Comments
//====OPERATIONS               Op-   Core    Input Vector
                              Code  Number  Count
VECACC      1 Y=Y+Ai          11    1       1
VECDACC     1 Y=Y-Ai          11    1       1
VECPROD     1 Y=Y*Ai          11    1       1
VECDPROD    1 Y=Y/Ai          11    1       1
VECMULACC   2 Y=Y+(Ai*Bi)     12    1       2
VECMULKACC  1 Y=Y+(Ai*K)      11    1       2
VECMULDACC  2 Y=Y-(Ai*Bi)     12    1       2
VECMULKDACC 1 Y=Y-(Ai*K)      11    1       2
VECDIVACC   2 Y=Y+(Ai/Bi)     12    1       2
VECDIVKACC  1 Y=Y+(Ai/K)      11    1       2
VECDIVDACC  2 Y=Y-(Ai/Bi)     12    1       2
VECDIVKDACC 1 Y=Y-(Ai/K)      11    1       2

//====LOAD/STORE OPERATIONS
LOADR           Reg#, [Address]   //Load to a data register from a given addr.
LOADRIMM        Reg#, Data        //Load to a data register an immediate data
LOADC           Cnt#, [Address]   //Load to a Index Counter from a given addr.
LOADCIMM        Cnt#, Data        //Load to a Index Counter an immediate data.
STORER          Reg#, [Address]   //Store from a data register to a given addr.
STOREC          Cnt#, [Address]   //Store from a Index Counter to a given addr.

//====OTHER INSTRUCTIONS
RUN                  //Start running
PAUSE   Cycl#        //Stop a core or a pipelined unit for a given # of cycles
STOP                 //Stop a core or a pipelined unit for an indefinite time
HALT                 //complately stop the system.
```